The MIDI Specification

MIDI (ie, Musical Instrument Digital Interface) consists of both a simple hardware interface, and a more elaborate transmission protocol.

For a simple, layman's explanation of what MIDI is, read What is MIDI?

The MIDI Specification is published by the MIDI Manufacturer's Association, ie, MMA (although this online document gives you the same information for free, in easier-to-understand language, and in many cases, with even more detail than the official document).

The MIDI Specification

MIDI (ie, Musical Instrument Digital Interface) consists of both a simple hardware interface, and a more elaborate transmission protocol.

For a simple, layman's explanation of what MIDI is, read What is MIDI?

The MIDI Specification is published by the MIDI Manufacturer's Association, ie, MMA (although this online document gives you the same information for free, in easier-to-understand language, and in many cases, with even more detail than the official document).

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.

Schematic
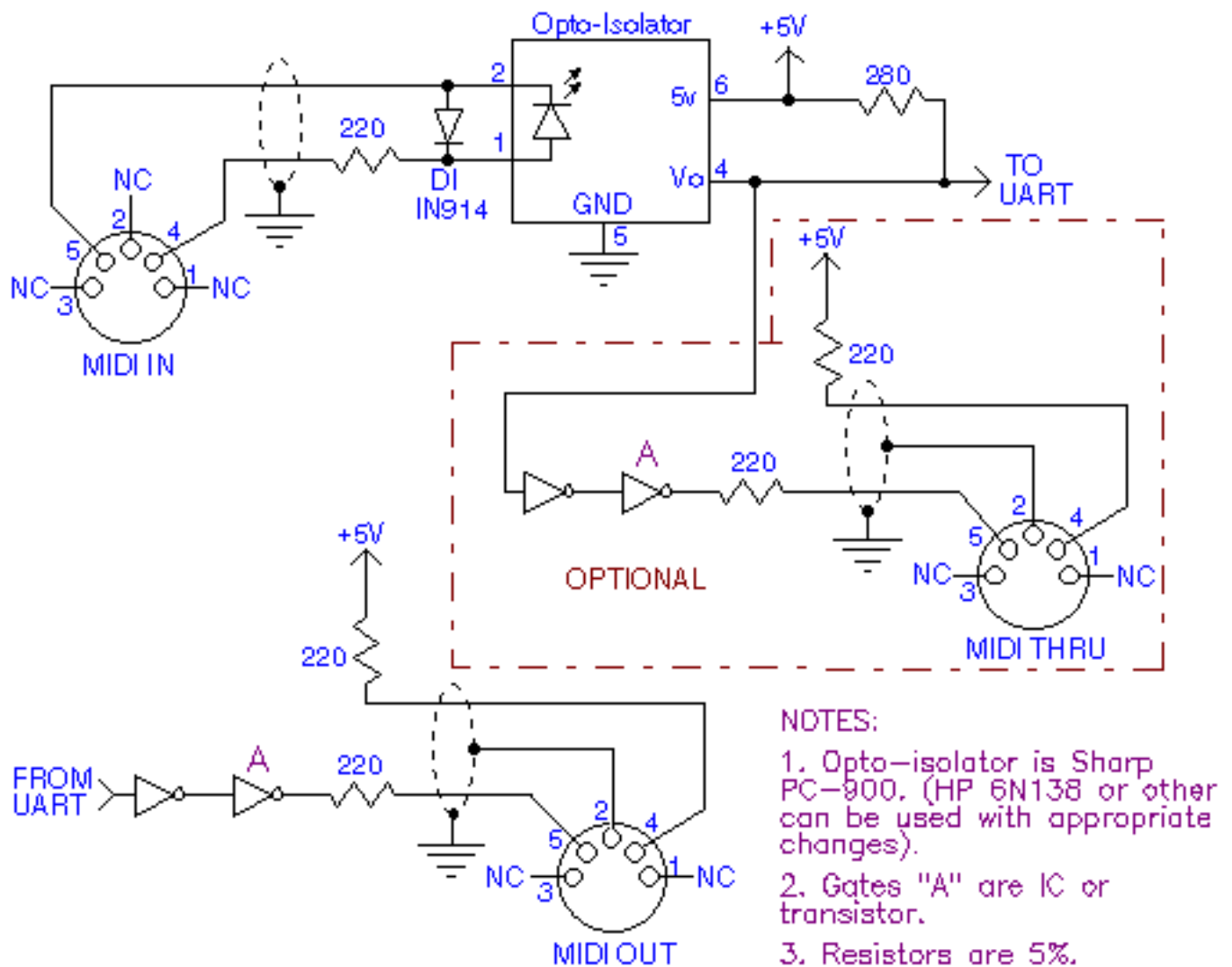
A schematic of a MIDI (IN and OUT) interface

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. (Note that I'm using the C programming language convention of prefacing a value with 0x to indicate hexadecimal).

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 categories. 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 the device's decision 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.

It is recommended that Aftertouch default to controlling the LFO amount (ie, a vibrato effect).

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 are 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 with 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.

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.

Note: In most MIDI literature, the coarse adjust is referred to with the designation "MSB" and the fine adjust is referred to with the designation "LSB". I prefer the terms "coarse" and "fine".

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 (coarse)**
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
95	Phaser Level
96	Data Button increment
97	Data Button decrement
98	Non-registered Parameter (fine)
99	Non-registered Parameter (coarse)
100	Registered Parameter (fine)
101	Registered Parameter (coarse)
120	All Sound Off
121	All Controllers Off
122	Local Keyboard (on/off)
123	All Notes Off
124	Omni Mode Off
125	Omni Mode On
126	Mono Operation
127	Poly Operation

Bank Select

Number: 0 (coarse) 32 (fine)

Affects:

Some MIDI devices have more than 128 Programs (ie, Patches, Instruments, Preset, etc). A MIDI Program Change message supports switching between only 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, 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 <u>then</u> utilized to switch to the specified patch in the new bank. For this reason, Bank Select must be sent <u>before</u> 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. Various manufacturers may number their banks differently. Consult the documentation for a MIDI device to determine what values to use with the Bank Select Coarse and Bank Select Fine messages to select particular banks.

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 a parameter such as what Aftertouch may control. 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)

Affects:

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 volume level for one MIDI channel. If a MultiTimbral device, then each Part has its own volume.

Note: 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.

Expression Controller also may affect the volume.

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.

It is recommended that a device use the volume value in a logarithmic manner, as specified by the following formula if only the coarse value is used:

```
40 log (Volume/127)
```

If both the coarse and fine values are used (and combined into a 14-bit volume), then use the following formula:

```
40 log (Volume/127²)
```

Note: In the above formula, "Volume" may be equal to Channel Volume * Expression.

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.

Pan should effect all notes on the channel, including notes that were triggered prior to the pan message being received, and are still sustaining.

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 <u>Volume Controller</u>). 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 (on a given channel), whereas Expression is used for doing crescendos and decrescendos. By having both a master Volume and sub-Volume (ie, Expression), it makes possible to do crescendos and decrescendos without having to do algebraic calculations to maintain the relative balance between instruments. When Expression is at 100% (ie, the maximum of 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.

Here's how Expression is typically used. Let's assume only the coarse adjust is used (ie, #11) and therefore only 128 steps are possible. Set the Expression for every MIDI channel to one initial value, for example 100. This gives you some leeway to increase the expression percentage (ie, up to 127 which is 100%) or decrease it. Now, set the channel (ie, instrument) "mix" using Volume Controllers. Maybe you'll want the drums louder than the piano, so the former has a Volume Controller value of 110 whereas the latter has a value of 90, for example. Now if, at some point, you want to drop the volumes of both instruments to half of their current Main Volumes, then send Expression values of 64 (ie, 64 represents a 50% volume percentage since 64 is half of 128 steps). This would result in the drums now having an effective volume of 55 and the piano having an effective volume of 45. If you wanted to drop the volumes to 25% of their current Main Volumes, then send Expression values of 32. This would result in the drums now having an effective volume of approximately 27 and the piano having an effective volume of approximately 22. And yet, you haven't had to change their Volume settings, and therefore still maintain that relative mix between the two instruments. So think of Volume Controllers as being the individual faders upon a mixing console. You set up the instrumental balance (ie, mix) using these values. Then you use Expression Controllers as "group faders", whereby you can increase or decrease the volumes of one or more tracks without upsetting the relative balance between them.

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.

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. This controller would be used for some effects parameter for which there is not already another defined controller.

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. This controller would be used for some effects parameter for which there is not already another defined controller.

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 (On/Off)

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) <u>when the pedal is turned on</u>. 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 <u>All Notes Off</u> 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 <u>and</u> 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. This controls how the filter shapes the "brightness" of the sound over time). 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 an overall "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. These controllers would be used to adjust sound parameters for which there are not already other, defined controllers.

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. These 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.

Preface

Before discussing what MIDI is, it is important to understand some basic principles about musical instruments.

There is one thing that all musical instruments do. All musical instruments make a sound under the control of a musician. In other words, at any time, the musician can cause an instrument to start making a sound. For example, a musician can push down a key on a piano to start a sound. Or, he can begin dragging a bow across a violin string to start a sound. Or he can fret and pick a guitar string to start a sound. Let's refer to the action of starting a sound as a "Note On".
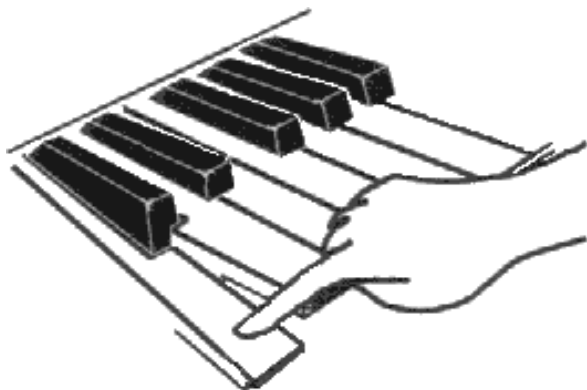
A musician pushes down (and holds down) a key on a keyboard. This sounds some musical note (which continues to sound while the musician continues holding down the key). This single gesture of the musician is known as a Note-On to MIDI.

Most instruments also allow the musician to stop the sound at any given time. For example, the musician can release that piano key, thus stopping the sound. Or, he can stop dragging a bow across the violin string. Or he can release his finger from the guitar fret. Let's refer to the action of stopping a sound as a "Note Off".
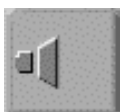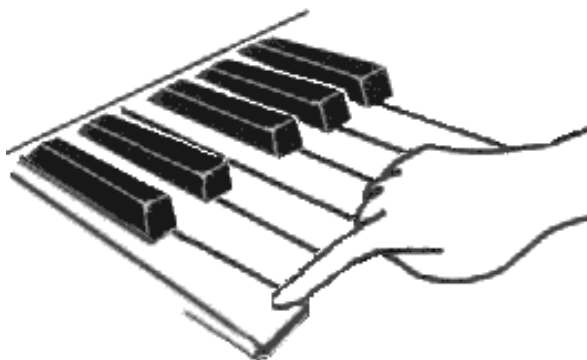
The musician releases the key (that he was holding down) on a keyboard. This stops the musical note from sounding. This single gesture of the musician is known as a Note-Off to MIDI.

Of course, many instruments can play distinct pitches (ie, a musical scale). For example, an acoustic piano has 88 keys, or 88 distinct pitches/notes.

There are other things that many musical instruments may have in common, for example, most instruments can make sounds at various volumes. (ie, They can sound notes at volumes ranging from very soft to very loud). For example, if the pianist pushes down a key with great force, the resulting note will be louder than if he were to gently press down the key.

Introduction

Musicians often want to be able to control electronic instruments remotely or automatically.

Remote control is when a musician plays one musical instrument, and that instrument controls (one or more) other musical instruments.

For example, musicians sometimes find it desirable to combine the sounds of several instruments playing in perfect unison to "thicken" or layer a musical part. The musician wants to blend certain patches upon those instruments. Perhaps he wishes to blend the sax patches upon 5 different instruments to create a more authentic-sounding sax section in a big band. But, since a musician has only two hands and feet, it's not possible to play 5 instruments at once unless he has some method of remote control.

Or, sometimes a musician wants to use only one physical keyboard to control several, separate sound modules. In the old days, every single musical instrument manufactured had its own built-in method of controlling it. For example, an electronic organ, an electronic piano, a string ensemble, a synthesizer, etc, each had its own built-in keyboard. This got to be rather expensive, as the physical keyboard is one of the more expensive parts of an instrument. Also, all of those keyboards tend to take up a lot of space, which is a problem for a gigging musician. So musicians thought *"Wouldn't it be great if I could buy a small box that made organ sounds into which I could plug a physical keyboard? And wouldn't it be great if I could buy other boxes that made piano, string, synth, etc, sounds, into which I could plug that same keyboard? And wouldn't it be great if I could attach them all together simultaneously, and switch the keyboard between playing any of them? I could save money and space. All I need is a standard for remotely controlling all of those boxes with that one keyboard."*

Automatic control is when the musician uses some other device to play a musical instrument as if another musician were playing it. (Such a device is referred to as a Sequencer).

For example, some musicians want to be able to have "backing tracks" in live performance, but they found it too cumbersome, unreliable, and limiting to use prerecorded tapes. They wanted a method that allowed more flexibility, perhaps to do things such as subtlely alter the arrangement live. To achieve this, rather than playing pre-recorded backing tracks, they wanted a method to automatically control their instruments during the performance using a device that could "intelligently" manipulate the arrangement (such as a computer).

So, musicians had a need to remotely or automatically control their musical instruments, and they wanted a method that wasn't tied to one particular manufacturer's product, nor one particular type of instrument. (ie, They wanted a method that worked as well with an electronic piano as it did with a drum box, for example). They wanted a standard that could be useful in controlling <u>any</u> electronic musical device. To satisfy this need, a few music manufacturers got together in mid 1983 and created **MIDI**, which **stands for Musical Instrument Digital Interface**. (For more information about the history of MIDI's development, see The beginnings of MIDI).

<p align="center">**Hardware/Connections**</p>

The visible **MIDI connectors** on an instrument **are female 5-pin DIN jacks**. There are separate jacks for incoming MIDI signals (received from another instrument that is sending MIDI signals), and outgoing MIDI signals (ie, MIDI signals that the instrument creates and sends to another device). The jacks look like these:
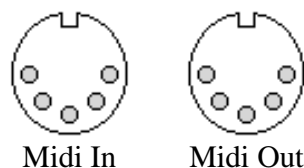
<p align="center">Midi In   Midi Out</p>

You use **MIDI cables** (with male DIN connectors) to **connect the MIDI jacks of various instruments together**, so that those instruments can pass MIDI signals to each other. You connect the MIDI OUT of one instrument to the MIDI IN of another instrument, and vice versa. For example, the following diagram shows the connection between a computer's MIDI interface and a MIDI keyboard that has built-in sounds.
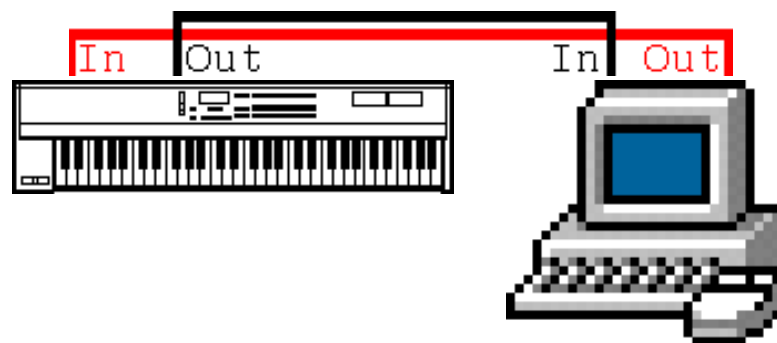
Some instruments have a third MIDI jack labeled "Thru". This is used as if it were an OUT jack, and therefore you attach a THRU jack only to another instrument's IN jack. In fact, the THRU jack is exactly like the OUT jack with one important difference. Any signals that the instrument itself creates (or modifies) are sent out its MIDI OUT jack but not the MIDI THRU jack. Think of the THRU jack as a stream-lined, unprocessed MIDI OUT jack.

MIDI messages

But MIDI is much more than just some jacks on an electronic instrument. In fact, **MIDI is a lot more than just hardware. Mostly, MIDI is an extensive set of "musical commands" which electronic instruments use to control each other**. The MIDI instruments pass these commands to each other over the cables connecting their MIDI jacks together. (ie, Those MIDI signals that I referred to above are these commands).

So, what is a MIDI command? **A MIDI command consists of a few (usually 2 or 3) "data bytes"** (like the data bytes within files that you have on your computer's hard drive). **These data bytes are merely a series of numbers. We refer to one of these groups of numbers as a "message"** (rather than a command). There are many different MIDI messages, and **each one correlates to a specific musical action**. For example, there is a certain group of numbers that tells an instrument to make a sound. (This would be that "Note On" message which I mentioned earlier). There is a different group of numbers that tells an instrument to stop making a sound. (This is the "Note Off" message). One of the numbers within that "Note On" or "Note Off" message tells the instrument which one of its "keys" (ie, notes) to start or stop sounding. (Remember that a piano has 88 notes. MIDI instruments can have a maximum of 128 different notes, although some instruments respond to only messages limited to a smaller range, say 72 notes).

Many electronic instruments not only respond to MIDI messages that they receive (at their MIDI IN jack), they also automatically generate MIDI messages while the musician plays the instrument (and send those messages out their MIDI OUT jacks).

Midi Out → 144 60 64
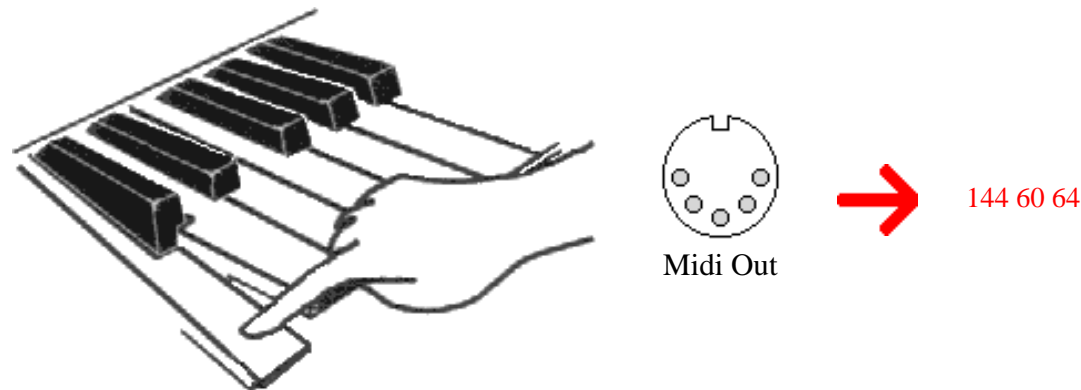
A musician pushes down (and holds down) the middle C key on a keyboard. Not only does this sound a musical note, it also causes a MIDI Note-On message to be sent out of the keyboard's MIDI OUT jack. That message consists of 3 numeric values as shown above.

128 60 64
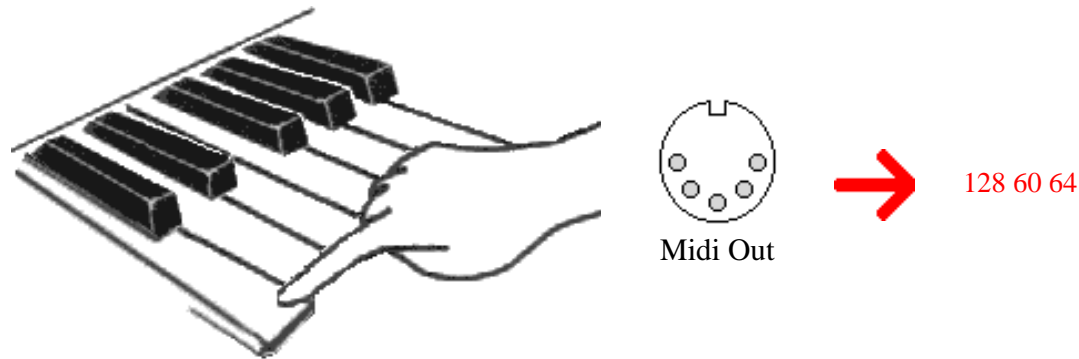
Midi Out

The musician now releases that middle C key. Not only does this stop sounding the musical note, it also causes another message -- a MIDI Note-Off message -- to be sent out of the keyboard's MIDI OUT jack. That message consists of 3 numeric values as shown above. Note that one of the values is different than the Note-On message.

You saw above that when the musician pushed down that middle C note, the instrument sent a MIDI Note On message for middle C out of its MIDI OUT jack. If you were to connect a second instrument's MIDI IN jack to the first instrument's MIDI OUT, then the second instrument would "hear" this MIDI message and sound its middle C too. When the musician released that middle C note, the first instrument would send out a MIDI Note Off message for that middle C to the second instrument. And then the second instrument would stop sounding its middle C note.
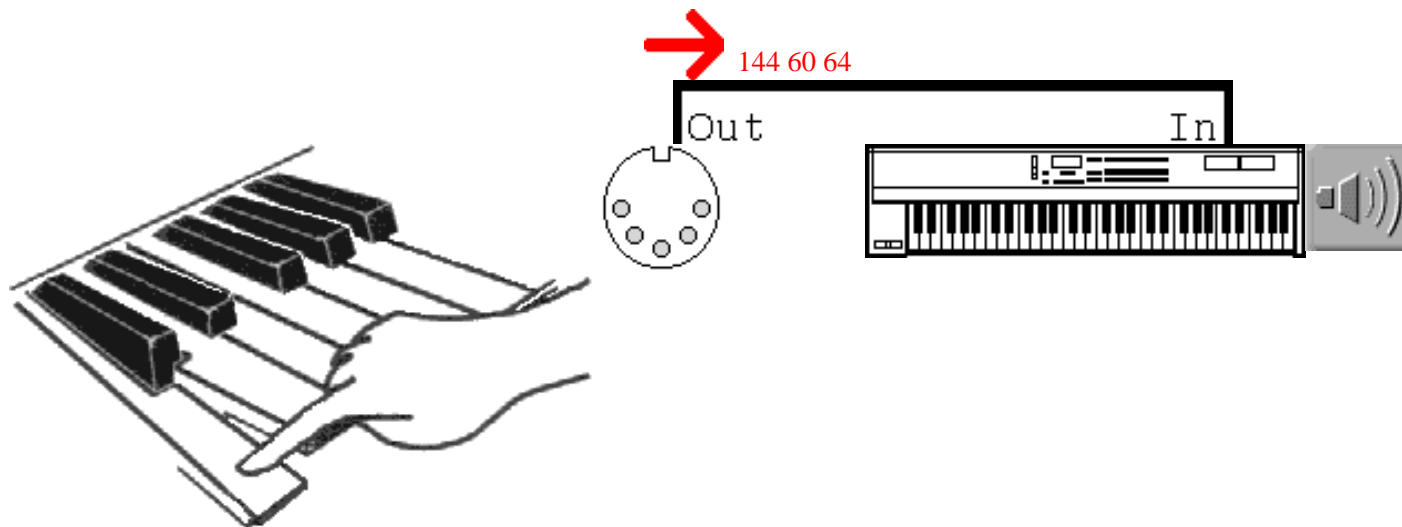
144 60 64

Out   In

A musician pushes down (and holds down) the middle C key on a keyboard. This causes a MIDI Note-On message to be sent out of the keyboard's MIDI OUT jack. That message is received by the second instrument which sounds its middle C in unison.

But MIDI is more than just "Note On" and "Note Off" messages. There are <u>lots</u> more messages. There's a message that tells an instrument to move its pitch wheel and by how much. There's a message that tells the instrument to press or release its sustain pedal. There's a message that tells the instrument to change its volume and by how much. There's a message that tells the instrument to change its patch (ie, maybe from an organ sound to a guitar sound). And of course, these are only a few of the many available messages in the MIDI command set.

And just like with Note On and Note Off messages, these other messages are automatically generated when a musician plays the instrument. For example, if the musician moves the pitch wheel, a pitch wheel MIDI message is sent out of the instrument's MIDI OUT jack. (Of course, the pitch wheel message is a different group of numbers than either the Note On or Note Off messages). What with all of the possible MIDI messages, everything that the musician did upon the first instrument would be echoed upon the second instrument. It would be like he had two left and two right hands that worked in perfect sync.

Daisy-chaining/MIDI Channels

You can attach a MIDI cable from the second instrument's MIDI THRU to a third instrument's MIDI IN, and the second instrument will pass onto the third instrument those messages that the first instrument sent. Now, all 3 instruments can play in unison. You could add a fourth, fifth, sixth, etc, instrument. **We call this "daisy-chaining" instruments**.
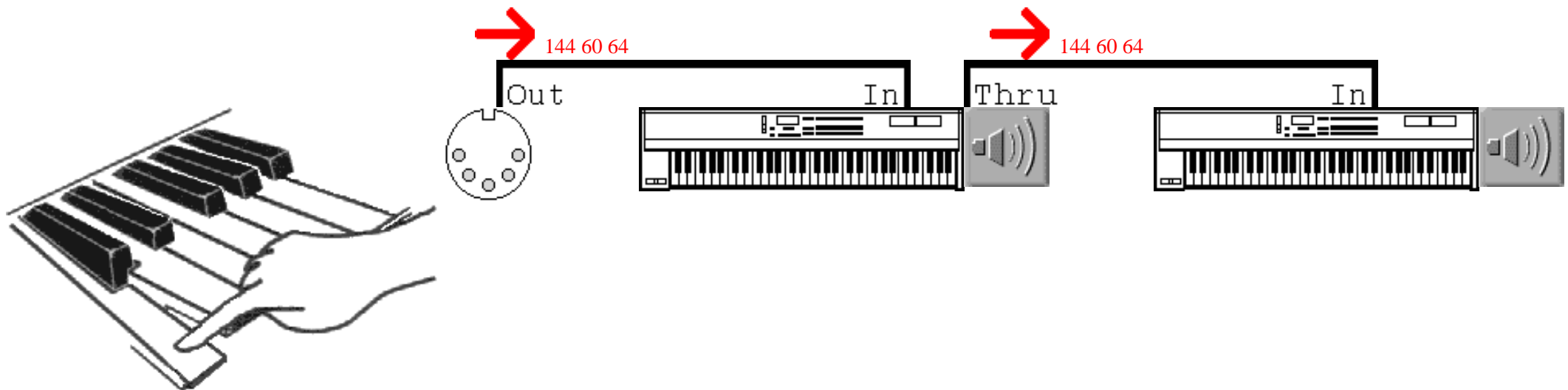
A musician pushes down (and holds down) the middle C key on a keyboard. This causes a MIDI Note-On message to be sent out of the keyboard's MIDI OUT jack. That message is received by the second instrument which sounds its middle C in unison. It is also passed from the second instrument to the third instrument which also sounds its middle C note.

But, daisy-chained instruments don't always have to play in unison either. Each can play its own, individual musical part even though all of the MIDI messages controlling those daisy-chained instruments pass through each instrument. How is this possible? **There are 16 MIDI "channels". They all exist in that one run of MIDI cables that daisy-chain 2 or more instruments** (and perhaps a computer) together. For example, that MIDI message for the middle C note can be sent on channel 1. Or, it can be sent on channel 2. Etc.

Midi Out   144 60 64
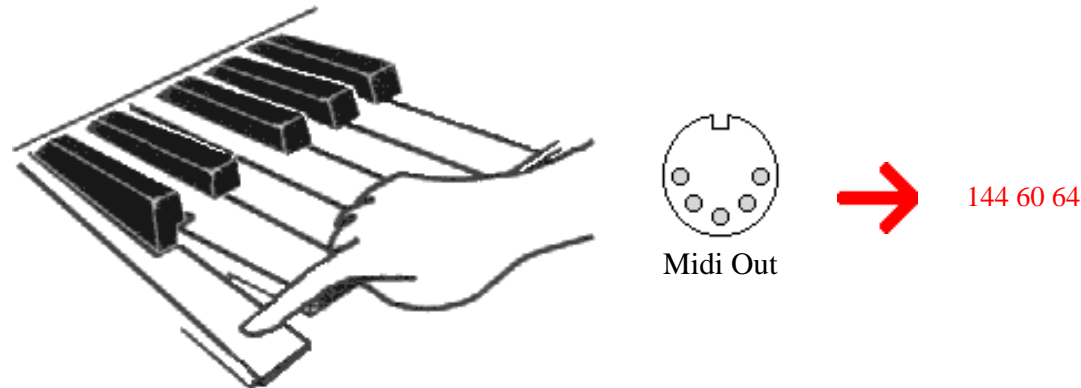
A musician sets his keyboard to send messages upon MIDI channel 1. Then, he pushes down the middle C key. Note the 3 values of the MIDI Note-On message sent out of the keyboard's MIDI OUT jack.

Midi Out   145 60 64
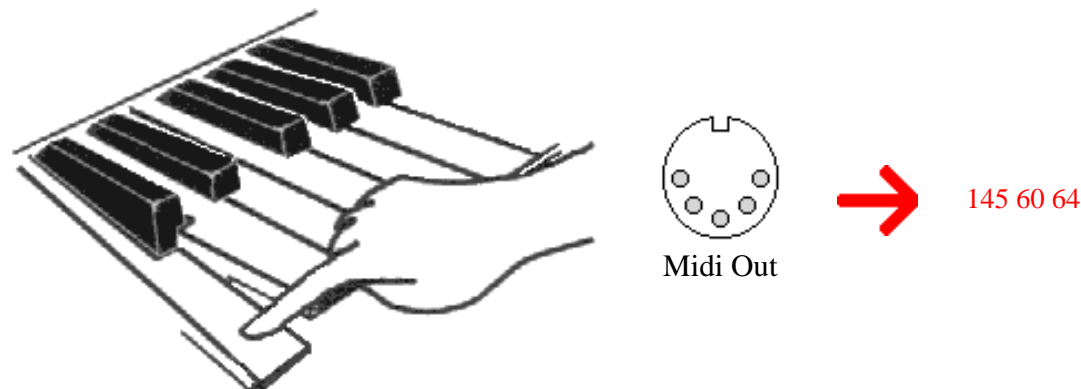
The musician changes the keyboard to send messages upon MIDI channel 2 instead. Then, he pushes down the middle C key. Compare the 3 values of this MIDI Note-On message to the preceding example. Note that the first value is different after the MIDI channel has been changed from 1 to 2.

Most all MIDI instruments allow the musician to select which channel(s) to respond to and which to ignore. For example, if you set an instrument to respond to MIDI messages only on channel 1, and you send a MIDI Note On on channel 2, then the instrument will not play the note. So, if a musician has several instruments daisy-chained, he can set them all to respond to different channels and therefore have independent control over each one.

145 60 64

145 60 64

Out   In   Thru   In
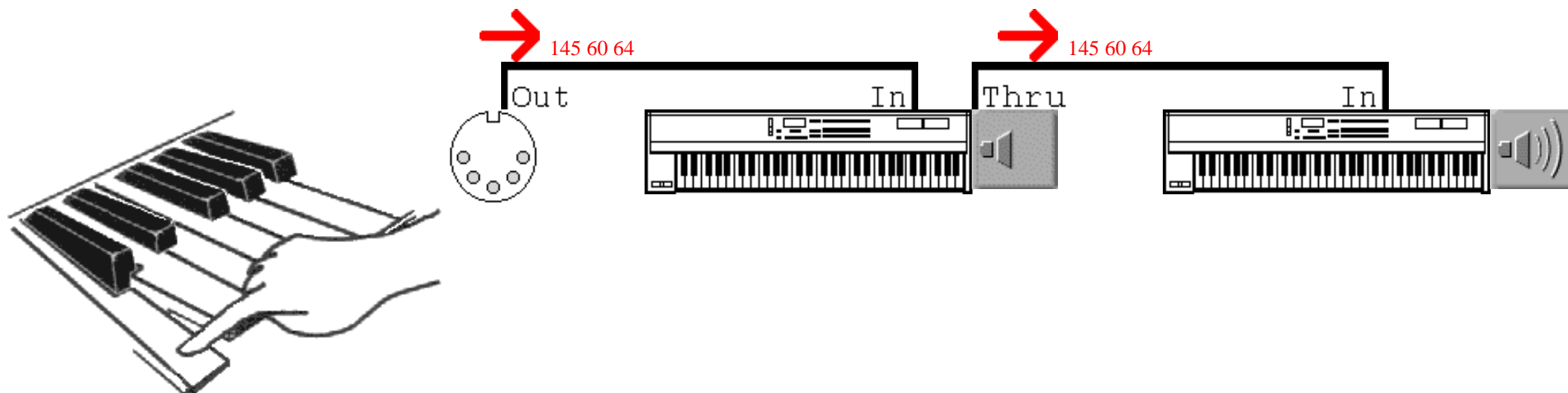
A musician sets his keyboard to send messages upon MIDI channel 2. Then, he pushes down the middle C key. This causes a MIDI Note-On message on MIDI channel 2 to be sent out of the keyboard's MIDI OUT jack. The second instrument is set to MIDI channel 1, so it ignores this MIDI message. (ie, It doesn't play its middle C note). The message is also passed on to the third instrument. This is set to MIDI channel 2, so it does play its middle C note.

Also, when the musican plays each instrument, it generates MIDI messages only on its one channel. So, it's very easy to keep the MIDI data separate for each instrument even though it all goes over one long run of cables. After all, the MIDI Note On message for middle C on channel 1 will be slightly different than the MIDI Note On message for middle C on channel 2.

In a nutshell, that's what MIDI is -- a set of commands that electronic devices digitally pass between their MIDI jacks to tell each other what to do (ie, what actions to perform).

Suffice it to say that MIDI can do everything musical to an electronic instrument that a human being can physically do, and a few things that humans can't do.

The advantages of MIDI

There are two main advantages of MIDI -- it's an easily edited/manipulated form of data, and also it's a compact form of data (ie, produces relatively small data files).

Because MIDI is a digital signal, it's very easy to interface electronic instruments to computers, and then do things with that MIDI data on the computer with software. For example, software can store MIDI messages to the computer's disk drive. Also, the software can playback MIDI messages upon all 16 channels with the same rhythms as the human who originally caused the instrument(s) to generate those messages. So, a musician can digitally record his musical performance and store it on the computer (to be played back by the computer). He does this not by digitizing the actual audio coming out of all of his electronic instruments, but rather by "recording" the MIDI OUT (ie, those MIDI messages) of all of his instruments. Remember that the MIDI messages for all of those instruments go over one run of cables, so if you put the computer at the end, it "hears" the messages from all instruments over just one incoming cable. The great advantage of MIDI is that the "notes" and other musical actions, such as moving the pitch wheel, pressing the sustain pedal, etc, are all still separated by messages on different channels. So the musician can store the messages generated by many instruments in one file, and yet the messages can be easily pulled apart on a per instrument basis because each instrument's MIDI messages are on a different MIDI channel. In other words, when using MIDI, a musician never loses control over every single individual action that he made upon each instrument, from playing a particular note at a particular point, to pushing the sustain pedal at a certain time, etc. The data is all there, but it's put together in such a way that every single musical action can be easily examined and edited.

Contrast this with digitizing the audio output of all of those electronic instruments. If you've got a system that has 16 stereo digital audio tracks, then you can keep each instrument's output separate. But, if you have only 2 digital audio tracks (typically), then you've got to mix the audio signals together before you digitize them. Those instruments' audio outputs don't produce digital signals. They're analog. Once you mix the analog signals together, it would take massive amounts of computation to later filter out separate instruments, and the process would undoubtably be far from perfect. So ultimately, you lose control over each instrument's output, and if you want to edit a certain note of one instrument's part, that's even less feasible.

Furthermore, it typically takes much more storage to digitize the audio output of an instrument than it does to record an instrument's MIDI messages. Why? Let's take an example. Say that you want to record a whole note. With MIDI, there are only 2 messages involved. There's a Note On message when you sound the note, and then the next message doesn't happen until you finally release the note (ie, a Note Off message). That's 6 bytes. In fact, you could hold down that note for an hour, and you're still going to have only 6 bytes; a Note On and a Note Off message. By contrast, if you want to digitize that whole note, you have to be recording all of the time that the note is sounding. So, for the entire time that you hold down the note, the computer is storing literally thousands of bytes of "waveform" data representing the sound coming out of the instrument's AUDIO OUT. You see, with MIDI a musician records his actions (ie, movements). He presses the note down. Then, he does nothing until he releases the note. With digital audio, you record the instrument's sound. So while the instrument is making sound, it must be recorded.

So why not always "record" and "play" MIDI data instead of WAVE data if the former offers so many advantages? OK, for electronic instruments that's a great idea. But what if you want to record someone singing? You can strip search the person, but you're not going to find a MIDI OUT jack on his body. (Of course, I anxiously await the day when scientists will be able to offer "human MIDI retrofits". I'd love to have a built-in MIDI OUT jack on my body, interpreting every one of my motions and thoughts into MIDI messages. I'd have it installed at the back of my neck, beneath my hairline. Nobody would ever see it, but when I needed to use it, I'd just push back my hair and plug in the cable). So, to record that singing, you're going to have to record the sound, and digitizing it into a WAVE file is the best digital option right now. That's why sequencer programs exist that record and play both MIDI and WAVE data, in sync.

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.

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 MMA.

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.

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 MMA 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).

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.

All Controllers Off

Number: 121

Affects:

Resets specific controllers to default states. In general, switches (such as Hold Pedal) are turned off, and 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.

For a minimum implementation of "All Controllers off", it is recommended that the following controllers should be reset to the following values:

Modulation Wheel should be set to 0.

Expression should be set to full value (ie, 127 for both coarse, as well as fine if used).

The pedals Hold, Portamento, Sustenuto, Soft, Legato, and Hold 2 should be set to 0 (ie, Off).

Volume should be set to 100.

Pan should be set to center (ie, 64 for both coarse, as well as fine if used).

Any Non-registered Parameters or Registered Parameters may be reset to default values, such as pitch bend range being reset to +/- 2 semitones.

Additionally, the Pitch Wheel should be reset to center position, and Channel Pressure should be set to 0, although these technically aren't adjusted via MIDI controller messages. If a module supports Aftertouch, then the Aftertouch amount for each note (on the MIDI channel) should also be reset to 0.

A device should <u>not</u> reset its mode (as done via Omni Mode Off, Omni Mode On, Mono Operation, and Poly Operation controllers).

A device should include documentation that lists what is reset by an All Controllers Off message.

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.

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.

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.

Receipt of a Program Change should not cut off any notes that were previously triggered on the channel, and which are still sustaining.

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 (average) 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.

It is recommended that Channel Pressure default to controlling the VCA level (ie, a volume swell/fade effect).

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):

```c
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 (ie, SysEx)

Category: System Common

Purpose

Used to send a large amount of data 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 one model device. For example, a SysEx message might be used to set the feedback level for an operator in a Roland Physical Modeling Synth. This information would likely be useless to an AKAI sample playing device. (By contrast, 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

MMA 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" (ie, actually "aborted" is a better description since such a scenario indicates an abnormal MIDI condition). For example, if a 0x90 happened to be sent sometime after a 0xF0 (but before the 0xF7), then the SysEx message would be considered aborted 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 (ie, the second byte in a System Exclusive message):

```
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
DigiDesign          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
Cheetah             0x36
Waldorf             0x3E
Kawai               0x40
Roland              0x41
Korg                0x42
Yamaha              0x43
Casio               0x44
Akai                0x45
```

You'll note that we use only one byte for the Manufacturer ID. And since a midi data byte can't be greater than 0x7F, that means we have only 127 IDs to dole out to manufacturers. Well, there are more than 127 manufacturers of MIDI products.

To accomodate a greater range of manufacturer IDs, the MMA decided to reserve a manufacturer ID of 0 for a special purpose. When you see a manufacturer ID of 0, then there will be two more data bytes after this. These two data bytes combine to make the real manufacturer ID. So, some manufacturers have IDs that are 3 bytes, where the first byte is always 0. Using this "trick", the range of unique manufacturer IDs is extended to accomodate over 16,000 MIDI manufacturers.

For example, Microsoft's manufacturer ID consists of the 3 bytes 0x00 0x00 0x41. Note that the first byte is 0 to indicate that the real ID is 0x0041, but is still different than Roland's ID which is only the single byte of 0x41.

A manufacturer must get a registered ID from the MMA 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.

Universal SysEx messages are SysEx messages that are not for any one particular manufacturer, but rather, meant to be utilized by all manufacturer's products. 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) were chosen as the most suitable vehicle to transmit the data over MIDI. But, it was decided not to use a particular manufacturer's ID for these SysEx messages. So, a universal ID was created. There are actually 2 IDs dedicated to Universal SysEx messages. 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). Here are those ID numbers:

```
RealTime ID         0x7F
Non-RealTime ID  0x7E
```

No particular manufacturer is ever assigned an ID consisting of the single byte 0x7F or 0x7E. These are reserved for Universal SysEx messages adopted by the MMA.

A general template for these two IDs was defined. After the ID byte is a *SysEx Channel* 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. There are several Sub IDs defined for particular messages. There is a Sub ID for a Universal SysEx message to set a device's master volume. (This is different than Volume controller which sets the volume for only one particular MIDI channel). There is a Sub ID for a Universal SysEx message to set a device's Pitch Wheel bend range. There are a couple of Sub IDs for some Universal SysEx messages to implement a waveform (sample) dump over MIDI. Etc.

The next sections of this online book document the defined Universal SysEx messages.

GM System Enable/Disable

This Universal SysEx message enables or disables the General MIDI mode of a sound module. 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 with a GM mode/patch set. 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
```

It is best to respond as quickly as possible to this message, and to be ready to accept incoming note (and other) messages soon after, as this message may be included at the start of a General MIDI file to ensure playback by a GM module. Most modules are fully setup in GM mode by 100 milliseconds after receiving the GM System Enable message.

When GM Mode is first enabled, a device should assume that the "Grand Piano" patch (ie, the first GM patch) is the currently selected patch upon all 16 MIDI channels. The device should also internally reset all controllers and assume the power-up state described in the General MIDI Specification.

While GM mode is enabled, a device should also ignore Bank Select messages (since GM does not have more than one bank of patches). Only when the GM Disable message is received (with Sub-ID2 = 0 to disable GM mode) will a device then respond to Bank Select messages (and knock itself out of GM mode).

Master Volume

This Universal SysEx message 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
```

Identity Request

Sometimes, a device may wish to know what other devices are connected to it. For example, a Patch Editor software running on a computer may wish to know what devices are connected to the computer's MIDI port, so that the software can configure itself to accept dumps from those devices.

The **Identity Request** Universal Sysex message can be sent by the Patch Editor software. When this message is received by some device connected to the computer, that device will respond by sending an **Identity Reply** Universal Sysex message back to the computer. The Patch Editor can then examine the information in the **Identity Reply** message to determine what make and model device is connected to the computer. Each device that understands the Identity Request will reply with its own Identity Reply message.

Here is the Identity Request message:

```
0xF0   SysEx
0x7E   Non-Realtime
0x7F   The SysEx channel. Could be from 0x00 to 0x7F.
       Here we set it to "disregard channel".
0x06   Sub-ID -- General Information
0x01   Sub-ID2 -- Identity Request
0xF7   End of SysEx
```

Here is the Identity Reply message:

```
0xF0   SysEx
0x7E   Non-Realtime
0x7F   The SysEx channel. Could be from 0x00 to 0x7F.
       Here we set it to "disregard channel".
0x06   Sub-ID -- General Information
0x02   Sub-ID2 -- Identity Reply
0xID   Manufacturer's ID
0xf1   The f1 and f2 bytes make up the family code. Each
0xf2   manufacturer assigns different family codes to his products.
0xp1   The p1 and p2 bytes make up the model number. Each
0xp2   manufacturer assigns different model numbers to his products.
0xv1   The v1, v2, v3 and v4 bytes make up the version number.
0xv2
0xv3
0xv4
```

0xF7   End of SysEx

0xF7   End of SysEx

MIDI Sample Dump Standard (SDS) is covered in the document MIDI Sample Dump Standard.

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). 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 slave device might receive (from a master device) 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 the master, and eventually a MIDI Stop to halt playback. See Syncing Sequence Playback.

Tick

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 Tick message is sent at regular intervals of one message every 10 milliseconds.

Status

0xF9

Data

None

Errata

While a master device's "clock" is playing back, it will send a continuous stream of MIDI Tick events at a rate of one per every 10 milliseconds.

A slave device might receive (from a master device) 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 Ticks in order to keep the playback in sync with the master, and eventually 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 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.

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.

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.
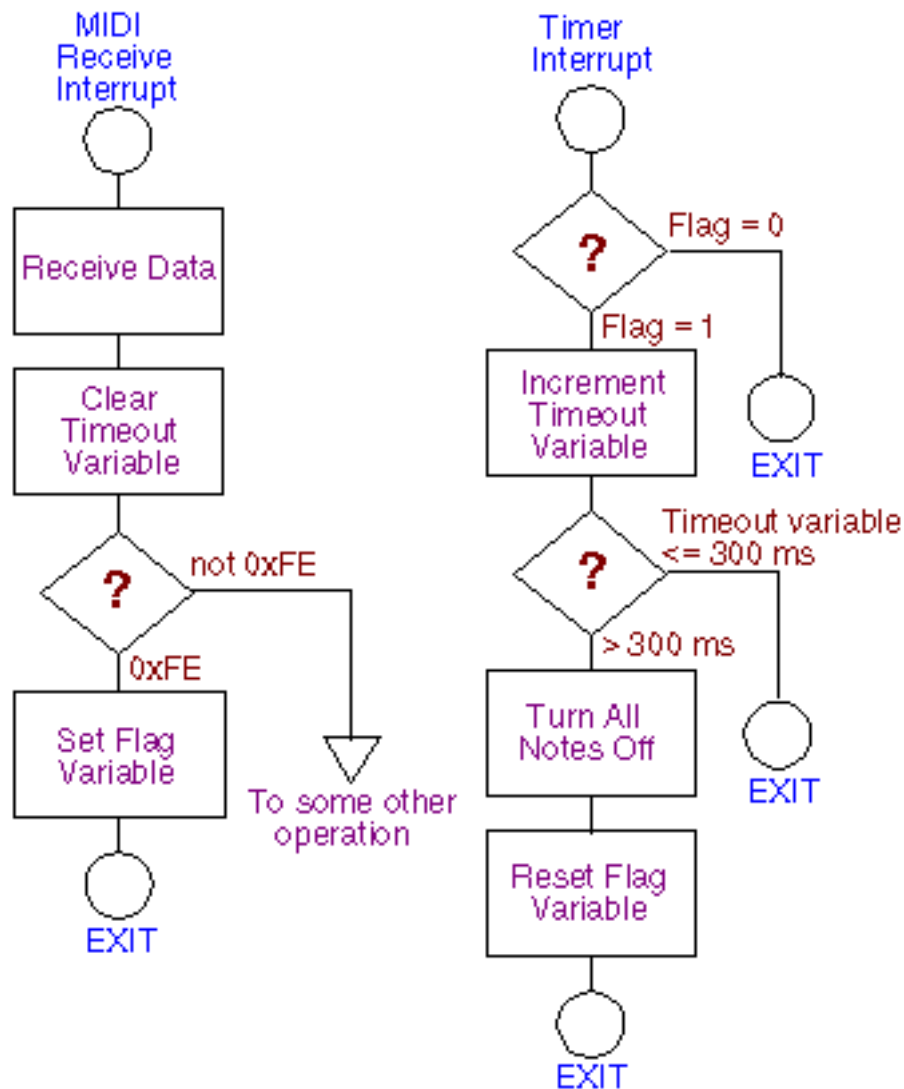
The logic for active sense detection

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.

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 <u>at any time</u>, 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 <u>Running Status</u>, <u>Ignoring MIDI Messages</u>, and <u>Syncing Sequence Playback</u>.

Running Status

The MIDI spec allows for a MIDI message to be sent without its Status byte (ie, just its data bytes are sent) <u>as long as the previous, transmitted message had the same Status</u>. 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).

Alternately, if a sequencer wishes to control playback independent of tempo, it can use Tick messages. These are sent at a rate of 1 message every 10 milliseconds. Of course, it is then up to the slave device to maintain and update its clock based upon these messages. The slave will be doing its own counting off of how many milliseconds are supposed to be in each "beat" at the current tempo.

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" <u>immediately</u> 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 <u>instead</u> 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, or 0xFD). In other words, a device is expected to be able to deal with <u>all</u> 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 <u>not</u> 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.

Definitions

sound module

This is an electronic musical instrument that can make musical sounds. It typically has numerous "patches" (sometimes called "programs" or "instruments") of sounds, often including acoustic instruments like Piano, Guitar, Woodwind and Brass instruments, Strings, etc, as well as perhaps an assortment of synthesizer sounds.

But, although a sound module can make sounds, it has no facility for a musician to directly control it. It has no built-in controller (ie, a keyboard, or a ribbon controller, or guitar fretboard, or any other device that the musician can use to trigger the sounds). A sound module has only its built-in sounds, and is therefore designed to work in conjunction with some other device that controls it.

There is no standardized terminology for describing such a musical instrument. Some people instead call it a **rackmount synth (or sampler)** because of the fact that many of these instruments are designed to fit inside a standard rack. Others call it an **external synth (or sampler)**. I prefer the term "sound module", and use that term throughout my web site.

Nowadays, most sound modules are multi-timbral and offer at least the General MIDI Patch set.

Most computer sound cards nowadays have a built-in sound module to "play" MIDI messages. These built-in sound modules may employ older analog circuits such as FM synthesis, or more modern digital waveform based circuits such as wavetable synths, or even more modern hybrids such as physical modeling synths.

A **drum box** is a special purpose sound module that has only drum sounds in it. But, it may also have pads or buttons to trigger those sounds, in which case it can be considered to have a very limited, built-in controller.

MIDI controller

This is an electronic musical instrument that can be directly played by a musician. It therefore has a keyboard, or ribbon controller, or guitar fretboard, or a mouthpiece (ie, as in a breath controller), or drum pads, or some other device to allow the musician to play it. It is often used to control sound modules or other MIDI devices which need to be remotely controlled.

Many controllers also have a built-in sound module, and are therefore self-contained

musical instruments playable by a musician. But others have no built-in sound module (although they will typically have buttons to select patches upon a remote sound module), and therefore need to be used with other sound modules.

MIDI mostly came about because keyboard players were buying synthesizers in increasing numbers from many different manufacturers, and they wanted to "sync" those various keyboards together, plus "play" them under "computer control" (since computer use was also increasing dramatically during that time). Companies were making analog sequencers, but each company's sequencer worked only with its own products. (See the article The historical significance of computers/MIDI for more explanation about what musicians wanted to do, and what problems they were facing back then). For example, Sequential Circuits made an analog sequencer that attached to a proprietary jack on the Prophet 5, and could be used to program the playback of a short phrase, but it wouldn't work on a Roland D-10 or Oberheim OBX-8 synth. And during the early 1980's, a lot of "pop dance music" was using sequenced phrases played on synths.

Two of the leading musical manufacturers back then were a California company called Sequential Circuits (who were making the best-selling synthesizer at the time -- early 1980's -- called the Prophet 5) and Roland, a japanese company who had been making popular musical keyboards for awhile, and were at the forefront of making musical interfaces for computers.

Indeed, Roland was one of the first companies to start making musical products that attached to computers. At the time, IBM had just released its first personal computer, the IBM PC, which gave a "serious endorsement" to personal computers. Also, Commodore had released the Commodore 64, one of the first really affordable "home computers", and sales were really starting to take off with personal computers. The Commodore 64 had a built-in analog synthesizer chip made by a musical company called Ensoniq. It wasn't very fancy, but nonetheless, musicians started playing around with it, and quickly began to discover that the programmability of digital computers, combined with a musical instrument, offered them a lot of potential solutions to those problems they were having back then.

Roland saw the potential musical use that computers offered. So Roland began work on a musical interface for the IBM PC. (Unlike the C-64, the IBM PC had no "fancy" built-in sound chip, so there was a void to be filled with a third party product). Roland envisioned a "digital sequencer" to replace the analog sequencers of the time, and it would be built around an IBM PC, which offered a lot of programmability/versatility using tools made by many other companies. Of course, since the PC had no built-in sound chip, and also, Roland wanted this digital sequencer to be able to work with Roland's entire line of new keyboards, Roland decided to build a hardware "musical interface" for the PC. Roland chose to make this an ISA card that plugged into one of the slots of an AT style IBM PC computer, to which an external box was attached that contained further circuitry. It had a lot of intelligent electronics in it, which primarily were designed to add features that turned a computer into a versatile sequencer. For example, it had a built-in metronome and also a tape sync jack so that the computer playback could be synced to a magnetic analog tape recorder such as a reel-to-reel or cassette recorder. It had built-in hardware timers to control the playback, and some filtering options of the digital data flowing between the computer and an external musical instrument. In designing this proprietary PC musical interface, Roland came up with a simple hardware circuit, and a fancy new, "digital language" that they planned to use with all of their upcoming musical keyboards. This digital language would allow the computer and musical instruments to transfer "control data" between them. This new interface would be known as a "Musical Instrument Digital Interface" or MIDI. (Roland loves anacronyms). And the PC card itself would become the MPU-401 ("Musical Processing Unit, model 401" -- gotta love those anacronyms), the first MIDI interface for a computer.

(To this day, the MIDI interfaces built into some modern sound cards still offer hardware compatibility to the original MPU-401 since it became such a widely used MIDI interface for a computer).

Roland and Sequential Circuit representatives used to see each other at NAMM (a business trade show for the music industry), and were talking about how customers were wishing that they had some sequencer that worked with the keyboards from all manufacturers. And Roland said, "You know, we're working on this new peripheral for the IBM PC to turn it into a musical sequencer, using a hardware/software protocol in all of our upcoming keyboards. Would you be interested in supporting this? We could both benefit from it since you make such a popular synth, and we'll be making a musical sequencer for these increasingly popular personal computers". The SQ reps said "Sure. We've got some ideas of our own that we're adding to upcoming synths. Maybe we can incorporate these into one standard between us". So Roland sent some design specs to the SQ guys, who made some suggested changes and additions. They both decided to go ahead and adopt this "standard". Then they thought, "Why not see if we can get some of the other leading musical manufacturers to adopt it, as long as it's not really a proprietary standard anymore?". So, they contacted other popular musical manufacturers such as Yamaha and Oberheim, and got them onboard too. MIDI was perhaps the first true effort at joint development among a large number of musical manufacturers.

The first keyboard on the market with a MIDI interface was the Prophet 600 by Sequential Circuits in 1983. Of course, the Roland MPU-401 appeared shortly thereafter with some PC software from Roland that turned the PC into a musical sequencer. Roland made some MPU-401 programming information available to other parties, and soon, there were other PC programs that supported the MPU-401 interface. (Indeed, those other programs proved to be more popular than Roland's initial MESA software. One of those products was called "Cakewalk", made by a small upstart known as "12 Tone Systems", who later changed their name to "Cakewalk" since that product became synonymous with the company). New Roland keyboards also sported a MIDI interface. Yamaha released the DX-7 later that same year, their first keyboard with a MIDI interface. The DX-7 proved to be a huge hit with musicians. They loved the sound of its new, "FM synthesis", bought it in droves, and started to fool around with this new "MIDI interface" thing. Apple computer made a MIDI interface available for its Macintosh computer, and started promoting the computer in the music market during the mid-to-late 1980's since that was a market completely ignored by their biggest competitor IBM, and one that offered them many potential sales, being that musicians loved the flexibility and ease of use of the new, computer-based "digital sequencers". MIDI took off then. By 1985, virtually every new musical keyboard on the market had a MIDI interface.

Roland and SQ thought "Well, we don't want to be responsible for disseminating new additions to MIDI to every music/computer company that supports MIDI. We have too much other work to do. We don't have the time to prepare and mail documents to everyone else". So, all of the musical manufacturers supporting MIDI agreed to start a new organization called the MIDI Manufacturer's Association (ie, MMA). It would be the duty of this new organization to produce/disseminate the paper documents for the MIDI standard, and be the clearing house for new additions/changes to the specification. Members of the MMA would pay some dues to fund the cost of operating this new organization. (Dues were $50 a year, as I recall). And thus, the MMA was born.

It was through the MMA that new additions to the MIDI specification were channeled. For example, Opcode offered the MIDI File Format specification to the MMA, and thus, everyone started creating MIDI sequencer software that could read/write each others' data files. (Prior to that, each sequencer program had its own proprietary file format, and couldn't read each others' data files). Digidesign offered MIDI Time Code (MTC), and thus a standard was created for syncing the playback of various sequencers. (Prior to that, most sequencers also had proprietary sync protocols, for example, a whole mess of incompatible FSK sync interfaces).

In recent years, the MMA hasn't been doing much of anything except for selling photocopies of specifications that were written years ago, and some people have almost forgotten that it even exists. Consequently, there have been very few changes/additions to MIDI for about the past 5 to 10 years. In fact, there have even been some proprietary things creeping in, such as proprietary redefinitions of MIDI status bytes used by multiple bus parallel and serial computer MIDI interfaces. Things often work in cycles though, so as musicians and manufacturers persue proprietary, solo endeavors and ultimately discover the perils of not persuing standards, then things may swing back the other way, and we may eventually see some problems with MIDI solved in a universal way (such as the fact that a single MIDI bus is limited to 16 channels -- too few to accomodate the newer "multi-timbral" synths that weren't around when MIDI was originally conceived, and also the slow-by-today's-standards 32K serial baud rate of MIDI).

Preface

After the MIDI 1.0 standard was finalized in the early 1980's, numerous musical instruments with MIDI jacks appeared upon the market. Musicians started to attach these instruments via their MIDI ports, and quickly discovered that the MIDI 1.0 specification had overlooked some important concerns.

One typical scenario may have been as follows:

A musician attaches his Roland D-10 to his Yamaha DX-7, because he prefers the front panel of the D-10, but prefers the sound of the DX-7, and he wants to use the D-10 to "play" the DX-7. He selects the patch labeled "Piano" on the D-10, and he plays the D-10 keyboard, and on the DX-7 he hears... a trumpet? How did this happen? Well, it happened because MIDI sends a program change message that contains only a patch number -- not the actual name of the patch. So if patch #1 on the DX-7 is a trumpet sound, then that's what he gets on the DX-7, despite the fact that selecting patch #1 on the D-10 yields a piano sound on the D-10. The MIDI 1.0 specification did not require that particular sounds be assigned to particular patch numbers, so every manufacturer used his own discretion as to "patch mapping".

But the real problem was with MIDI files that the musician made. MIDI files contain only MIDI messages. So, any program change event in a MIDI file refers only to a patch number as well -- not the actual patch name. So, this musician creates a MIDI file using his D-10. He has a piano track, so he puts a program change event at the track's beginning to select patch #1, which happens to be a Piano sound on his D-10. He takes that MIDI file to a friend's house. The friend has a DX-7. They play the MIDI file on that DX-7, and suddenly, the piano part is playing with a trumpet sound. Well, that's because patch #1 on the DX-7 is not a piano -- it's a trumpet sound. To "fix" the MIDI file, now the musician with the DX-7 has to edit the MIDI tracks and change every MIDI Program Change event so that it refers to the correct patch number on his DX-7. This deviance among MIDI sound modules made it very difficult for musicians to create MIDI arrangements that played properly upon various MIDI sound modules.

There were also some other deviances among early MIDI modules that made it more difficult to use them together via MIDI. To address these concerns, Roland proposed an addendum to the MIDI 1.0 specification in the late 1980's. This new addendum was called "General MIDI" (GM). It added some new requirements to the base MIDI 1.0 specification (but does not supplant any parts of the 1.0 specification -- the 1.0 specification is still the base level to which all MIDI devices should adhere). GM has now been adopted as part of the MIDI 2.0 specification.

General MIDI Patches

So to make MIDI Program Change messages of more practical use, Roland found it necessary to adopt a standard "patch bank". In other words, what was needed was to assign specific instrument sounds to specific patch numbers. For example, it was decided that patch number 1 upon all sound modules should be the sound of an *Acoustic Grand Piano*. In this way, no matter what MIDI sound

module you use, when you select patch number 1, you always hear some sort of Acoustic Grand Piano sound. A standard was set for 128 patches which must appear in a specific order, and this standard is called *General MIDI* (GM). For example, patch number 25 upon a GM module must be a *Nylon String Guitar*. The chart, GM Patches, shows you the names of all GM Patches, and their respective *Program Change* numbers.

Nowadays, most modules (including the built-in sound modules of computer sound cards) ship with a GM bank (of 128 patches) so that it is easy to play MIDI files upon any MIDI module, without needing to edit all of the Program Change events in the file.

General MIDI Multi-Timbral requirement

Another burgeoning technology in the late 1980's was the multi-timbral module. Typically, there were deviances in the way that various manufacturers implemented this, since the 1.0 specification did not specifically address such devices. For example, some early multi-timbral modules supported only a limited set of the 16 MIDI channels simultaneously, so if you had a MIDI file with tracks upon unsupported MIDI channels, you wouldn't hear those tracks play back. You may not have even realized that those parts weren't being played.

So, one requirement of a GM-compliant module is that it must be fully multi-timbral, meaning that it can play MIDI messages upon all 16 channels simultaneously, with a different GM Patch sounding for each channel.

General MIDI Note Number assignments

There were also deviances in regards to Note Number mapping. For example, some manufacturers mapped middle 'C' to MIDI Note Number 60. Others mapped it to Note Numbers 72 or 48. Some modules even had middle C mapped to various places in different patches, depending upon the instrument. For example, a bass guitar patch may have middle C mapped to the highest C on the keyboard (since the most useful range on a bass guitar is below middle C). A flute patch may have middle C mapped to the lowest C on the keyboard.

The result was that, it became confusing to keep track of which key (ie, MIDI Note Number) played middle C for each patch. Also, when a MIDI track was played back upon certain modules, the part may play back an octave too high or low.

It therefore was decided that all patches must sound an A440 pitch when receiving a MIDI note number of 69. (ie, Note Number 69 plays the A above middle C, and therefore Note Number 60 is middle C).

There were deviances in regards to "drum machines" as well. Most drum machines (and drum units built into multi-timbral modules) play a different drum sound for each MIDI Note Number. But the 1.0 specification never spelled out which drum sounds were assigned to which MIDI note numbers. So, whereas note number 60 may play a snare upon one drum unit, upon another drum unit, it may play a crash cymbal. Again, this caused trouble with MIDI files, since sometimes a drum part would play back with the wrong drum sounds.

To address this discrepancy, the GM addendum contains a "drum map". This assigns about 48 common drum sounds to 48 specific MIDI Note Numbers. The assignments of drum sounds to MIDI notes is shown in the chart, GM Drum Sounds. Also, it was decided that a GM drum unit should default to using MIDI channel 10 to receive MIDI messages. Therefore, a composer of a GM MIDI file can safely assume that his drum part will play correctly if he uses the GM Drum note assignments and records the drum part upon MIDI channel 10.

General MIDI polyphony

Polyphony is how many notes a module can sound simultaneously. For example, perhaps a module can sound 32 notes simultaneously. Early MIDI modules typically had very limited polyphony. For example, the Prophet 5 could sound only 5 notes simultaneously.

This discrepancy in polyphony among MIDI modules made it difficult for arrangers to create MIDI files that played properly upon various modules. For example, if the arranger created too "busy" an arrangement, it could exceed the polyphony of a particular module, and therefore some of the notes may not be heard.

To address this discrepancy, the GM addendum stipulated that a GM module should be capable of sounding at least 24 notes simultaneously. (Ie, It must have 24 note polyphony). It could exceed 24 note polyphony, but it had to have at least that level of polyphony. In this way, if an arranger ensured that he never had more than 24 notes sounding simultaneously in his MIDI file, all notes of his arrangement would be heard upon any GM module.

Other General MIDI requirements

Finally, the GM addendum attempted to address some other discrepancies by spelling out a few more requirements.

A GM module should respond to velocity (ie, for note messages). This typically controls the VCA level (ie, volume) of each note, but the GM addendum unfortunately did not set a specific function for velocity. Some modules may allow velocity to affect other parameters on some patches.

The pitch wheel bend range should default to +/- 2 semitones. This allows an arranger to use pitch bend messages in his arrangement without worrying whether a bend that is supposed to be up 2 whole steps will instead jump up 2 octaves upon a certain sound module.

The module also should respond to Channel Pressure (often used to control VCA level or VCO level for vibrato depth). Again, the GM addendum unfortunately did not set a specific function for channel pressure, although typically it defaults to controlling the volume of a note while it is being held.

Finally, a GM module should also respond to the following MIDI controller messages: Modulation (1) (usually hard-wired to control LFO amount, ie, vibrato), Channel Volume (7), Pan (10), Expression (11), Sustain (64), Reset All Controllers (121), and All Notes Off (123). Additionally, the module should respond to these Registered Parameter Numbers: Pitch Wheel Bend Range (0), Fine Tuning (1), and Coarse Tuning (2).

There were also some default settings that a GM module should apply upon power up. Channel Volume should default to 90, with all other controllers and effects off (including pitch wheel offset of 0). Initial tuning should be standard, A440 reference.

General MIDI messages

The GM addendum did specify a couple System Exclusive messages to alter settings that are common to all GM units, but which were not addressed by the 1.0 specification.

One such message is for Master Volume -- not just the volume of a patch upon any one MIDI channel, but the master volume of the module itself.

There is also a System Exclusive message that can be used to turn a module's General MIDI mode on or off. This is useful for modules that also offer more expansive, non-GM playback modes or extra, programmable banks of patches beyond the GM set, but need to allow the musician to switch to GM mode when desired.

Conclusion

GM Standard makes it easy for musicians to put *Program Change* messages in their MIDI (sequencer) song files, confident that those messages will select the correct instruments on all GM sound modules, and the song file would therefore play all of the correct instrumentation automatically. Furthermore, musicians need not worry about parts being played back in the wrong octave. Finally, musicians didn't have to worry that a snare drum part, for example, would be played back on a Cymbal. The GM Standard also spells out other minimum requirements that a GM module should meet, such as being able to respond to Pitch and Modulation Wheels, and also being able to

play 24 notes simultaneously (with dynamic voice allocation between the 16 Parts). All of these standards help to ensure that MIDI Files play back properly upon setups of various equipment.

The GM standard is actually not encompassed in the MIDI specification proper (ie, it's an addendum), and there's no reason why someone can't set up the Patches in his sound module to be entirely different sounds than the GM set. After all, most MIDI sound modules offer such programmability. But, most have a GM option so that musicians can easily play the many MIDI files that expect a GM module.

NOTE: The GM Standard doesn't dictate how a module produces sound. For example, one module could use cheap FM synthesis to simulate the Acoustic Grand Piano patch. Another module could use 24 digital audio waveforms of various notes on a piano, mapped out across the MIDI note range, to create that one Piano patch. Obviously, the 2 patches won't sound exactly alike, but at least they will both be piano patches on the 2 modules. So too, GM doesn't dictate VCA envelopes for the various patches, so for example, the Sax patch upon one module may have a longer release time than the same patch upon another module.

GM Patches

This chart shows the names of all 128 GM Instruments, and the MIDI Program Change numbers which select those Instruments.

The patches are arranged into 16 "families" of instruments, with each family containing 8 instruments. For example, there is a *Reed* family. Among the 8 instruments within the Reed family, you will find Saxophone, Oboe, and Clarinet.

Prog#	Instrument	Prog#	Instrument
PIANO		**CHROMATIC PERCUSSION**	
1	Acoustic Grand	**9**	Celesta
2	Bright Acoustic	**10**	Glockenspiel
3	Electric Grand	**11**	Music Box
4	Honky-Tonk	**12**	Vibraphone
5	Electric Piano 1	**13**	Marimba
6	Electric Piano 2	**14**	Xylophone
7	Harpsichord	**15**	Tubular Bells
8	Clavinet	**16**	Dulcimer
ORGAN		**GUITAR**	
17	Drawbar Organ	**25**	Nylon String Guitar
18	Percussive Organ	**26**	Steel String Guitar
19	Rock Organ	**27**	Electric Jazz Guitar
20	Church Organ	**28**	Electric Clean Guitar

21	Reed Organ	29	Electric Muted Guitar
22	Accoridan	30	Overdriven Guitar
23	Harmonica	31	Distortion Guitar
24	Tango Accordian	32	Guitar Harmonics

BASS		**SOLO STRINGS**	
33	Acoustic Bass	41	Violin
34	Electric Bass(finger)	42	Viola
35	Electric Bass(pick)	43	Cello
36	Fretless Bass	44	Contrabass
37	Slap Bass 1	45	Tremolo Strings
38	Slap Bass 2	46	Pizzicato Strings
39	Synth Bass 1	47	Orchestral Strings
40	Synth Bass 2	48	Timpani

ENSEMBLE		**BRASS**	
49	String Ensemble 1	57	Trumpet
50	String Ensemble 2	58	Trombone
51	SynthStrings 1	59	Tuba
52	SynthStrings 2	60	Muted Trumpet
53	Choir Aahs	61	French Horn
54	Voice Oohs	62	Brass Section
55	Synth Voice	63	SynthBrass 1
56	Orchestra Hit	64	SynthBrass 2

REED		**PIPE**	
65	Soprano Sax	73	Piccolo
66	Alto Sax	74	Flute
67	Tenor Sax	75	Recorder
68	Baritone Sax	76	Pan Flute
69	Oboe	77	Blown Bottle
70	English Horn	78	Skakuhachi
71	Bassoon	79	Whistle
72	Clarinet	80	Ocarina

SYNTH LEAD		**SYNTH PAD**	
81	Lead 1 (square)	89	Pad 1 (new age)
82	Lead 2 (sawtooth)	90	Pad 2 (warm)
83	Lead 3 (calliope)	91	Pad 3 (polysynth)
84	Lead 4 (chiff)	92	Pad 4 (choir)
85	Lead 5 (charang)	93	Pad 5 (bowed)
86	Lead 6 (voice)	94	Pad 6 (metallic)
87	Lead 7 (fifths)	95	Pad 7 (halo)
88	Lead 8 (bass+lead)	96	Pad 8 (sweep)

SYNTH EFFECTS		**ETHNIC**	

97	FX 1 (rain)	**105**	Sitar
98	FX 2 (soundtrack)	**106**	Banjo
99	FX 3 (crystal)	**107**	Shamisen
100	FX 4 (atmosphere)	**108**	Koto
101	FX 5 (brightness)	**109**	Kalimba
102	FX 6 (goblins)	**110**	Bagpipe
103	FX 7 (echoes)	**111**	Fiddle
104	FX 8 (sci-fi)	**112**	Shanai

	PERCUSSIVE		SOUND EFFECTS
113	Tinkle Bell	**121**	Guitar Fret Noise
114	Agogo	**122**	Breath Noise
115	Steel Drums	**123**	Seashore
116	Woodblock	**124**	Bird Tweet
117	Taiko Drum	**125**	Telephone Ring
118	Melodic Tom	**126**	Helicopter
119	Synth Drum	**127**	Applause
120	Reverse Cymbal	**128**	Gunshot

Prog# refers to the MIDI Program Change number that causes this *Patch* to be selected. These decimal numbers are what the user normally sees on his module's display (or in a sequencer's "Event List"), but note that MIDI modules count the first Patch as 0, not 1. So, the value that is sent in the Program Change message would actually be one less. For example, the Patch number for Reverse Cymbal is actually sent as 119 rather than 120. But, when entering that Patch number using sequencer software or your module's control panel, the software or module understands that humans normally start counting from 1, and so would expect that you'd count the Reverse Cymbal as Patch 120. Therefore, the software or module automatically does this subtraction when it generates the MIDI Program Change message.

So, sending a MIDI Program Change with a value of 120 (ie, actually 119) to a Part causes the Reverse Cymbal Patch to be selected for playing that Part's MIDI data.

GM Drum Sounds

This chart shows what drum sounds are assigned to each MIDI note for a GM module (ie, that has a drum part).

MIDI Note #	Drum Sound	MIDI Note #	Drum Sound
35	Acoustic Bass Drum	59	Ride Cymbal 2
36	Bass Drum 1	60	Hi Bongo
37	Side Stick	61	Low Bongo

38	Acoustic Snare	**62**	Mute Hi Conga
39	Hand Clap	**63**	Open Hi Conga
40	Electric Snare	**64**	Low Conga
41	Low Floor Tom	**65**	High Timbale
42	Closed Hi-Hat	**66**	Low Timbale
43	High Floor Tom	**67**	High Agogo
44	Pedal Hi-Hat	**68**	Low Agogo
45	Low Tom	**69**	Cabasa
46	Open Hi-Hat	**70**	Maracas
47	Low-Mid Tom	**71**	Short Whistle
48	Hi-Mid Tom	**72**	Long Whistle
49	Crash Cymbal 1	**73**	Short Guiro
50	High Tom	**74**	Long Guiro
51	Ride Cymbal 1	**75**	Claves
52	Chinese Cymbal	**76**	Hi Wood Block
53	Ride Bell	**77**	Low Wood Block
54	Tambourine	**78**	Mute Cuica
55	Splash Cymbal	**79**	Open Cuica
56	Cowbell	**80**	Mute Triangle
57	Crash Cymbal 2	**81**	Open Triangle
58	Vibraslap		

A note-on with note number 42 will trigger a Closed Hi-Hat. This should cut off any Open Hi-Hat or Pedal Hi-Hat sound that may be sustaining. So too, a Pedal Hi-Hat should cut off a sustaining Open Hi-Hat or Closed Hi-Hat. In other words, only one of these three drum sounds can be sounding at any given time.

Similiarly, a Short Whistle should cut off a Long Whistle. A Short Guiro should cut off a Long Guiro. An Mute Triangle should cut off an Open Triangle. A Mute Cuica should cut off an Open Cuica.

Normally, all the above drum sounds have a fixed duration. Regardless of the time between when a Note-On is received and when a matching Note-Off is received, the drum sound always plays for a given duration. For example, assume that a device has a "Crash Cymbal 1" sound that plays for 4 seconds. If a Note-On for note number 49 is received, that cymbal sound starts playing. If a Note-Off for note number 49 is received only 1 second later, that should not cut off the remaining 3 seconds of the sound. The exceptions may be Long Whistle and Long Guiro, which may use the duration between the Note-On and Note-off to determine how "long" the sound plays.

If a drum is still sounding when another one of its Note-Ons is received, typically, another voice "stacks" another instance of that sound playing.

Standard MIDI File (SMF) Format

The **Standard MIDI File** (SMF) is a file format used to store MIDI data (plus some other kinds of data typically needed by a sequencer).

This format stores the standard MIDI messages (ie, status bytes with appropriate data bytes) plus a time-stamp for each message (ie, a series of bytes that represent how many clock pulses to wait before "playing" the event). The format also allows saving information about tempo, time and key signatures, the names of tracks and patterns, and other information typically needed by a sequencer. One SMF can store information for numerous patterns and tracks so that any sequencer can preserve these structures when loading the file.

NOTE: A **track** usually is analogous to one musical part, such as a Trumpet part. A **pattern** would be analogous to all of the musical parts (ie, Trumpet, Drums, Piano, etc) for one song.

The format was designed to be generic so that the most important data can be read by all sequencers. Think of a MIDI file as a musical version of an ASCII text file (except that the MIDI file contains binary data), and the various sequencer programs as text editors all capable of reading that file. But, unlike ASCII, MIDI file format saves data in **chunks** (ie, groups of bytes preceded by an ID and size) which can be parsed, loaded, skipped, etc. Therefore, SMF format is flexible enough for a particular sequencer to store its own proprietary, "extra" data in such a way that another sequencer won't be confused when loading the file and can safely ignore this extra stuff that it doesn't need. For example, maybe a sequencer wants to save a "flag byte" that indicates whether the user has turned on an audible metronome click. The sequencer can save this flag byte in such a way that another sequencer can skip this byte without having to understand what that byte is for. In the future, the SMF format can also be extended to include new "official" chunks that all sequencer programs may elect to load and use. This can be done without making old data files obsolete, nor making old sequencers no longer able to load the new files. So, the format is designed to be extensible in a backwardly compatible way.

Of course, SMF files may be used by other MIDI software than just sequencers. Since SMF files can store any and all types of MIDI

messages, including System Exclusive messages, they may be used to store/load data by all kinds of MIDI software, such as a Patch Editor that wants to save some System Exclusive messages it received from a MIDI module. (The "timestamp" for each message may be irrelevant to such a Patch Editor. But it's easily ignored for programs that don't really need it).

In conclusion, any software that saves or loads MIDI data should use SMF format for its data files.

Many devices use playback of digital audio waveforms as their audio source. It was desirable to implement a sub-protocol within MIDI in which devices could exchange this digital audio waveform data. In other words, a protocol was needed that allowed devices to exchange waveform data over MIDI cables within the parameters of MIDI. The only way to do this was with System Exclusive messages, and so several specific SysEx messages were defined in order to implement *Sample Dump Standard* (SDS). Many samplers support this protocol.

The device that sends the waveform data is the transmitter, and the device that receives it is the receiver.

A waveform exchange (ie, *dump*) can be done with or without handshaking. In the non-handshaking version, the transmitter's MIDI OUT is connected to the receiver's MIDI IN, and only the transmitter sends MIDI messages to the receiver. In the handshaking version, the transmitter's MIDI OUT is connected to the receiver's MIDI IN and the receiver's MIDI OUT is connected to the transmitter's MIDI IN. The transmitter sends a portion of the waveform data, after which it expects some sort of acknowledgement from the receiver that the portion has been received successfully or otherwise, and then the transmitter sends the next portion. All of this is accomplished with the devices passing defined SysEx messages between themselves.

The SysEx messages are the DUMP REQUEST, ACK, NAK, WAIT, CANCEL, Dump Header, and Data Packet messages. The first 5 (capitalized) are generated by the receiver. The last 2 are generated by the transmitter.

The dump procedure works as follows. The transmitter sends a Dump Header to indicate a dump start to the receiver. (This could have happened as a result of the receiver requesting the transmitter to start a dump via the DUMP REQUEST). The transmitter then waits for upto 2 seconds for a response from the receiver. This gives the receiver a chance to decide if it wants to and can accept the waveform. If no response is received, then the transmitter assumes a non-handshaking action, and proceeds to send out the first Data Packet. If an expected response (ie, handshake) is received instead, then the transmitter bases its next action upon the receiver's response. If the response is an ACK, the transmitter proceeds to send out the first Data Packet. If the response is a NAK, the transmitter sends the Dump Header again. If the response is a CANCEL, the transmitter aborts the dump. If the response is a WAIT, the transmitter pauses indefinitely until it subsequently receives one of the preceding responses. After the first data packet is sent, the transmitter waits for upto 20 milliseconds for a response from the receiver. This gives the receiver time to perform certain error-checking on the packet's contents. If no response is received, then the transmitter assumes a non-handshaking action, and proceeds to send out the next Data Packet. If an expected response is received instead, then the transmitter bases its next action upon the receiver's response. If the response is an ACK, the transmitter proceeds to send out the next Data Packet. If the response is a NAK, the transmitter resends that same (ie, first) Data Packet again. If the response is a CANCEL, the transmitter aborts the dump. If the response is a WAIT, the transmitter pauses indefinitely until it subsequently receives one of the preceding responses. Eventually, the transmitter sends out as many Data Packets as are needed to pass all of the waveform data to the receiver, repeating this handshake procedure after each packet (or assuming a non-handshaking action after each packet). After that happens, the dump is done.

Here are the messages (with all bytes in hex). In each message, the byte notated as **cc** represents the SysEx channel that the message is being sent upon. There are 128 possible SysEx channels that a device can be set to (ie, 0 to 127).This allows various devices to be set to different SysEx channels along the daisy-chain, and have the dump occur between 2 particular devices with matching SysEx channels.

DUMP REQUEST

F0 7E **cc** 03 **sl sh** F7

If a receiving device wishes to initiate the dump (ie, tell some other device to send some waveform data), then the receiver sends the DUMP REQUEST. The **sl sh** is the 14-bit number (ie 0 to 16,384) of the waveform which the receiver is requesting from the transmitter. Most samplers number their internal waveforms from 0 to how ever many waveforms there are. Note that the 14-bit sample number is transmitted as 2 bytes where the first byte (sl) contains bits 0 to 6 (with high bit clear), and the second byte (sh) contains bits 7 to 13, right-justified (with high bit clear). When the transmitter gets this request, if such a sample number is available, the transmitter will kick off the dump with a Dump Header. Otherwise, the transmitter will cancel the dump. Typically, the receiver will wait for the Dump Header for a few seconds, and if not received, will abort the operation.

Dump Header

F0 7E **cc** 01 **sl sh** *ee* **pl pm ph** *gl gm gh* **hl hm hh** *il im ih* **jj** F7

The transmitter sends this to the receiver to provide information about the waveform data that is about to be sent (in Data Packet messages). The **sl sh** is the waveform's 14-bit number (ie 0 to 16,384). See Dump Request.

ee is the number of significant bits of the waveform. For example, a 16-bit resolution waveform would have a 16 here.

pl pm ph is the sample period in nanoseconds (ie, 1,000,000,000/sample rate in Hertz). For example, a waveform sampled at 41667 Hertz will have a period of 23,999 nanoseconds. This value is transmitted as 3 bytes where pl is bits 0 to 6, pm is bits 7 to 13 **right-justified**, and ph is bits 14 to 20 **right-justified** (ie, for a total of 20 bits of resolution) with the high bit of all 3 bytes clear. So, our 23,999 (0x5DBF) becomes the 3 bytes 3F 3B 01.

gl gm gh is the waveform length **in words**. (What this implies is that if you have 8-bit or less resolution, the waveform length will be half the number of sample points that you intend to dump. You always end up having to send an even number of points).

hl hm hh is the word offset (from 0, ie, the very first sample point in the waveform) where the *sustain loop* starts. **il im ih** is the word offset where the *sustain loop* ends (ie, where the playback loops back to the *sustain loop* start). **jj** is the looptype where 00 means "forward only" (most

common) and 01 means "backward/forward", and 7F means "no loop point" (ie, the waveform is played through once only without looping). Note that older MIDI samplers didn't support the 7F value for looptype. For these older samplers, usually, if you set both th start and end loop points to the same value as the waveform length, a sampler will consider this to be a non-looped waveform. So to be safe, when you want to indicate that a waveform is not to be looped, you should set looptype to 7F, and set the start and end loop positions to the same value as the waveform's length.

Data Packet

F0 7E **cc** 02 **kk** *[120 bytes here]* **ll** F7

The data packet is what is used to transfer the actual waveform data. It transfers 120 bytes of waveform data at a time. So, the total size of a packet is 127 bytes.

kk is the packet number from 0 to 127. The first packet that is sent is number 0. The second packet is number 1. After packet number 127, the count rolls over to 0 again (ie, packet 128 becomes 0 again). This number is used by the receiver to ensure that it hasn't missed any packets. The packet number is also used to distinguish new packets from resent packets. After all, if packet number 3 follows packet number 1, then either packet number 2 has been missed by the receiver, or the transmitter sent packets out of order. For example, assume that a device has gotten packet 1, found an error in it, and sends a NAK to the transmitter. But, the transmitter has already assumed non-handshaking and started sending packet 2. The receiver would note that the next arriving packet is number 2. Then, the transmitter finally sees the receiver's late NAK to packet number 1, and resends that packet. The receiver can then note that it has received packet 1 out of order.

The 120 bytes of waveform data follow. The transmitter has to pack up each sample point of its waveform data. With a 16-bit waveform, the transmitter must break up each 16-bit word into 3 bytes for transmission where the first contains bits 15 to 9, the second contains bits 8 to 2, and the third contains bits 1 and 0. In other words, unlike with the waveform length of the DUMP HEADER, the DATA PACKET's bytes are **left-justified**. The first data byte contains the highest 7 bits (which are placed in bit positions 0 to 6, since you'll remember that all transmitted data bytes must have bit 7 clear). The second data byte contains the next highest 7 bits. And the last data byte contains the remaining, lowest bits, which for a 16-bit point means the last two bits. For example, the 16-bit sample word 0xF0F0 would be 0x78 0x3C 0x00. Because each 16-bit word must be broken up into 3 bytes, and because there must be only 120 bytes in a packet, that means that a packet can contain 40, 16-bit sample points. In fact, waveforms with resolutions of 15 to 21 bits pack up likewise. Waveforms with resolutions of 8 to 14 bits pack each sample point into 2 bytes (for 60 points per packet). Waveforms with resolutions of 22 to 28 bits pack each sample point into 4 bytes (for 30 points per packet). Sample points are represented by 0 being full negative value. So, in a 16-bit waveform, 0x0000 is full negative value and 0xFFFF is full positive value (ie, signed shorts aren't used, unlike in the WAVE file format, so you have to subtract a 16-bit point by 0x8000 after unpacking the 3 bytes into an unsigned short, if you want to adjust to a signed short).

Here's a C example of how to unpack 3 bytes of a DATA PACKET into a 16-bit point (ie, # of significant bits = 16). It is passed a pointer to the first of those 3 bytes, and returns a signed 16-bit

point.

```
short unpack3(unsigned char * ptr)
{
    unsigned short num;

    /* Unpack 3 bytes into an unsigned short */
    num = ((unsigned short)(*ptr) << 9) |
        ((unsigned short)(*(ptr+1)) << 2) |
        (*(ptr+2) >> 5);

    /* Change unsigned range to signed range */
    num -= 0x8000;

    return((short)num);
}
```

NOTE: Even the last packet must have 120 data bytes in it. If a particular waveform packs up such that there aren't 120 bytes for the last packet, then that last packet's data should be padded out with 0 bytes to 120 bytes total. The receiver should ACK this last packet also.

ll is the checksum. This is the XOR of the bytes 0x7E, **cc**, 0x02, **kk**, and all 120 bytes of waveform data (with bit 7 of result masked off to 0). The receiver uses this to check that no errors occurred in the packet transmission. If so, the receiver will NAK this packet, and expect the transmitter to resend it.

ACK

F0 7E **cc** 7F **kk** F7

The receiver sends this after successfully receiving a Dump Header and after each successfully received Data Packet. It means "the last message was received correctly. Proceed with the next message". **kk** is the packet number that was received correctly (0 if responding to a Dump Header). The transmitter uses this to determine which particular packet the receiver has accepted (in case packet dumps get out of order).

NAK

F0 7E **cc** 7E **kk** F7

The receiver sends this after unsuccessfully receiving a Dump Header and after each unsuccessfully received Data Packet. It means "the last message was **not** received correctly. Resend that message". **kk** is the packet number that was received incorrectly (0 if responding to a Dump Header). The transmitter uses this to determine which particular packet the receiver has rejected (in case packet dumps get out of order).

CANCEL

F0 7E **cc** 7D **kk** F7

The receiver sends this when it wishes the transmitter to stop the dump. **kk** is the packet number upon which the dump is aborted (0 if responding to a Dump Header).

WAIT

F0 7E **cc** 7C **kk** F7

The receiver sends this when it wants the transmitter to pause the dump operation. The transmitter will send nothing until it receives another message from the receiver; an ACK to continue, a NAK to resend, or a CANCEL to abort the dump. **kk** is the packet number upon which the wait was initiated (0 if responding to a Dump Header).

This is useful for receivers which need to perform lengthy operations at certain times, such as writing data to floppy disk. If the receiver did not issue a WAIT, then the transmitter might count down its 20 millisecond timeout, and assume a non-handshaking action such as sending the next packet, without waiting for a response from the receiver. A WAIT tells the transmitter to wait indefinitely for a response.

Some people like to use computer-based wave editing software to find and set loops points. This is because the computer's large display, and mouse support, is more conducive to displaying a waveform and quickly locating satisfactory loop points, than the typically small LCD upon MIDI samplers (plus a lack of pointing devices such as a mouse). The task of finding satisfactory loop points typically involves much trial-and-error. The user has to set the start and end loop points, listen to the result, and then choose other points if the result is not yet satisfactory. Because the waveform usually has to be sent back to the sampler in order to properly judge the results, and because a MIDI Sample Dump can be a time-consuming procedure, this means that the user typically wastes a lot of time waiting for samples to be transferred. For this reason, 2 messages were added to the SDS specification. (Note that many early MIDI samplers do not support these newer messages). One message allows a transmitter (such as a computer) to ask the receiver to send only the position of the loop points for a given waveform. That means that the transmitter can quickly get information about loop points without needing to transfer an entire waveform dump. The other message allows a transmitter (such as a computer) to tell the receiver to set the start and end loop points to particular positions for a given waveform. That means that the transmitter can quickly set new loop positions without needing to transfer an entire waveform dump.

It is also possible to send/receive multiple loop points (up to 16384) in one message (as described below).

LOOP POINT TRANSMIT

F0 7E **cc** 05 01 **sl sh** *ll lh* **jj** *hl hm hh* **il im ih** *more* F7

The transmitter sends this to the receiver to set the start loop and end loop positions for a particular waveform. The receiver should set those loop positions for that waveform and ACK this message if successful. Otherwise, a NAK is returned. The **sl sh** is the waveform's 14-bit number (ie 0 to 16,384). See Dump Request.

The **ll lh** is the loop's 14-bit number (ie 0 to 16,384). Many samplers allow more than one loop to set for a given waveform, for example, there can be a sustain loop (ie, the part of the waveform looped while the user holds down a key and the sustain portion of a VCA is sustaining the sound), and a release loop (ie, the part of the waveform looped after the user releases the key and the release portion of a VCA is slowly fading out the sound). The sampler numbers the loops from 0 to how ever many loops are supported per waveform. Note that the 14-bit sample number is transmitted as 2 bytes where the first byte (ll) contains bits 0 to 6 (with high bit clear), and the second byte (lh) contains bits 7 to 13, right-justified (with high bit clear). The number of loops supported will likely vary from manufacturer to manufacturer, but a loop number of 00 00 always refers to the sustain loop. A loop number of 7F 7F is reserved to mean "delete all loops" (ie, the sampler will delete all loops that are currently set for the waveform. This is an easy way to start with a "clean slate", but note that not all samplers support this special request).

hl hm hh, **il im ih**, and **jj** are the loop start position, loop end position, and looptype. They are specified in the same way as per the Dump Header message.

It is also possible to specify more loop points (up to 16384) in one message. Where you see *more* in the above template, you could put another loop number, followed by its looptype, loop start position, and loop end position. After this, you could repeat these fields for the next loop, etc. So how does the receiver know how many loop points he is getting? Well, if he doesn't find an F7 where he expects one, then he must be dealing with the *ll* byte of the next loop's number. Therefore he should expect to find a following *lh* **jj** *hl hm hh* **il im ih** bytes. After that should be an F7, but of course, it could be yet another loop's ll byte.

LOOP POINT REQUEST

F0 7E **cc** 05 02 **sl sh** *ll lh* F7

The transmitter sends this to the receiver to ask it to send the start loop and end loop positions for a particular waveform. The receiver will then return a Loop Point Transmit message containing the requested information, or a NAK if it can't handle the request successfully. The **sl sh** is the waveform's 14-bit number (ie 0 to 16,384). See Dump Request.

The **ll lh** is the loop's 14-bit number (ie 0 to 16,384). See Loop Point Transmit.

I don't have enough information to determine what happens when you use a loop number of 7F 7F. This may cause the receiver to return a Loop Point Transmit containing all of the loops for that waveform. Or, I don't know as if you can specify several loop numbers in the above message, in order to have the receiver return all of those loops in one Loop Point Transmit message. You'll have to experiment to deduce this information. If someone does some experiments with a sampler that supports these Loop Point messages, please inform me of the results.

MIDI Time Code (MTC) is a sub-protocol within MIDI, and is used to keep 2 devices that control some sort of timed performance (ie, maybe a sequencer and a video deck) in sync. MTC messages are an alternative to using MIDI Clocks and Song Position Pointer messages. MTC is essentially SMPTE mutated for transmission over MIDI. SMPTE timing is referenced from an absolute "time of day". On the other hand, MIDI Clocks and Song Position Pointer are based upon musical beats from the start of a song, played at a specific Tempo. For many (non-musical) cues, it's easier for humans to reference time in some absolute way rather than based upon musical beats at a certain tempo.

There are several MIDI messages which make up the MTC protocol. All but one are specially defined SysEx messages.

Quarter Frame

The most important message is the *Quarter Frame* message (which is not a SysEx message). It has a status of 0xF1, and one subsequent data byte. This message is sent periodically to keep track of the running SMPTE time. It's analogous to the MIDI Clock message. The Quarter Frame messages are sent at a rate of 4 per each SMPTE Frame. In other words, by the time that a slave has received 4 Quarter Frame messages, a SMPTE Frame has passed. So, the Quarter Frame messages provide a "sub-frame" clock reference. (With 30 fps SMPTE, this "clock tick" happens every 8.3 milliseconds).

But the Quarter Frame is more than just a quarter frame "clock tick". The Quarter Frame message's data byte contains the SMPTE time (ie, hours, minutes, seconds, and frames). SMPTE time is normally expressed in 80 bits. Obviously, this is too many bits to be contained in 1 8-bit data byte. So, each Quarter Frame message contains just one piece of the time (for example, one Quarter Frame may contain only the hours). In order to get the entire SMPTE time at any given point, a slave needs to receive several Quarter Frame messages, and piece the current SMPTE time together from those messages. It takes 8 Quarter Frame messages to convey the current SMPTE time. In other words, by the time that a slave can piece together the current SMPTE time, two SMPTE frames have passed (ie, since there are 4 Quarter Frame messages in each frame). So, MTC's version of SMPTE time actually counts in increments of 2 SMPTE Frames per each update of the current SMPTE time.

The first (of 8) Quarter Frame message contains the low nibble (ie, bits 0 to 3) of the Frame Time. The second Quarter Frame message contains the high nibble (ie, bits 4 to 7) of the Frame Time. The third and fourth messages contain the low and high nibbles of the Seconds Time. The fifth and sixth messages contain the low and high nibbles of the Minutes Time. The seventh and eighth messages contain the low and high nibbles of the Hours Time. The eighth message also contains the SMPTE frames-per-second Type (ie, 24, 25, 30 drop, or 30 fps). If you were to break up the Quarter Frame's data byte into its 7 bits, the format is:

0nnn dddd

where **nnn** is one of 7 possible values which tell you what **dddd** represents. Here are the 7 values,

and what each causes **dddd** to represent.

```
Value           dddd
  0         Current Frames Low Nibble
  1         Current Frames High Nibble
  2         Current Seconds Low Nibble
  3         Current Seconds High Nibble
  4         Current Minutes Low Nibble
  5         Current Minutes High Nibble
  6         Current Hours Low Nibble
  7         Current Hours High Nibble and SMPTE Type
```

0xF1 0x25

means that the 5 is the low nibble of the Seconds Time (because **nnn** is 2). If the following Quarter Frame is subsequently received,

0xF1 0x32

then this means that 2 is the high nibble of the Seconds Time. Therefore, the current SMPTE Seconds is 0x25 (ie, 37 seconds).

In the data byte for the Hours High Nibble and SMPTE Type, the bits are interpreted as follows:

0nnn **x** yy **d**

where **nnn** is 7. **x** is unused and set to 0. **d** is bit 4 of the Hours Time. **yy** tells the SMPTE Type as follows:

```
0 = 24 fps
1 = 25 fps
2 = 30 fps (Drop-Frame)
3 = 30 fps
```

When MTC is running in the forward direction (ie, time is advancing), the Quarter Frame messages are sent in the order of Frames Low Nibble to Hours High Nibble. In other words, the order looks something like this:

```
0xF1 0x0n
   where n is the current Frames Low Nibble
```

```
0xF1 0x1n
   where n is the current Frames High Nibble
```

```
0xF1 0x2n
```

```
   where n etc.
```

0xF1 0x3**n**

0xF1 0x4**n**

0xF1 0x5**n**

0xF1 0x6**n**

0xF1 0x7**n**

When MTC is running in reverse (ie, time is going backwards), these are sent in the opposite order, ie, the Hours High Nibble is sent first and the Frames Low Nibble is last.

The arrival of the 0xF1 0x0**n** and 0xF1 0x4**n** messages always denote where SMPTE Frames actually occur in realtime.

Since 8 Quarter Frame messages are required to piece together the current SMPTE time, timing lock can't be achieved until the slave has received all 8 messages. This will take from 2 to 4 SMPTE Frames, depending upon when the slave comes online.

The Frame number (contained in the first 2 Quarter Frame messages) is the SMPTE Frames Time for when the first Quarter Frame message is sent. But, because it takes 7 more Quarter Frames to piece together the current SMPTE Time, when the slave does finally piece the time together, it is actually 2 SMPTE Frames behind the real current time. So, for display purposes, the slave should always add 2 frames to the current time.

Full Frame

For cueing the slave to a particular start point, Quarter Frame messages are not used. Instead, an MTC *Full Frame* message should be sent. The Full Frame is a SysEx message that encodes the entire SMPTE time in one message as so (in hex):

F0 7F **cc** 01 01 **hr** *mn* **sc** *fr* F7

cc is the SysEx channel (0 to 127). It is suggested that a device default to using its Manufacturer's SysEx ID number for this channel, giving the musician the option of changing it. Channel number 0x7F is used to indicate that all devices on the daisy-chain should recognize this Full Frame message.

The **hr**, **mn**, **sc**, and **fr** are the hours, minutes, seconds, and frames of the current SMPTE time. The hours byte also contains the SMPTE Type as per the Quarter Frame's Hours High Nibble message.

The Full Frame simply cues a slave to a particular SMPTE time. The slave doesn't actually start running until it starts receiving Quarter Frame messages. (Which implies that a slave is stopped whenever it is not receiving Quarter Frame messages). The master should pause after sending a Full Frame, and before sending a Quarter Frame, in order to give the slave time to cue to the desired SMPTE time.

During fast forward or rewind (ie, shuttle) modes, the master should not continuously send Quarter Frame messages, but rather, send Full Frame messages at regular intervals.

User Bits

SMPTE also provides for 32 "user bits", information for special functions which vary with each product. (Usually, these bits can only be programmed from equipment that supports such). Upto 4 characters or 8 digits can be written. Examples of use are adding a date code or reel number to a tape. The user bits tend not to change throughout a run of time code, so rather than stuffing this information into a Quarter Frame, MTC provides a separate SysEx message to transmit this info.

F0 7F **cc** 01 02 **u1 u2 u3 u4 u5 u6 u7 u8 u9** F7

cc is the SysEx channel (0 to 127). Only the low nibble of each of the first 8 data bytes is used. Only the 2 low bits of u9 is used.

u1 = 0000**aaaa**
u2 = 0000**bbbb**
u3 = 0000**cccc**
u4 = 0000**dddd**
u5 = 0000**eeee**
u6 = 0000**ffff**
u7 = 0000**gggg**
u8 = 0000**hhhh**
u9 = 000000**ii**

These nibbles decode into an 8-bit format of aaaabbbb ccccdddd eeeeffff gggghhhh ii. It forms 4 8-bit characters, and a 2 bit Format Code. u1 through u8 correspond to the SMPTE Binary Groups 1 through 8. u9 are the 2 Binary Group Flag Bits, defined by SMPTE.

The Users Bits messages can be sent at any time, whenever these values must be passed to some device on the daisy-chain.

Notation Information

There are two Notation Information messages which can be used to setup a device that needs to interact with the musician using musical bars and beats.

Time Signature

The Time Signature message can setup Time Signature or indicate a change of meter.

F0 7F **cc** 03 *ts* **ln** *nn dd* **qq** [nn dd...] F7

cc is the SysEx channel (0 to 127).

ts is 02 if the Time Signature is to be changed now, or 42 if the Time Signature is to be changed at the end of the currently playing measure.

ln is the number of data bytes following this field. Normally, this will be a 3 if there is not a compound time signature in the measure.

nn dd are the Numerator and Denominator of the Time Signature, respectively. Like with MIDI File Format's Time Signature MetaEvent, the Denominator is expressed as a power of 2.

qq is the number of notated 32nd notes in a MIDI quarter note. Again, this is similiar to the same field in MIDI File Format's Time Signature MetaEvent.

[nn dd ...] are optional, additional pairs of num/denom, to define a compound time signature within the same measure.

Bar Marker

The Bar Marker message indicates the start of a musical measure. It could also be used to setup and mark off bars of an introductory "count down".

F0 7F **cc** 03 01 **lb mb** F7

cc is the SysEx channel (0 to 127).

lb mb is the desired bar number, with the LSB first (ie, Intel order). This is a signed 14-bit value (low 7 bits are in lb, right-justified, and bits 8 to 14 are in mb, right-justified). Zero and negative numbers up to -8,190 indicate count off measures. For example, a value of -1 (ie, lb mb = 7F 7F) means that there is a one measure introduction. A value of zero would indicate no count off. Positive values indicate measures of the piece. The first measure is bar 1 (ie, lb mb = 01 00). A maximum neg number (lb mb = 00 40) indicates "stopped play" condition. A maximum positive value (lb mb = 7E 3F) indicates running condition, but no idea about measure number. This would be used by a device

wishing to mark the passage of measures without keeping track of the actual measure number.

Setup Message

The Setup message can be used to implement one of 19 defined "events". A master device uses this message to tell slave units what "events" to perform, and when to perform those events. Here's the general template for the message.

F0 7F **cc** 04 *id* **hr mn sc fr ff** *sl sm* **[more info]** F7

cc is the SysEx channel (0 to 127).

hr mn sc fr ff is the SMPTE time when the event is to occur. This is just like the Full Frame message, except that there is also a fractional frame parameter, ff, which is 1/100 of a frame (ie, a value from 0 to 99).

sl sm is this event's 14-bit Event Number (0 to 16,383). sl is bits 0 to 6, and sm is bits 7 to 13.

id tells what this Event Type is. Depending upon the Type, the message may have additional bytes placed where is. The following values for Event Types are defined, and here's what each does.

Special (00)

Contains the setup information that affects a device globally, as opposed to individual tracks, sounds, programs, sequences, etc.). In this case, the Event Number is actually a word which further describes what the event is, as so:

Time Code Offset (00 00) refers to a relative Time Code offset for each unit. For example, a piece of video and a piece of music that are supposed to go together may be created at different times, and likely have different absolute time code positions. Therefore, one must be offset from the other so that they will match up. Each slave on the daisy-chain needs its own offset so that all can be matched up to the master's SMPTE start time.

Enable Event List (01 00) means for a slave to enable execution of events in its internal "list of events" when each one's respective SMPTE time occurs.

Disable Event List (02 00) means for a slave to disable execution of events in its internal "list of events", but not to erase the list.

Clear Event List (03 00) means for a slave to erase all events in its internal list.

System Stop (04 00) refers to a time when the slave may shut down. This serves as a protection against Event Starts without Event Stops, tape machines running past the end of a reel, etc.

Event List Request (05 00) is sent by the master, and requests the slave (whose channel matches the message) to send all events in its list as a series of Setup messages, starting from the SMPTE time in this message.

NOTE: For the first 5 Special messages, the SMPTE time isn't used and is ignored.

Punch In (01) and Punch Out (02)

These refer to the enabling and disabling of record mode on a slave. The Event Number refers to the track to be recorded. Multiple Punch In and Punch Out points (and any of the other Event Types below) may be specified by sending multiple Setup messages with different SMPTE times.

Delete Punch In (03) and Delete Punch Out (04)

Deletes the Punch In or Punch Out (with the matching Event Number and SMPTE Time) from the slave's event list. In other words, it deletes a previously sent Punch In or Punch Out Setup message.

Event Start (05) and Event Stop (06)

These refer to the start/stop (ie, playback) of some continuous action (ie, an action that begins when an Event Start is received, and continues until an Event Stop is received). The Event Number refers to which action on the slave is to be started/stopped. Such actions may include playback of a specific looped waveform, a fader moving on an automated mixer, etc.

Event Start (07) and Event Stop (08) with additional info

Almost the same as the above 2 Event Types, but these have additional bytes before the final 0xF7. Such additional bytes could be for an effect unit's changing parameters, the volume level of a sound effect being adjusted, etc. The additional info should be nibblized with the lowest bits first. For example, if the Note On message 0x91 0x46 0x7F was to be encoded in some additional info bytes, they would be 0x01 0x09 0x06 0x04 0x0F 0x07.

Delete Event Start (09) and Delete Event Stop (0A)

Deletes the Event Start or Event Stop (with the matching Event Number and SMPTE Time) from the slave's event list. In other words, it deletes a previously sent Event Start or Event Stop Setup message (either the Types without additional info, or with additional info).

Cue Point (0B)

Sets an action to be triggered (ie, an action that does something once and automatically stops

afterward) or a marker at the specified SMPTE time. These include a "hit" point for a sound effect, a marker for an edit point, etc. The Event Number should represent the action or marker. For example, Event Number 3 could be to trigger a car crash sound effect. Then, several car crashes could be specified by sending several Cue Point Setup messages, each with Event Number 3, but different SMPTE times.

Cue Point (0C) with additional info

Like the above, but this message may have additional bytes before the final 0xF7. Such additional bytes could be for an effect unit's parameters, the volume level of a sound effect, etc. The additional info should be nibblized with the lowest bits first.

Delete Cue Point (0D)

Deletes one of the preceding 2 Setup messages (with the same Event Number and SMPTE time) from the slave's event list.

Event Name (0E) with additional info)

This assigns an ascii name to the event with the matching Event Number and SMPTE time. It for the musician's point of reference. The additional info bytes are the ascii name. For a newline character, include both a carriage return (0x0A) and line feed (0x0D). The ascii bytes are nibblized. For example, ascii 'A' (0x41) becomes the two bytes, 0x01 0x04.

Summary of Play Mode

To summarize the interaction between master and slave depending upon "play mode":

Play Mode

The master is in normal play at normal or vari-speed rates. The master is sending Quarter Frame messages to the slave. The messages are in ascending order, starting with 0xF1 0x0**n** and ending with 0xF1 0x7**n**. If the master is capable of reverse play, then the messages are sent in reverse, starting with 0xF1 0x7**n** and ending with 0xF1 0x0**n**.

Cue Mode

The master is being "rocked" or "cued" by hand. For example, a tape machine may have the tape still in contact with the playback head so that the musician can cue the contents of the tape to a specific point. The master is sending Quarter Frame messages to the slave. The messages are in ascending order, starting with 0xF1 0x0**n** and ending with 0xF1 0x7**n**. If the master is playing in a reverse

direction, then the messages are sent in reverse, starting with 0xF1 0x7**n** and ending with 0xF1 0x0**n**. Because the musician may be changing the tape direction rapidly, the order of the Quarter Frames must change along with the tape direction.

Fast Forward or Rewind Mode

The master is rewinding or fast forwarding tape. No contact is made with the playback head. So, no cueing is happening. Therefore, the master only need send the slave periodic Full Frame messages at regular intervals as a rough indication of the master's position. The SMPTE time indicated by the last Full Frame message actually takes affect upon the reception of the next Quarter Frame message (ie, when Play Mode resumes).

Introduction

Most MIDI sound modules today are "multi-timbral". This means that the module can listen to all 16 MIDI channels at once, and play any 16 of its "patches" simultaneously, with each of the 16 patches set to a different MIDI channel.

It's as if the module had 16 smaller "sub-modules" inside of it. Each sub-module plays its own patch (ie, instrument) upon its own MIDI channel.

MIDI Channels and the Multi-Timbral Module

It may help if I draw some analogies here to explain the above.

Think of these sub-modules as robotic musicians. I'll call them "robomusicians". You have 16 of them inside one multi-timbral module.

Now think of MIDI channels as channels (ie, inputs) upon a mixing console. You have 16 of them in any one MIDI setup. (I assume one discrete MIDI bus in this "MIDI setup". Some setups have multiple MIDI Ins/Outs with more than 16 MIDI channels. But here, let's talk about a typical MIDI setup which is limited to 16 channels).

Each robomusician (ie, sub-module) has his own microphone plugged into one channel of that 16 channel mixer, so you have individual control over his volume, panning, reverb and chorus levels, and perhaps other settings.

The diagram below illustrates the concept of a multi-timbral module in my MIDI setup. There are 16 "robomusicians" representing the 16 sub-modules. And there are 16 channels on the mixer, representing the 16 MIDI channels. Each robomusician has his own channel on the mixer, which means he has his own volume, pan, reverb level, and other such settings.
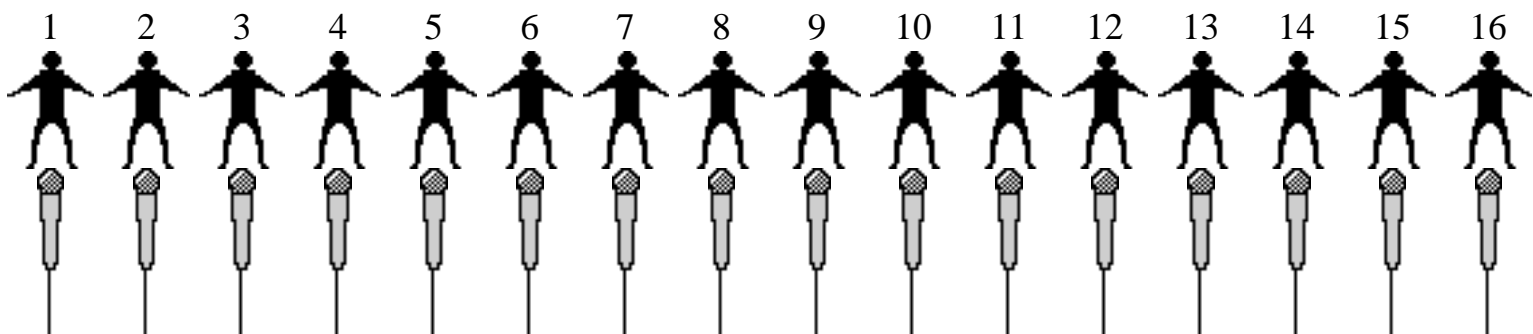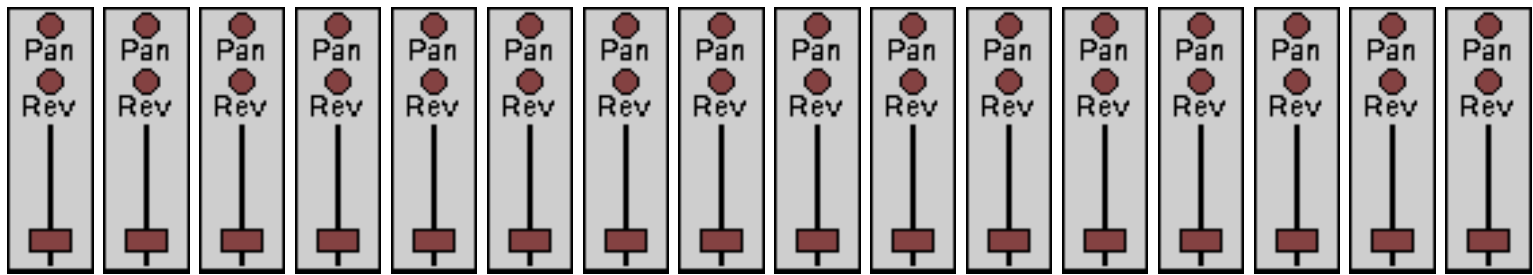
But MIDI Channels aren't exactly like inputs on a mixer, because not only are they inputs from those robomusicians, they are also outputs to those robomusicians. In other words, let's say that there is a button on each of those mixer channels. When you push that button and speak into a microphone, your voice is heard through the headphones worn by only the one robomusician plugged into that channel. For example, if you push the button on channel 10, only robomusician 10 hears what you say to him. In this way, you can give individual instructions to each robomusician.

Assigning a patch to a "robomusician"

Think of a patch as a "musical instrument". For example, you typically have Piano, Flute, Saxophone, Bass Guitar, etc, patches in a sound module (even the ones built into a computer sound card -- often referred to as a "wavetable synth"). Typically, most modules have hundreds of patches (ie, musical instruments) to chose from. The patches are numbered. For example, a trumpet patch may be the fifty-seventh patch available among all of the choices.

Since you have 16 robomusicians, you can pick out any 16 instruments (ie, patches) among those hundreds, to be played simultaneously by your 16 robomusicians. Each robomusician can of course play only one instrument at a time. (On the other hand, each robomusician can play chords upon any instrument he plays, even if it's traditionally an instrument that can't play chords. For example, if the robomusician plays a trumpet patch, he can play chords on it, even though a real trumpet is incapable of sounding more than one pitch at a time).

As an example, maybe your arrangement needs a drum kit, a bass guitar, a piano, and a saxophone. Let's say that the drums are played by robomusician #10. (He's on MIDI channel 10 of the mixer). (In fact, with some MIDI modules, channel #10 is reserved for only drums. In other words, robomusician #10 can play only drums, and maybe he's the only robomusician who can play the drums). The other robomusicians are super musicians. Each robomusician can play any of the hundreds of instruments (ie, patches) in your module, but of course, he still is restricted to playing only one instrument at a time. So let's say that you tell robomusician 1 to sit at a piano, and robomusician 2 to pick up a bass guitar, and robomusician 3 to pick up a saxophone. Let's say that you tell the remaining 12 robomusicians to pick up an accordian, violin, acoustic guitar, flute, harp, cello, harmonica, trumpet, clarinet, etc, so that each robomusician has a different instrument to play.

Here's what we want the instrument assignment to look like:
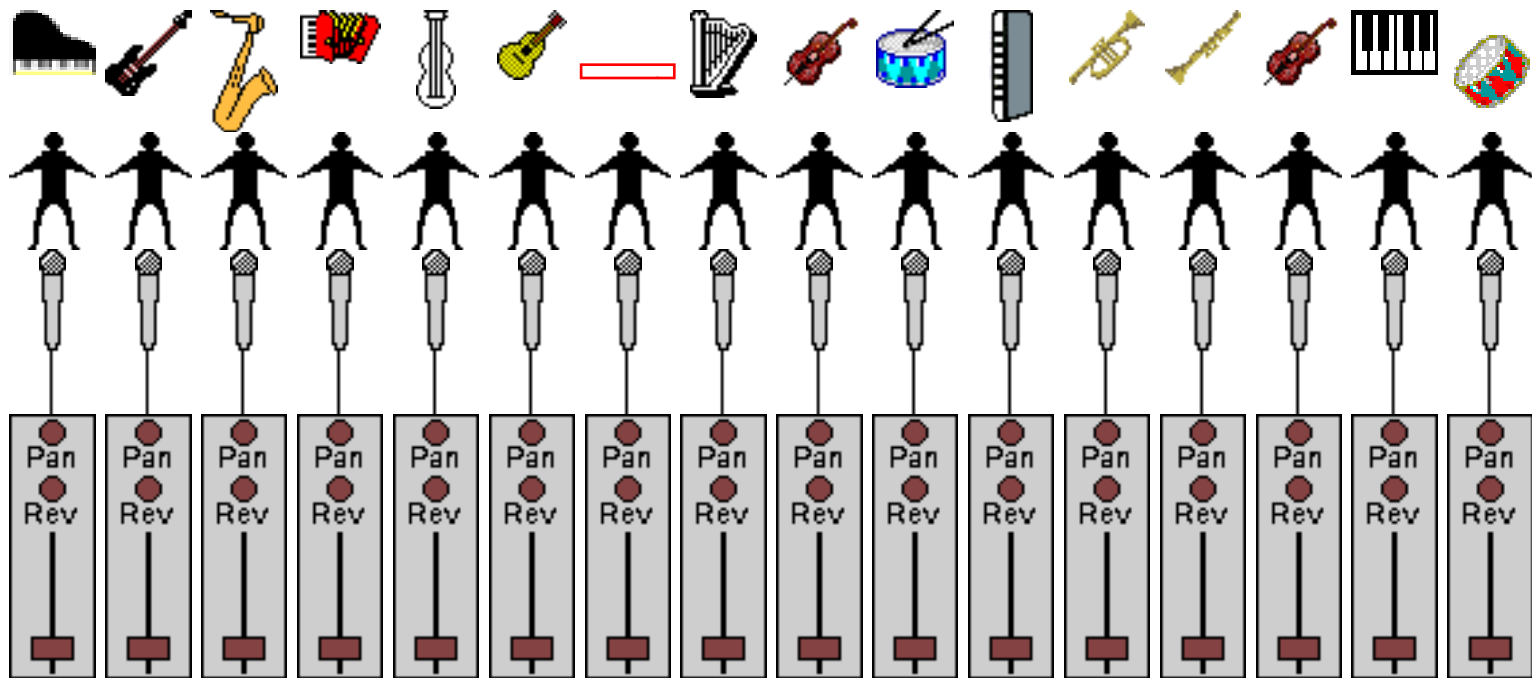
How do you tell the robomusician to pick up a certain instrument? Hit that button upon his channel and give him a message telling him the number of the patch/instrument you want him to play. How do you do that over MIDI? Well, that's what MIDI messages are for. The MIDI Program Change message is the one that instructs a robomusician to pick up a certain instrument. Contained in the MIDI Program Change message is the number of the desired patch/instrument. (For example, above that would be #57 for the Trumpet patch). So, you send (to the multi-timbral module's MIDI In) a MIDI Program Change message upon the MIDI channel for that robomusician. For example, to tell robomusician 3 to pick up a sax, you send a MIDI Program Change (with a value that selects the Saxophone patch) on MIDI channel 3.

Note: To discover what value (ie, number) you need for the Program Change message, in order to select a particular patch, consult the manual for your sound module. If your module follows the General MIDI Patch set, then consult that standard for what numbers select which patches.

Individual control via each MIDI Channel

After you've told the 16 robomusicians what instruments to pick up, you can now have them play a MIDI arrangement with these 16 instruments -- each robomusician playing simultaneously with individual control over his volume, panning, etc.

How do you tell a robomusician what notes to play? You send him MIDI Note messages on his channel. Remember that only that one robomusician "hears" these messages. The other robomusicians see only those messages on their respective channels. (ie, Each robomusician ignores messages that aren't on his channel, and takes notice of only those messages that are on his channel). For example, the sax player is robomusician 3, so you send him note messages on MIDI channel 3.

How do you tell a robomusician to change his volume? You send him Volume Controller messages

on his MIDI channel. How do you tell a robomusician to bend his pitch? You send him Pitch Wheel messages on his MIDI channel. In fact, there are many different things that a robomusician can do independently of the other 15 robomusicians, because there are many different MIDI controller messages that can be sent on any given MIDI channel.

And that's why I say that it's as if there are are 16 "sub-modules" inside of one multi-timbral module -- because these 16 robomusicians really do have independent control over their musical performances, thanks to there being 16 MIDI channels in that one MIDI cable that runs to the multi-timbral module's MIDI In.

Changing instrumentation

OK, let's say that at one point in your arrangement, a 17th instrument needs to be played -- maybe a Banjo. Well, at that point you've got to have one of your 16 robomusicians put down his current instrument and pick up a Banjo instead. Let's say that the sax player isn't supposed to be playing anything at this point in the arrangement. So, you send a MIDI Program Change to robomusician 3 (ie, on MIDI channel 3 -- remember that he's the guy who was playing the sax), telling him to pick up a Banjo. Now when you send him note messages, he'll be playing that banjo. Later on, you can send him another MIDI Program Change to tell him to put down the Banjo and pick up the saxophone again (or some other instrument). So, although you're limited to 16 robomusicians playing 16 instruments simultaneously, any of your robomusicians can change their instruments during the arrangement. (Well, maybe robomusician 10 is limited to playing only drums. Even then, he may be able to choose from among several different drum kits).

Parts

So is there a name for these 16 "robomusicians" or "sub-modules" inside of your MIDI module? Well, different manufacturers refer to them in different ways, and I'm going to use the Roland preference, a **Part**. A Roland multi-timbral module has 16 Parts inside of it, and each usually has its own settings for such things as Volume, Panning, Reverb and Chorus levels, etc, and its MIDI channel (ie, which MIDI data the Part "plays"). Furthermore, each Part has its own way of reacting to MIDI data such as *Channel Pressure* (often used to adjust volume or brightness), *MOD Wheel controller* (often used for a vibrato effect), and *Pitch Wheel* (used to slide the pitch up and down). For example, one Part can cause its patch to sound brighter when it receives Channel Pressure messages that increase in value. On the other hand, another Part could make its volume increase when it receives increasing Channel Pressure messages. These Parts are completely independent of each other. Just because one Part is receiving a Pitch Wheel message and bending its pitch doesn't mean that another Part has to do the same.

Stereo output

You'll note that sound of all 16 robomusicians typically comes out of a stereo output of your sound module (or computer card). That's because most multi-timbral modules have an internal mixer (which can be adjusted by MIDI controller messages to set volume, panning, brightness, reverb level, etc) that mixes the output of all 16 Parts to a pair of stereo output jacks. (ie, The 16 microphones and 16 channel mixing console I alluded to earlier are built into the MIDI module itself. The stereo outputs of the module are like the stereo outputs of that mixing console).

Recommended reading:

What's MIDI?

I still remember back in the old days before computers and MIDI became the useful musical tools that they are now, and I **definitely** don't want to go back to those days.

Before the days of MIDI sequencers, and keyboards with realistic-sounding samples of instruments, in order to even hear what one of my arrangements sounded like, I had to assemble all of the musicians required to perform the piece, and get them to play it for me. If I didn't have access to a multi-track tape recorder, and a mixing console, etc, that meant that I had to get them all together simultaneously to play the piece live in order to hear the full arrangement. If one of them couldn't show up at a scheduled time (and musicians are notorious for not being able to stick to a schedule), then it was tough luck for me. I had to wait until the guy finally got around to playing his part so that I could actually hear what my arrangment **really** sounded like. In the meantime, I had to twiddle my thumbs. And how else can a composer know if his arrangement is what he wants until he hears it? Of course, if I decided to rewrite the parts after hearing the arrangement actually played, that meant that I had to reassemble the musicians again at a later date to hear the new arrangement (and usually, they don't want to have to redo something unless you're paying them, which can be expensive, especially if you need to "motivate" them with cocaine).

MIDI/computers make it possible for a composer to hear his arrangement played without needing other musicians. It thereby frees the composer from scheduling problems, and allows him more opportunity to audition his composition during the composition process, which provides invaluable feedback to aid in composition.

I've **still** got a stack of arrangements sitting around from years back which I've never even heard because I can't find the musicians to play the scores. The music I write often requires some difficult technique. You know how hard it is to find musicians who can play this stuff in a small town like the one in which I live? Almost impossible. And that's assuming that the musicians in this area even **want** to play the music I've written. I write "progressive rock" which is not all that popular music. Most musicians prefer to play music they like, and that would be "popular music". I don't write pop music. So it's doubly hard for me to find musicians since I write difficult-to-play, esoteric music.

MIDI/computers made it easier for composers to hear difficult-to-execute, or esoteric, music. It is often easier to have a computer play a difficult performance than it is to find a musician capable of playing that performance. Likewise, it is often easier to have a computer play esoteric music (ie, unusual instrumentation) than find musicians who have appropriate skills and the desire to play that music.

And even if you could get all of the musicians, the time it took to do a multi-track recording on the old analog equipment, what with setting up microphones, running cables, rewinding the tape, etc, takes **so** much longer then pressing the record button on an electronic keyboard with a built-in sequencer and its own samples of instruments, and immediately playing your part. You write a long piece (such as my 8 minute long "Joan of Arc" arrangement on my web site) and you can spend minutes just waiting for the tape to rewind to hear just one of what will likely be hundreds of "takes" to build up the parts. And don't forget that with analog tape, you can't zero on an individual musical event and "correct" the wrong note just by entering the "correct" MIDI digital value like you can with a sequencer. If you make a mistake, you have erase over it, and literally play it again. To save time,

we did "punch ins" (ie, recording over the area on the tape only where the mistake occurs), but this is hard to do. If the guy pressing the record button didn't happen to press it at just the perfect moment (ie, we didn't have a computer to automatically punch in at a certain time -- we had to do it manually -- so the musician playing the overdub always needed someone else around to press buttons), you could miss the attack on a note, or destroy the release on a previously recorded note that you didn't want to record over. And if you blew it, then you had to back up the tape and punch in earlier (ie, each time you made a mistake, you had to start earlier in the piece, forcing you to replay sections that were good, thus perhaps introducing a mistake in those section). Suffice it to say, that punch-ins and redoing takes, and trying to get rid of mistakes on analog tape is **much, much** more difficult and time-consuming than with a MIDI sequencer. And wasting time correcting mistakes means that you have less time to do more productive things.

And don't forget that with analog tape, you can't slow it down, record your part at a slower tempo (so that it's easier to play), and then speed the tape up for playback. If you do that, it changes the quality of the sound (ie, changes the overtones, vibrato speed, tremulo speed, etc). You end up with a "Mickey Mouse" effect.

Oh, and if you need to change the key (ie, transpose the parts) after you recorded all of the parts, because the singer has trouble hitting the "high notes"? Too bad. Start all over again from the beginning.

Let me tell you, the first thing I did when I switched to using MIDI to play my arrangements was **throw away my analog, multi-track reel-to-reel tape recorder.** Good riddance!

MIDI/computers allow for more efficient recording sessions by streamlining/improving the process of correcting mistakes in the recording, and streamlining the recording process itself. MIDI/computers also eliminate the need to have an engineer with special skills, as the computer takes over some of those functions, such as "punching in" overdubs. MIDI/computers also offer efficient ways to make useful changes, such as transposing key.

And if you wanted a really professional sounding recording of your music, you had to either buy really expensive recording equipment, or you had to rent time in expensive recording studios. Remember that back in the old days, we didn't have ICs with really dense circuitry. Manufacturers had to use a lot more electronics to deliver less. I bought a Prophet 5 synthesisor. It had analog, basic waveforms that could maybe produce somewhat tolerable renditions of real instruments. It cost me $5,500 for that one synth. Today, you can get better sound than that synth out of a $100 computer audio card. I'm not kidding you. I was there. Trying to get a good sax sound out of that synth, for example, took **days** of programming it (ie, some of the early synths didn't even have programmable, battery-backed memory so you lost all of your settings every time you turned it off), and then you'd get something that wasn't even **nearly** as realistic sounding as what you can get in a cheap (couple of hundred dollars), modern instrument with digital sampling (ie, the technology used on today's computer sound cards).

And the price of going into a recording studio? We're talking **hundreds of dollars an hour**. You could go broke before the microphones were even setup in front of all of the instruments. You want

to know how they got the drum sounds that you hear out of your basic sound card? They had to individually tune each drumhead, and dampen its overtones (toilet paper taped to the drum head in certain spots worked great -- I was a recording engineer years ago, in case you're wondering). It typically took a minimum of 2 hours to tune an entire kit, just to get the sounds that you can get out of a digital sampler today within seconds of turning it on. And you had to **pay** for that 2 hours of time. And don't forget the musicians' salaries who are playing for your session.

MIDI/computers allow for professional recordings to be made on a budget, in small, "home" environments that don't require things like special soundproofing. Furthermore, the technology in computer audio often eliminates expensive, time-consuming tasks, such as tuning drumkits and placing microphones/cables. The storage capacity of computers also makes recalling a given setup very efficient and inexpensive, which is important since often a setup needs to be recalled during multiple sessions.

Now contrast this with MIDI (ie, computer sequencers using MIDI, and computer sound cards with the ability to play numerous musical parts with realistic-sounding instruments). With MIDI, I can play all of the musical parts myself. (Well, really the MIDI sequencer is my "band". It obeys all of my instructions, whenever I want, even when I get a great musical idea at 3AM. Try assembling a band at 3AM in order to check out a musical idea. Good luck. With MIDI, I plug in my headphones, and start arranging my great idea immediately, and my sleeping neighbors don't even know that I'm arranging **and listening to the playback** of my musical piece written for a 40 piece orchestra).

With a MIDI sequencer, you press the rewind button and it's instantaneously ready for playback. No waiting for rewinding tape. You can have the sequencer automatically punch-in for you. (ie, You don't need another guy to punch the record button while you're busy playing your overdub). You can fix your mistakes just by quickly editing the individual "musical events". Most sequencers can display the data in the representation of a musical manuscript upon the computer's display. You just click on the graphical notes with the computer "mouse" to edit them. There's no equivalent to that in the old analog days.

And you can slow down the tempo to record your part. Speed it up for playback, and no Mickey Mouse effect.

And transposing? The computer does it for me. I just tell it how many half steps up or down, or tell it what new key signature to use.

And it's helped improve my arranging immensely. For one thing, I get instantaneous feedback on my arrangements **while** I'm arranging, just by pressing the play button on the sequencer. I don't have to wait for musicians to play my parts. (And I don't have to pay them either because "them" is my multi-timbral MIDI module playing all of their parts on all of their instruments. For the price of hiring musicians for one recording session, I've purchased equipment that I can use for hundreds of sessions). And I can play instruments myself that I never could play before. I can play the violin. I can play the flute. I can play the trumpet. I can play the timpani. I can play an elkherders' moohoo

horn. How? Because I've got a keyboard that has realistic-sounding renditions of all of those instruments. (Well, not the moohoo horn. It's hard to find a good moohoo horn sample). And, I can play **that**. It really helps to get a feel for how to use an instrument in an arrangement when you actually get to "play" it yourself, listening to how it sounds in various ranges (ie, good samplers use multiple samples to adequately reproduce the character of the instrument across its note range) and hearing how it blends in with other instruments in an arrangement.

I'm finally getting to hear some of those old arrangements, because now I've stopped waiting for Rich or Mike to play my guitar solo (which both said that he would do years ago but never got around to learning and recording the part. Damn those useless guitar players! I say, we keyboard players should strive to make them obsolete as we did with drummers, violinists, etc. After all, it's musical darwinism, so it can't be stopped. The weaker string strummers and skin beaters were destined to be replaced by the stronger ivory ticklers). I'm playing it myself using MIDI and a good sampler with a guitar sound. And my MIDI module never complains about what I'm forcing it to play. It never says "Hey, I don't like this esoteric, weird-sounding music you write, and I don't want to play it". MIDI has been a boon to musicians who write esoteric music. Check out Frank Zappa's Civilization Phase III, done entirely on a synclavier (ie, really fancy digital sampler with built-in sequencer). I doubt you could find many musicians who could play that music, and probably none who could duplicate those highly-electronically-processed sounds with traditional instruments (even though he is manipulating samples of those traditional instruments).

With MIDI, I make recordings that sound like they were done in an expensive professional recording studio, and I do it in my home at a fraction of the cost and time and physical space. I wish I had it all **years** ago.

By the way, did I mention that I used to have to walk 5 miles to school everyday through 10 foot high ridges of freezing snow (when I was a four foot tall kid)? You young "MIDI kids" of today have it **easy**... (Hey, I had to listen to that crap from adults when I was a kid. Now it's **your** turn).

Standard MIDI File (SMF) Format

The **Standard MIDI File** (SMF) is a file format used to store MIDI data (plus some other kinds of data typically needed by a <u>sequencer</u>).

This format stores the standard MIDI messages (ie, status bytes with appropriate data bytes) plus a time-stamp for each message (ie, a series of bytes that represent how many clock pulses to wait before "playing" the event). The format also allows saving information about tempo, time and key signatures, the names of tracks and patterns, and other information typically needed by a sequencer. One SMF can store information for numerous patterns and tracks so that any sequencer can preserve these structures when loading the file.

NOTE: A **track** usually is analogous to one musical part, such as a Trumpet part. A **pattern** would be analogous to all of the musical parts (ie, Trumpet, Drums, Piano, etc) for one song.

The format was designed to be generic so that the most important data can be read by all sequencers. Think of a MIDI file as a musical version of an ASCII text file (except that the MIDI file contains binary data), and the various sequencer programs as text editors all capable of reading that file. But, unlike ASCII, MIDI file format saves data in **chunks** (ie, groups of bytes preceded by an ID and size) which can be parsed, loaded, skipped, etc. Therefore, SMF format is flexible enough for a particular sequencer to store its own proprietary, "extra" data in such a way that another sequencer won't be confused when loading the file and can safely ignore this extra stuff that it doesn't need. For example, maybe a sequencer wants to save a "flag byte" that indicates whether the user has turned on an audible metronome click. The sequencer can save this flag byte in such a way that another sequencer can skip this byte without having to understand what that byte is for. In the future, the SMF format can also be extended to include new "official" chunks that all sequencer programs may elect to load and use. This can be done without making old data files obsolete, nor making old sequencers no longer able to load the new files. So, the format is designed to be extensible in a backwardly compatible way.

Of course, SMF files may be used by other MIDI software than just sequencers. Since SMF files can store any and all types of MIDI messages, including System Exclusive messages, they may be used to store/load data by all kinds of MIDI software, such as a Patch Editor that wants to save some System Exclusive messages it received from a

MIDI module. (The "timestamp" for each message may be irrelevant to such a Patch Editor. But it's easily ignored for programs that don't really need it).

In conclusion, any software that saves or loads MIDI data should use SMF format for its data files.

Data is always saved within a **chunk**. There can be many chunks inside of a MIDI file.

Each chunk can be a different size (and likely will be). A chunk's size is how many (8-bit) bytes are contained in the chunk.

The data bytes in a chunk are typically related in some way. For example, all of the bytes in one chunk may be for one particular sequencer track. The bytes for another sequencer track may be put in a different chunk. So, a chunk is simply a group of related bytes.

Each chunk must begin with a 4 character (ie, 4 ascii bytes) **ID** which tells what "type" of chunk this is.

The next 4 bytes must form a 32-bit length (ie, size) of the chunk.

All chunks must begin with these two fields (ie, 8 bytes), which are referred to as the **chunk header**.

Here's what a chunk's header looks like if you defined it in C:

```
struct CHUNK_HEADER
{
   char            ID[4];
   unsigned long  Length;
};
```

NOTE: The **Length** does not include the 8 byte chunk header. It simply tells you how many bytes of data are in the chunk following this header.

And here's an example chunk header (with bytes expressed in hex) if you examined it with a hex editor:

4D 54 68 64 00 00 00 06

Note that the first 4 bytes make up the ascii ID of **MThd** (ie, the first four bytes are the ascii values for 'M', 'T', 'h', and 'd'). The next 4 bytes tell us that there should be 6 more data bytes in the chunk (and after that we should find the next chunk header or the end of the file).

NOTE: The 4 bytes that make up the **Length** are stored in (Motorola) "Big Endian" byte order, not (Intel) "Little Endian" reverse byte order. (ie, The 06 is the fourth byte

instead of the first of the four).

In fact, all MIDI files begin with the above **MThd header** (and that's how you know that it's a MIDI file).

The MThd header has an ID of **MThd**, and a Length of **6**.

Let's examine the 6 data bytes (which follow the MThd header) in an MThd chunk.

The first two data bytes tell the **Format** (which I prefer to call "type"). There are actually 3 different types (ie, formats) of MIDI files. A type of 0 means that the file contains one single track containing midi data on possibly all 16 midi channels. If your sequencer sorts/stores all of its midi data in one single block of memory with the data in the order that it's "played", then it should read/write this type. A type of 1 means that the file contains one or more simultaneous (ie, all start from an assumed time of 0) tracks, perhaps each on a single midi channel. Together, all of these tracks are considered one sequence or pattern. If your sequencer separates its midi data (i.e. tracks) into different blocks of memory but plays them back simultaneously (ie, as one "pattern"), it will read/write this type. A type of 2 means that the file contains one or more sequentially independant single-track patterns. If your sequencer separates its midi data into different blocks of memory, but plays only one block at a time (ie, each block is considered a different "excerpt" or "song"), then it will read/write this type.

The next 2 bytes tell how many tracks are stored in the file, **NumTracks**. Of course, for format type 0, this is always 1. For the other 2 types, there can be numerous tracks.

The last two bytes indicate how many Pulses (i.e. clocks) Per Quarter Note (abbreviated as PPQN) resolution the time-stamps are based upon, **Division**. For example, if your sequencer has 96 ppqn, this field would be (in hex):

00 60

NOTE: The 2 bytes that make up the **Division** are stored in (Motorola) "Big Endian" byte order, not (Intel) "Little Endian" reverse byte order. The same is true for the **NumTracks** and **Format**.

Alternately, if the first byte of Division is negative, then this represents the division of a second that the time-stamps are based upon. The first byte will be -24, -25, -29, or -30, corresponding to the 4 SMPTE standards representing frames per second. The second byte (a positive number) is the resolution within a frame (ie, subframe). Typical values may be 4 (MIDI Time Code), 8, 10, 80 (SMPTE bit resolution), or 100.

You can specify millisecond-based timing by the data bytes of -25 and 40 subframes.

Here's what an MThd chunk looks like if you defined it in C:

```c
struct MTHD_CHUNK
{
   /* Here's the 8 byte header that all chunks must have */
   char           ID[4];  /* This will be 'M','T','h','d' */
   unsigned long  Length; /* This will be 6 */

   /* Here are the 6 bytes */
   unsigned short Format;
   unsigned short NumTracks;
   unsigned short Division;
};
```

And here's an example of a complete MThd chunk (with header) if you examined it in a hex editor:

```
4D 54 68 64       MThd ID
00 00 00 06       Length of the MThd chunk is always 6.
00 01             The Format type is 1.
00 02             There are 2 MTrk chunks in this file.
E7 28             Each increment of delta-time represents a millisecond.
```

After the MThd chunk, you should find an **MTrk chunk**, as this is the only other currently defined chunk. (If you find some other chunk ID, it must be proprietary to some other program, so skip it by ignoring the following data bytes indicated by the chunk's Length).

An MTrk chunk contains all of the midi data (with timing bytes), plus optional non-midi data for <u>one track</u>. Obviously, you should encounter as many MTrk chunks in the file as the MThd chunk's NumTracks field indicated.

The MTrk header begins with the ID of **MTrk**, followed by the Length (ie, number of data bytes for this track). The Length will likely be different for each track. (After all, a track containing the violin part for a Bach concerto will likely contain more data than a track containing a simple 2 bar drum beat).

Here's what an MTrk chunk looks like if you defined it in C:

```c
struct MTRK_CHUNK
{
   /* Here's the 8 byte header that all chunks must have */
   char            ID[4];    /* This will be 'M','T','r','k' */
   unsigned long   Length;   /* This will be the actual size of Data[] */

   /* Here are the data bytes */
   unsigned char   Data[];   /* Its actual size is Data[Length] */
};
```

Think of a track in the MIDI file in the same way that you normally think of a track in a sequencer. A sequencer track contains a series of **events**. For example, the first event in the track may be to sound a middle C note. The second event may be to sound the E above middle C. These two events may both happen at the same time. The third event may be to release the middle C note. This event may happen a few musical beats after the first two events (ie, the middle C note is held down for a few musical beats). Each event has a "time" when it must occur, and the events are arranged within a "chunk" of memory in the order that they occur.

In a MIDI file, an event's "time" precedes the data bytes that make up that event itself. In other words, the bytes that make up the event's time-stamp come first. A given event's time-stamp is referenced from the previous event. For example, if the first event occurs 4 clocks after the start of play, then its "delta-time" is 04. If the next event occurs simultaneously with that first event, its time is 00. So, a delta-time is the duration (in clocks) between an event and the preceding event.

NOTE: Since all tracks start with an assumed time of 0, the first event's delta-time is referenced from 0.

A delta-time is stored as a series of bytes which is called a **variable length quantity**. Only the first 7 bits of each byte is significant (right-justified; sort of like an ASCII byte). So, if you have a 32-bit delta-time, you have to unpack it into a series of 7-bit bytes (ie, as if you were going to transmit it over midi in a SYSEX message). Of course, you will have a variable number of bytes depending upon your delta-time. To indicate which is the last byte of the series, you leave bit #7 clear. In all of the preceding bytes, you set bit #7. So, if a delta-time is between 0-127, it can be represented as one byte. The largest delta-time allowed is 0FFFFFFF, which translates to 4 bytes variable length. Here are examples of delta-times as 32-bit values, and the variable length quantities that they translate to:

NUMBER	VARIABLE QUANTITY
00000000	00
00000040	40
0000007F	7F
00000080	81 00
00002000	C0 00
00003FFF	FF 7F
00004000	81 80 00
00100000	C0 80 00
001FFFFF	FF FF 7F

```
00200000          81 80 80 00
08000000          C0 80 80 00
0FFFFFFF          FF FF FF 7F
```

Here are some C routines to read and write variable length quantities such as delta-times. With **WriteVarLen()**, you pass a 32-bit value (ie, unsigned long) and it spits out the correct series of bytes to a file. **ReadVarLen()** reads a series of bytes from a file until it reaches the last byte of a variable length quantity, and returns a 32-bit value.

```c
void WriteVarLen(register unsigned long value)
{
   register unsigned long buffer;
   buffer = value & 0x7F;

   while ( (value >>= 7) )
   {
     buffer <<= 8;
     buffer |= ((value & 0x7F) | 0x80);
   }

   while (TRUE)
   {
      putc(buffer,outfile);
      if (buffer & 0x80)
          buffer >>= 8;
      else
          break;
   }
}


unsigned long ReadVarLen()
{
    register unsigned long value;
    register unsigned char c;

    if ( (value = getc(infile)) & 0x80 )
    {
       value &= 0x7F;
       do
       {
         value = (value << 7) + ((c = getc(infile)) & 0x7F);
       } while (c & 0x80);
    }

    return(value);
```

}

NOTE: The concept of variable length quantities (ie, breaking up a large value into a series of bytes) is used with other fields in a MIDI file besides delta-times, as you'll see later.

For those not writing in C, you may benefit from a psuedo-code explanation of the above routines. In pseudo-code, ReadVarLen() is:

1. Initialize the variable which will hold the value. Set it to 0. We'll call this variable 'result'.
2. Read the next byte of the Variable Length quantity from the MIDI file.
3. Shift all of the bits in 'result' 7 places to the left. (ie, Multiply 'result' by 128).
4. Logically OR 'result' with the byte that was read in, but first mask off bit #7 of the byte. (ie, AND the byte with hexadecimal 7F before you OR with 'result'. But make sure you save the original value of the byte for the test in the next step).
5. Test if bit #7 of the byte is set. (ie, Is the byte AND hexadecimal 80 equal to hexadecimal 80)? If so, loop back to step #2. Otherwise, you're done, and 'result' now has the appropriate value.

In pseudo code, WriteVarLen() could be:

1. Assume that you have a variable named 'result' which contains the value to write out as a Variable Length Quantity.
2. Declare an array which can contain 4 numbers. We'll call this variable 'array'. Initialize a variable named 'count' to 0.
3. Is 'result' less than 128? If so, skip to step #8.
4. Take the value 'result' AND with hexadecimal 7F, and OR with hexadecimal 80, and store it in 'count' element of 'array'. (ie, The first time through the loop, this gets stored in the first element of 'array'). NOTE: Don't alter the value of 'result' itself.
5. Increment 'count' by 1.
6. Shift all bits in 'result' 7 places to the right. (This can be done by dividing by 128).
7. Loop back to step #3.
8. Take the value 'result' AND with hexadecimal 7F, and store it in 'count' element of 'array'.
9. Increment 'count' by 1.
10. Write out the values stored in 'array'. Start with the last element stored above, and finish with the first element stored. (ie, Write them out in reverse order so that the first element of 'array' gets written to the MIDI file last). NOTE: The variable

'count' tells you how many total bytes to write. It also can be used as an index into the array (if you subtract one from it, and keep writing out bytes until it is -1).

An MTrk can contain MIDI events and non-MIDI events (ie, events that contain data such as tempo settings, track names, etc).

The first (1 to 4) byte(s) in an MTrk will be the first event's delta-time as a variable length quantity. The next data byte is actually the first byte of that event itself. I'll refer to this as the event's **Status**. For MIDI events, this will be the actual MIDI Status byte (or the first midi data byte if running status). For example, if the byte is hex 90, then this event is a **Note-On** upon midi channel 0. If for example, the byte was hex 23, you'd have to recall the previous event's status (ie, midi running status). Obviously, the first MIDI event in the MTrk <u>must</u> have a status byte. After a midi status byte comes its 1 or 2 data bytes (depending upon the status - some MIDI messages only have 1 subsequent data byte). After that you'll find the next event's delta time (as a variable quantity), etc.

SYSEX events

SYSEX (system exclusive) events (status = F0) are a special case because a SYSEX event can be any length. After the F0 status (which is always stored -- no running status here), you'll find yet another series of variable length bytes. Combine them with ReadVarLen() and you'll come up with a 32-bit value that tells you how many more bytes follow which make up this SYSEX event. This length doesn't include the F0 status.

For example, consider the following SYSEX MIDI message:

F0 7F 7F 04 01 7F 7F F7

This would be stored in a MIDI file as the following series of bytes (minus the delta-time bytes which would precede it):

F0 07 7F 7F 04 01 7F 7F F7

The 07 above is the variable length quantity (which happens to fit in just one byte for this example). It indicates that there are seven, following bytes that comprise this SYSEX message.

Really oddball midi units send a system exclusive message as a series of small "packets" (with a time delay inbetween transmission of each packet). The first packet begins with an F0, but it doesn't end with an F7. The subsequent packets don't start with

an F0 nor end with F7. The last packet doesn't start with an F0, but does end with the F7. So, between the first packet's opening F0 and the last packet's closing F7, there's 1 SYSEX message there. (Note: only extremely poor designs, such as the crap marketed by Casio exhibit such horrid behavior). Of course, since a delay is needed inbetween each packet, you need to store each packet as a separate event with its own time in the MTrk. Also, you need some way of knowing which events shouldn't begin with an F0 (ie, all of them except the first packet). So, the MIDI file redefines a midi status of F7 (normally used as an end mark for SYSEX packets) as a way to indicate an event that doesn't begin with F0. If such an event follows an F0 event, then it's assumed that the F7 event is the second "packet" of a series. In this context, it's referred to as a SYSEX CONTINUATION event. Just like the F0 type of event, it has a variable length followed by data bytes. On the other hand, the F7 event could be used to store MIDI REALTIME or MIDI COMMON messages. In this case, after the variable length bytes, you should expect to find a MIDI Status byte of F1, F2, F3, F6, F8, FA, FB, FC, or FE. (Note that you wouldn't find any such bytes inside of a SYSEX CONTINUATION event). When used in this manner, the F7 event is referred to as an ESCAPED event.

Non-MIDI events

A status of FF is reserved to indicate a special non-MIDI event. (Note that FF is used in MIDI to mean "reset", so it wouldn't be all that useful to store in a data file. Therefore, the MIDI file arbitrarily redefines the use of this status). After the FF status byte is another byte that tells you what **Type** of non-MIDI event it is. It's sort of like a second status byte. Then after this byte is another byte(s -- a variable length quantity again) that tells how many more data bytes follow in this event (ie, its Length). This Length doesn't include the FF, Type byte, nor the Length byte. These special, non-MIDI events are called **Meta-Events**, and most are optional unless otherwise noted. The section of this online book entitled "Meta-Events" lists the currently defined Meta-Events. Note that unless otherwise mentioned, more than one of these events can be placed in an MTrk (even the same Meta-Event) at any delta-time. (Just like all midi events, Meta-Events have a delta-time from the previous event regardless of what type of event that may be. So, you can freely intermix MIDI and Meta events).

Sequence Number

FF 00 02 *ss ss*

or...

FF 00 00

This optional event must occur at the beginning of a MTrk (ie, before any non-zero delta-times and before any midi events). It specifies the sequence number. The two data bytes *ss ss*, are that number which corresponds to the **MIDI Cue** message. In a format 2 MIDI file, this number identifies each "pattern" (ie, Mtrk) so that a "song" sequence can use the MIDI Cue message to refer to patterns.

If the *ss ss* numbers are omitted (ie, the second form shown above), then the MTrk's location in the file is used. (ie, The first MTrk chunk is sequence number 0. The second MTrk is sequence number 1. Etc).

In format 0 or 1, which contain only one "pattern" (even though format 1 contains several MTrks), this event is placed in only the first MTrk. So, a group of format 0 or 1 files with different sequence numbers can comprise a "song collection".

There can be only one of these events per MTrk chunk in a Format 2. There can be only one of these events in a Format 0 or 1, and it must be in the first MTrk.

Text

FF 01 *len text*

Any amount of text (amount of bytes = *len*) for any purpose. It's best to put this event at the beginning of an MTrk. Although this text could be used for any purpose, there are other text-based Meta-Events for such things as orchestration, lyrics, track name, etc. This event is primarily used to add "comments" to a MIDI file which a program would be expected to ignore when loading that file.

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

Copyright

FF 02 *len* *text*

A copyright message. It's best to put this event at the beginning of an MTrk.

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

Sequence/Track Name

FF 03 *len* *text*

The name of the sequence or track. It's best to put this event at the beginning of an MTrk.

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

Instrument

FF 04 *len* *text*

The name of the instrument (ie, MIDI module) being used to play the track. This may be different than the Sequence/Track Name. For example, maybe the name of your sequence (ie, Mtrk) is "Butterfly", but since the track is played upon a Roland S-770, you may also include an Instrument Name of "Roland S-770".

It's best to put one (or more) of this event at the beginning of an MTrk to provide the user with identification of what instrument(s) is playing the track. Usually, the instruments (ie, patches, tones, banks, etc) are setup on the audio devices via **MIDI Program Change** and **MIDI Bank Select Controller** events within the MTrk. So, this event exists merely to provide the user with visual feedback of what instruments are used for a track.

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

Lyric

FF 05 *len* *text*

A song lyric which occurs on a given beat. A single Lyric MetaEvent should contain only one syllable.

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

Marker

FF 06 *len* *text*

The text for a marker which occurs on a given beat. Marker events might be used to denote a loop start and loop end (ie, where the sequence loops back to a previous event).

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

Cue Point

FF 07 *len text*

The text for a cue point which occurs on a given beat. A Cue Point might be used to denote where a WAVE (ie, sampled sound) file starts playing, for example, where the *text* would be the WAVE's filename.

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

Program Name

FF 08 *len text*

The name of the program (ie, patch) used to play the MTrk. This may be different than the Sequence/Track Name. For example, maybe the name of your sequence (ie, Mtrk) is "Butterfly", but since the track is played upon an electric piano patch, you may also include a Program Name of "ELECTRIC PIANO".

Usually, the instruments (ie, patches, tones, banks, etc) are setup on the audio devices via **MIDI Program Change** and **MIDI Bank Select Controller** events within the MTrk. So, this event exists merely to provide the user with visual feedback of what particular patch is used for a track. But it can also give a hint to intelligent software if patch remapping needs to be done. For example, if the MIDI file was created on a non-General MIDI instrument, then the **MIDI Program Change** event will likely contain the wrong value when played on a General MIDI instrument. Intelligent software can use the Program Name event to look up the correct value for the **MIDI Program Change** event.

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

Device (Port) Name

FF 09 *len* *text*

The name of the MIDI device (port) where the track is routed. This replaces the "MIDI Port" Meta-Event which some sequencers formally used to route MIDI tracks to various MIDI ports (in order to support more than 16 MIDI channels).

For example, assume that you have a MIDI interface that has 4 MIDI output ports. They are listed as "MIDI Out 1", "MIDI Out 2", "MIDI Out 3", and "MIDI Out 4". If you wished a particular MTrk to use "MIDI Out 1" then you would put a Port Name Meta-event at the beginning of the MTrk, with "MIDI Out 1" as the *text*.

All MIDI events that occur in the MTrk, after a given Port Name event, will be routed to that port.

In a format 0 MIDI file, it would be permissible to have numerous Port Name events intermixed with MIDI events, so that the one MTrk could address numerous ports. But that would likely make the MIDI file much larger than it need be. The Port Name event is useful primarily in format 1 MIDI files, where each MTrk gets routed to one particular port.

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

End of Track

FF 2F 00

This event is <u>not</u> optional. It must be the last event in every MTrk. It's used as a definitive marking of the end of an MTrk. Only 1 per MTrk.

Tempo

FF 51 03 *tt tt tt*

Indicates a tempo change. The 3 data bytes of **tt tt tt** are the tempo in microseconds per quarter note. In other words, the microsecond tempo value tells you how long each one of your sequencer's "quarter notes" should be. For example, if you have the 3 bytes of 07 A1 20, then each quarter note should be 0x07A120 (or 500,000) microseconds long.

So, the MIDI file format expresses tempo as "the amount of time (ie, microseconds) per quarter note".

NOTE: If there are no tempo events in a MIDI file, then the tempo is assumed to be 120 BPM

In a format 0 file, the tempo changes are scattered throughout the one MTrk. In format 1, the very first MTrk should consist of only the tempo (and time signature) events so that it could be read by some device capable of generating a "tempo map". It is best not to place MIDI events in this MTrk. In format 2, each MTrk should begin with at least one initial tempo (and time signature) event.

See also: Tempo and Timebase.

SMPTE Offset

FF 54 05 *hr mn se fr ff*

Designates the SMPTE start time (hours, minutes, seconds, frames, subframes) of the MTrk. It should be at the start of the MTrk. The hour should not be encoded with the SMPTE format as it is in **MIDI Time Code**. In a format 1 file, the SMPTE OFFSET must be stored with the tempo map (ie, the first MTrk), and has no meaning in any other MTrk. The *ff* field contains fractional frames in 100ths of a frame, even in SMPTE based MTrks which specify a different frame subdivision for delta-times (ie, different from the subframe setting in the MThd).

Time Signature

FF 58 04 *nn dd cc bb*

Time signature is expressed as 4 numbers. *nn* and *dd* represent the "numerator" and "denominator" of the signature as notated on sheet music. The denominator is a negative power of 2: 2 = quarter note, 3 = eighth, etc.

The *cc* expresses the number of MIDI clocks in a metronome click.

The *bb* parameter expresses the number of notated 32nd notes in a MIDI quarter note (24 MIDI clocks). This event allows a program to relate what MIDI thinks of as a quarter, to something entirely different.

For example, 6/8 time with a metronome click every 3 eighth notes and 24 clocks per quarter note would be the following event:

FF 58 04 06 03 18 08

NOTE: If there are no time signature events in a MIDI file, then the time signature is assumed to be 4/4.

In a format 0 file, the time signatures changes are scattered throughout the one MTrk. In format 1, the very first MTrk should consist of only the time signature (and tempo) events so that it could be read by some device capable of generating a "tempo map". It is best not to place MIDI events in this MTrk. In format 2, each MTrk should begin with at least one initial time signature (and tempo) event.

Key Signature

FF 59 02 *sf mi*

sf = -7 for 7 flats, -1 for 1 flat, etc, 0 for key of c, 1 for 1 sharp, etc.

mi = 0 for major, 1 for minor

Proprietary Event

FF 7F *len data*

This can be used by a program to store proprietary data. The first byte(s) should be a unique ID of some sort so that a program can identity whether the event belongs to it, or to some other program. A 4 character (ie, ascii) ID is recommended for such.

Note that *len* could be a series of bytes since it is expressed as a variable length quantity.

The MIDI file format's Tempo Meta-Event expresses tempo as "the amount of time (ie, microseconds) per quarter note". For example, if a Tempo Meta-Event contains the 3 bytes of 07 A1 20, then each quarter note should be 0x07A120 (or 500,000) microseconds long.

BPM

Normally, musicians express tempo as "the amount of quarter notes in every minute (ie, time period)". This is the opposite of the way that the MIDI file format expresses it.

When musicians refer to a "beat" in terms of tempo, they are referring to a quarter note (ie, a quarter note is always 1 beat when talking about tempo, regardless of the time signature. Yes, it's a bit confusing to non-musicians that the time signature's "beat" may not be the same thing as the tempo's "beat" -- it won't be unless the time signature's beat also happens to be a quarter note. But that's the traditional definition of BPM tempo). To a musician, tempo is therefore always "how many quarter notes happen during every minute". Musicians refer to this measurement as BPM (ie, Beats Per Minute). So a tempo of 100 BPM means that a musician must be able to play 100 steady quarter notes, one right after the other, in one minute. That's how "fast" the "musical tempo" is at 100 BPM. It's very important that you understand the concept of how a musician expresses "musical tempo" (ie, BPM) in order to properly present tempo settings to a musician, and yet be able to relate it to how the MIDI file format expresses tempo.

To convert the Tempo Meta-Event's tempo (ie, the 3 bytes that specify the amount of microseconds per quarter note) to BPM:

BPM = 60,000,000/(tt tt tt)

For example, a tempo of 120 BPM = 07 A1 20 microseconds per quarter note.

So why does the MIDI file format use "time per quarter note" instead of "quarter notes per time" to specify its tempo? Well, its easier to specify more precise tempos with the former. With BPM, sometimes you have to deal with fractional tempos (for example, 100.3 BPM) if you want to allow a finer resolution to the tempo. Using microseconds to express tempo offers plenty of resolution.

Also, SMPTE is a time-based protocol (ie, it's based upon seconds, minutes, and hours, rather than a musical tempo). Therefore it's easier to relate the MIDI file's tempo to SMPTE timing if you express it as microseconds. Many musical devices now use

SMPTE to sync their playback.

PPQN Clock

A sequencer typically uses some internal hardware timer counting off steady time (ie, microseconds perhaps) to generate a software "PPQN clock" that counts off the timebase (Division) "ticks". In this way, the time upon which an event occurs can be expressed to the musician in terms of a musical bar:beat:PPQN-tick rather than how many microseconds from the start of the playback. Remember that musicians always think in terms of a beat, not the passage of seconds, minutes, etc.

As mentioned, the microsecond tempo value tells you how long each one of your sequencer's "quarter notes" should be. From here, you can figure out how long each one of your sequencer's PPQN clocks should be by dividing that microsecond value by your MIDI file's Division. For example, if your MIDI file's Division is 96 PPQN, then that means that each of your sequencer's PPQN clock ticks at the above tempo should be 500,000 / 96 (or 5,208.3) microseconds long (ie, there should be 5,208.3 microseconds inbetween each PPQN clock tick in order to yield a tempo of 120 BPM at 96 PPQN. And there should always be 96 of these clock ticks in each quarter note, 48 ticks in each eighth note, 24 ticks in each sixteenth, etc).

Note that you can have any timebase at any tempo. For example, you can have a 96 PPQN file playing at 100 BPM just as you can have a 192 PPQN file playing at 100 BPM. You can also have a 96 PPQN file playing at either 100 BPM or 120 BPM. Timebase and tempo are two entirely separate quantities. Of course, they both are needed when you setup your hardware timer (ie, when you set how many microseconds are in each PPQN tick). And of course, at slower tempos, your PPQN clock tick is going to be longer than at faster tempos.

MIDI Clock

MIDI clock bytes are sent over MIDI, in order to sync the playback of 2 devices (ie, one device is generating MIDI clocks at its current tempo which it internally counts off, and the other device is syncing its playback to the receipt of these bytes). Unlike with SMPTE frames, MIDI clock bytes are sent at a rate related to the musical tempo.

Since there are 24 MIDI Clocks in every quarter note, the length of a MIDI Clock (ie, time inbetween each MIDI Clock message) is the microsecond tempo divided by 24. In the above example, that would be 500,000/24, or 20,833.3 microseconds in every MIDI Clock. Alternately, you can relate this to your timebase (ie, PPQN clock). If you have 96 PPQN, then that means that a MIDI Clock byte must occur every 96 / 24 (ie, 4) PPQN clocks.

SMPTE

SMPTE counts off the passage of time in terms of seconds, minutes, and hours (ie, the way that non-musicians count time). It also breaks down the seconds into smaller units called "frames". The movie industry created SMPTE, and they adopted 4 different frame rates. You can divide a second into 24, 25, 29, or 30 frames. Later on, even finer resolution was needed by musical devices, and so each frame was broken down into "subframes".

So, SMPTE is not directly related to musical tempo. SMPTE time doesn't vary with "musical tempo".

Many devices use SMPTE to sync their playback. If you need to sychronize with such a device, then you may need to deal with SMPTE timing. Of course, you're probably still going to have to maintain some sort of PPQN clock, based upon the passing SMPTE subframes, so that the user can adjust the tempo of the playback in terms of BPM, and can consider the time of each event in terms of bar:beat:tick. But since SMPTE doesn't directly relate to musical tempo, you have to interpolate (ie, calculate) your PPQN clocks from the passing of subframes/frames/seconds/minutes/hours (just as we previously calculated the PPQN clock from a hardware timer counting off microseconds).

Let's take the easy example of 25 Frames and 40 SubFrames. As previously mentioned in the discussion of Division, this is analogous to millisecond based timing because you have 1,000 SMPTE subframes per second. (You have 25 frames per second. Each second is divided up into 40 subframes, and you therefore have 25 * 40 subframes per second. And remember that 1,000 milliseconds are also in every second). Every millisecond therefore means that another subframe has passed (and vice versa). Every time you count off 40 subframes, a SMPTE frame has passed (and vice versa). Etc.

Let's assume you desire 96 PPQN and a tempo of 500,000 microseconds. Considering

that with 25-40 Frame-SubFrame SMPTE timing 1 millisecond = 1 subframe (and remember that 1 millisecond = 1,000 microseconds), there should be 500,000 / 1,000 (ie, 500) subframes per quarter note. Since you have 96 PPQN in every quarter note, then every PPQN ends up being 500 / 96 subframes long, or 5.2083 milliseconds (ie, there's how we end up with that 5,208.3 microseconds PPQN clock tick just as we did above in discussing PPQN clock). And since 1 millisecond = 1 subframe, every PPQN clock tick also equals 5.2083 subframes at the above tempo and timebase.

Formulas

BPM = 60,000,000/MicroTempo
MicrosPerPPQN = MicroTempo/TimeBase
MicrosPerMIDIClock = MicroTempo/24
PPQNPerMIDIClock = TimeBase/24
MicrosPerSubFrame = 1000000 * Frames * SubFrames
SubFramesPerQuarterNote = MicroTempo/(Frames * SubFrames)
SubFramesPerPPQN = SubFramesPerQuarterNote/TimeBase
MicrosPerPPQN = SubFramesPerPPQN * Frames * SubFrames

To illustrate how best to use various Meta-Events when you save a MIDI file, let's consider an example sequencer track.

Assume that the user has created a sequencer track that he named "My track". He is using a Roland JV-1080 to play this track. His JV-1080 is connected to the third MIDI Out port on his computer's MIDI interface (which supports multiple MIDI Outs -- for more than 16 MIDI channels). The operating system lists this port with a name of "MIDI Out 3". He has picked out a patch on his JV-1080 called "Gonzo Harp". This is not a General MIDI patch. It happens to be in the third bank of patches, and is patch number 0. This track will play on the JV-1080's MIDI channel 1.

How would you write out an MTrk for this sequencer track?

First, you would start off with a Sequence/Track Name Meta-Event. The text for this event would be "My track". (Note that you do not need to nul-terminate text in a Meta-Event. The Meta-Event's variable quantity length tells how many characters are in the name). You would put this at a delta-time of zero. So let's examine our MTrk chunk so far, as if we were using a hex editor:

```
4D 54 72 6B        MTrk ID
00 00 00 0C        Length of the MTrk chunk so far.

                   Our Track Name Meta-Event
00                 Delta-time is 0.
FF                 A Meta-Event.
03                 Track Name type.
08                 Length of "My track"
4D                 ASCII 'M'
79                 ASCII 'y'
20                 ASCII ' '
74                 ASCII 't'
72                 ASCII 'r'
61                 ASCII 'a'
63                 ASCII 'c'
6B                 ASCII 'k'
```

Next, you may choose to put an Instrument Name Meta-Event (although this isn't as important as other events, and you may skip it if your software neither knows, nor cares, what specific MIDI module is playing the track). The text for this event would be "Roland JV-1080".

```
                      Our Instrument Name Meta-Event
00                    Delta-time is 0.
FF                    A Meta-Event.
04                    Instrument Name type.
0E                    Length of "Roland JV-1080"
52                    ASCII 'R'
6F                    ASCII 'o'
6C                    ASCII 'l'
61                    ASCII 'a'
6E                    ASCII 'n'
64                    ASCII 'd'
20                    ASCII ' '
4A                    ASCII 'J'
56                    ASCII 'V'
2D                    ASCII '-'
31                    ASCII '1'
30                    ASCII '0'
38                    ASCII '8'
30                    ASCII '0'
```

Next, you would put a Device (Port) Name Meta-Event. The text for this event would be "MIDI Out 3".

```
                      Our Device (Port) Name Meta-Event
00                    Delta-time is 0.
FF                    A Meta-Event.
09                    Device (Port) Name type.
0A                    Length of "MIDI Out 3"
4D                    ASCII 'M'
49                    ASCII 'I'
44                    ASCII 'D'
49                    ASCII 'I'
20                    ASCII ' '
4F                    ASCII 'O'
75                    ASCII 'u'
74                    ASCII 't'
20                    ASCII ' '
33                    ASCII '3'
```

Next, you would put an Program (Patch) Name Meta-Event to indicate the name of the patch being used. The text for this event would be "Gonzo Harp".

```
                    Our Program (Patch) Name Meta-Event
00                  Delta-time is 0.
FF                  A Meta-Event.
08                  Program (Patch) Name type.
0A                  Length of "Gonzo Harp"
47                  ASCII 'G'
6F                  ASCII 'o'
6E                  ASCII 'n'
7A                  ASCII 'z'
6F                  ASCII 'o'
20                  ASCII ' '
48                  ASCII 'H'
61                  ASCII 'a'
72                  ASCII 'r'
70                  ASCII 'p'
```

Next, you need to put the MIDI Bank Select Controller events and the MIDI Program Change event that select the desired patch ("Gonzo Harp") on the JV-1080. Let's assume that the Bank Select values will be 0 for MSB controller and 3 for LSB. We already said that the patch number is 0.

```
                    Bank Select MSB
00                  Delta-time is 0.
B0                  Controller on channel 1.
00                  Bank Select MSB.
00                  MSB = 0
                    Bank Select LSB
00                  Delta-time is 0.
B0                  Controller on channel 1.
20                  Bank Select LSB.
03                  LSB = 3
                    Program Change
00                  Delta-time is 0.
C0                  Program Change on channel 1.
00                  Patch = 0
```

After this, you would put other data for the track, such as MIDI Note-on, Note-off, etc,

and other Meta-Events.

Now let's examine loading that same file created above.

First, you'll encounter the Track Name event. Use this to set the name for the sequencer track which will store this MTrk's data.

Next, you'll encounter the Instrument Name. If your program keeps a database of instruments that are "installed" on the computer, this could come in useful for several reasons. First, if your database lists which instruments are attached to which MIDI ports on the system, then you could ignore any Device (Port) Name event (just in case it happens to refer to someone else's MIDI port that isn't applicable on your system). You'll know which MIDI port to use for this MTrk based solely upon the Instrument Name and your own database. Secondly, if you have that instrument listed in your database, then you choose to forego any patch remapping. (ie, You may choose not to check that the values in any Bank Select and Program Change messages are correct, under the assumption that the desired patch is already located where it should be). At least, it may make any patch remapping easier.

Next, you'll encounter the Device (Port) Name event. If you weren't able to use the Instrument Name to deduce which MIDI port to use on your system, then you should check this port name. If it exists on your system, use that port. If not, use a default port.

Next, you'll encounter the Program Name event. You can use this to implement patch remapping. If your instrument database also lists what patches are available upon each instrument, then you can check for the existence of this patch in your database. (If you were able to use the Instrument Name, then you already know upon what instrument that patch should be found). Once you find that patch in your database, hopefully your database will contain the correct Bank Select and Program Change values needed to select that patch. In that case, you could ignore the next Bank Select and Program Change events you encounter in the MTrk. What you've effectively done is remap a patch change. So for example, if the MTrk was created upon an instrument with a unique arrangement of patches (and therefore its Bank Select and Program Change events have the wrong values when played upon a different instrument), you've remapped it to play the correct patch on another instrument.

Finally, you'll encounter the Bank Select and Program Change events. If you weren't able to use the Instrument and Program Name Meta-Events to deduce the correct Bank

Select and Program Change values, then you may use these events verbatim.

A free Dynamic Link Library called "GenMidi" is available to programmers writing MIDI software. This DLL can be used to maintain a database of Instrument names, and the names of all patches on those instruments (including which Bank Select and Program Change values select each patch). Included with the DLL is a tool that allows a user to create a database of such.

By using this DLL in your program, you can more easily implement patch remapping, and better coordinate with other programs that also use this DLL. Plus, you don't have to develop your own tools to allow users to create such databases.

The DLL is downloadable from the Software Programs page.

The method of saving data in chunks (ie, where the data is preceded by an 8 byte header consisting of a 4 char ID and a 32-bit size field) is the basis for Interchange File Format. You should now read the article About Interchange File Format for background information.

As mentioned, MIDI File format is a "broken" IFF. It lacks a file header at the start of the file. One bad thing about this is that a standard IFF parsing routine will choke on a MIDI file (because it will expect the first 12 bytes to be the group ID, filesize, and type ID fields). In order to fix the MIDI File format so that it strictly adheres to IFF, Microsoft simply made up a 12-byte header that is prepended to MIDI files, and thereby came up with the RMID format. An RMID file begins with the group ID (4 ascii chars) of 'R', 'I', 'F', 'F', followed by the 32-bit filesize field, and then the type ID of 'R', 'M', 'I', 'D'. Then, the chunks of a MIDI file follow (ie, the MThd and MTrk chunks). So, somewhere after the first 12 bytes of an RMID file, then you should find an embedded MIDI file (although there may be other "chunks" of data before it. Simply skip over those as you would any unknown chunk).

Note that chunks within a MIDI file are not padded out (with an extra 0 byte) to an even number of bytes. I don't know as if the RMID format corrects this aberration of the MIDI file format too.

The following Meta-Events are considered obsolete and should not be used. (The MMA would like you to know that they never endorsed their use, although since certain versions of CakeWalk utilized them, you may find existing MIDI files with these events). Use the Device (Port) Name Meta-Event instead of the MIDI Port Meta-Event.

MIDI Channel

FF 20 01 *cc*

This optional event which normally occurs at the beginning of an MTrk (ie, before any non-zero delta-times and before any MetaEvents except Sequence Number) specifies to which MIDI Channel any subsequent MetaEvent or System Exclusive events are associated. The data byte **cc**, is the MIDI channel, where 0 would be the first channel.

The MIDI spec does not give a MIDI channel to System Exclusive events. Nor do MetaEvents have an imbedded channel. When creating a Format 0 MIDI file, all of the System Exclusive and MetaEvents go into one track, so its hard to associate these events with respective MIDI Voice messages. (ie, For example, if you wanted to name the musical part on MIDI channel 1 "Flute Solo", and the part on MIDI Channel 2 "Trumpet Solo", you'd need to use 2 Track Name MetaEvents. Since both events would be in the one track of a Format 0 file, in order to distinguish which track name was associated with which MIDI channel, you would place a MIDI Channel MetaEvent with a channel number of 0 before the "Flute Solo" Track Name MetaEvent, and then place another MIDI Channel MetaEvent with a channel number of 1 before the "Trumpet Solo" Track Name MetaEvent.

It is acceptable to have more than one MIDI channel event in a given track, if that track needs to associate various events with various channels.

MIDI Port

FF 21 01 *pp*

This optional event which normally occurs at the beginning of an MTrk (ie, before any non-zero delta-times and before any midi events) specifies out of which MIDI Port (ie, buss) the MIDI events in the MTrk go. The data byte *pp*, is the port number, where 0 would be the first MIDI buss in the system.

The MIDI spec has a limit of 16 MIDI channels per MIDI input/output (ie, port, buss, jack, or whatever terminology you use to describe the hardware for a single MIDI input/

output). The MIDI channel number for a given event is encoded into the lowest 4 bits of the event's Status byte. Therefore, the channel number is always 0 to 15. Many MIDI interfaces have multiple MIDI input/output busses in order to work around limitations in the MIDI bandwidth (ie, allow the MIDI data to be sent/received more efficiently to/ from several external modules), and to give the musician more than 16 MIDI Channels. Also, some sequencers support more than one MIDI interface used for simultaneous input/output. Unfortunately, there is no way to encode more than 16 MIDI channels into a MIDI status byte, so a method was needed to identify events that would be output on, for example, channel 1 of the second MIDI port versus channel 1 of the first MIDI port. This MetaEvent allows a sequencer to identify which MTrk events get sent out of which MIDI port. The MIDI events following a MIDI Port MetaEvent get sent out that specified port.

It is acceptable to have more than one Port event in a given track, if that track needs to output to another port at some point in the track.

Introduction

A sequencer is a machine that "plays" musical performances. It tells equipment that can make musical sounds (ie, play pitches, chords, or make any kind of noise) <u>what</u> musical notes to play, and <u>when</u> to play them. **It does this using MIDI messages.**

Recording a musical performance using MIDI

Most sequencers allow a musician to record his performance into the sequencer just as if he were recording his performance upon a cassette tape. <u>But</u>, there is no tape involved, and the sequencer records MIDI messages instead of the actual sounds. (The messages are usually stored in RAM during recording, and later saved to a floppy disk or hard drive for permanent storage. Most sequencers today save the MIDI messages in MIDI File Format data files). In other words, the sequencer electronically records all of your finger, foot, what-have-you movements. It does this by storing all of the MIDI messages which the electronic musical instrument generates when you physically play that instrument.

The sequencer also keeps track of a "musical beat" during the recording so that it can record the "rhythm" of your movements (ie, the rhythms of your playing). Usually, the sequencer generates a metronome sound which the musician follows during the recording phase in order to have the sequencer accurately capture his rhythms while it is storing the generated MIDI messages.

Playing a musical performance using MIDI

Then, upon playback, the sequencer plays the instrument (that you just did) by electronically recreating all of your gestures. The sequencer does this by sending back (to that instrument's MIDI IN jack) those MIDI messages that the musical instrument generated when you played it. The sequencer doesn't have fingers, so it may send a MIDI message that tells an electronic keyboard to pretend that a human has just pressed middle C for example. The keyboard then sounds middle C.

Sequencers can send MIDI messages that tell the keyboard to release a key that it played (ie, remove the finger from middle C), move the pitch wheel, press down a sustain pedal, turn the volume knob, etc. In other words, a sequencer can fully duplicate a musician's performance. (Note that the keyboard only pretends to do these things. That C key doesn't physically depress as if some ghost is sitting at the keyboard. It does make the appropriate musical pitch though, so it <u>sounds</u> as if there is a ghost playing the keyboard).

Advantages of sequencers

A sequencer usually has a tempo control so that it can play the performance faster or slower. Unlike with digital audio playback, when you change the tempo of the MIDI sequencer's playback, it doesn't alter the pitch nor the timbre of the sound. (One exception to this general rule is the use of time-stretching to change the tempo of a digital audio recording of a musical performance). It alters only the tempo. So, a sequencer offers complete control of tempo independent of pitch or timbre changes. So too, this means that you can record a part at a slower tempo, and then speed up the playback without any change in timbre or pitch.

MIDI sequencers also offer the facilities to easily transpose musical parts (to other key signatures), mute or solo individual parts, reroute musical parts to play upon various sound modules, and make many other changes in timbre/volume/panning/effects levels/etc to individual musical parts. These changes can all be (theoretically) done in real-time (ie, during playback) without time-consuming calculations made upon the data, unlike with digital audio playback.

The sequencer may have many other features to easily edit the performance (even upon a per note basis) so that the sequencer can correct the mistakes that you recorded into it. It's infinitely easier than trying to correct mistakes recorded onto magnetic tape, or even digital audio recorders, which is why musicians use sequencers so much nowadays.

Most sequencers offer the facility to enter notes (and other control data such as volume/pan/vibrato/ etc settings) by "step-entering" the data. You don't have to play the musical part in real-time. Instead, you can slowly enter each musical note (and other setting), and specify what musical bar and beat upon which it gets "played". This is a boon to people whose musical technique is not sufficiently developed to physically play the musical parts in real-time. Some sequencers even offer very intuitive interfaces, such as allowing you to step-enter notes by clicking the mouse pointer upon a graphical sheet of music manuscript drawn upon the screen. Digital audio recorders do not typically have such features. With digital audio recorders, you pretty much have to be able to play some musical parts.
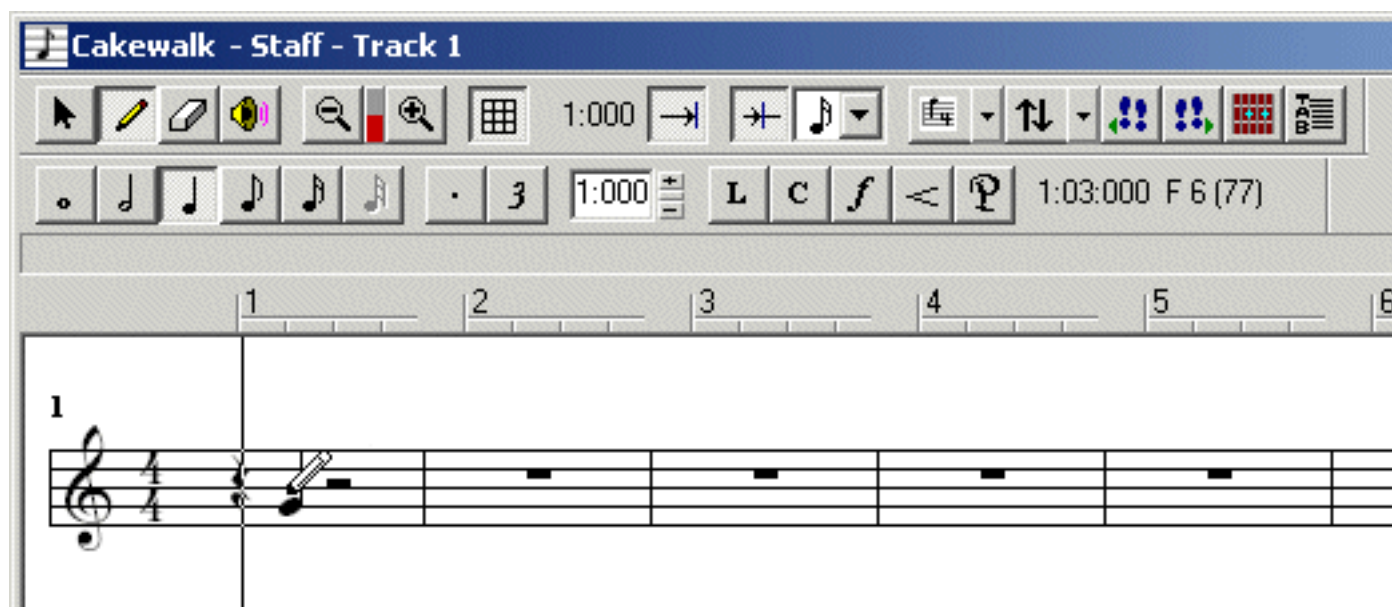
An example of step entry upon Cakewalk's staff view. I moved the pencil over the staff where a G note would occur upon the second beat, and clicked once. This inserted a G note. Because I have the quarter note length button depressed, a quarter note is inserted upon that beat.

A specialized breed of sequencers called "Algorithmic Composition programs" even have the capacity to create a complete musical arrangement on your behalf. See Algorithmic Composition for more details. Digital audio recorders do not offer such a feature.

Computers as sequencers

Most computers have software programs which turn the computer into a sequencer. With a sound card installed inside of the computer, the sequencer can playback musical performances without even needing external MIDI sound modules (since most sound cards now have an internal, multitimbral General MIDI module, usually a wavetable synth, that can recognize and properly "play" the MIDI messages that the sequencer outputs to the sound card's driver).

Examples of sequencer software are CakeWalk, Cubase, Logic, Mark of the Unicorn, etc. There are many such programs, as sequencers are perhaps the most often used tools in music production.

An example of a very simple sequencer is Windows MultiMedia Player. This software does not have any facility to record MIDI messages, nor edit them. It can only playback a MIDI performance (stored in MIDI File Format). Such **simple "playback only" sequencers are often called "MIDI Players".**

Support for digital audio playback

Because MIDI controls only electronic instruments, and can't record the sound of acoustic instruments or human voice, digital audio recording/playback is a feature that is often added to sequencers. The sequencer is designed to record/playback tracks of digital audio in sync with the MIDI tracks. See Digital Audio on a computer for more information about digital audio recording/playback.

Many companies have released MIDI interfaces that feature multiple output busses (MIDI OUTS). Most of these models connect to the parallel or USB port. For examples of such models, consult Audio Cards and MIDI Interfaces for a computer.

Note: Do not be confused by MIDI interfaces that have more than 1 MIDI OUT connector but do not have multiple busses. These connectors all tap into one midi bus. What this means is that, at any given time, all of the outputs are pumping out the exact same MIDI data. This is really only a matter of convenience because it allows you to hook up several MIDI devices directly to the MIDI interface as opposed to daisy-chaining these devices to themselves (via the MIDI thru of each device). Big deal. You still have only 16 midi channels, and all of the sequencer data must go out this 1 bus.

What do these multiple busses offer?

Let's assume that we have a device with 3 independent busses (and therefore at least 3 MIDI OUT connectors). At any one time, each of these connectors can be outputting completely different MIDI data. So, you have a total of 48 midi channels (16 per jack). The big advantage is that your sequencer can split up the outgoing midi data for various MIDI devices among those 3 connectors. It doesn't have to pump all midi data out of 1 bus.

What's the significance of this?

Midi is a serial transmission. What this means is that only 1 piece of information can be sent through a single midi cord (or bus) at a time, and it takes a relatively long time to send that piece of info. (It seems like a blindingly fast time to a human, but to machines like computers and MIDI devices, it's quite slow.) For example, it takes a sequencer (at least) 960 microseconds to send a midi note-on to a midi device (ie, play a single note on a midi device). Now, to our ears that sounds instantaneous. Well, what if, on a particular downbeat of a musical piece, our sequencer has to send a 5 note chord to a MIDI piano module, a kick and hihat strike to a drum box, and a bass note to some MIDI bass module. That's 8 midi note-ons that have to be sent out on the midi bus. All of these notes are supposed to sound together; 5 piano notes, 1 kick, 1 hihat, and 1 bass note. But, since midi can only send one note at a time on a single bus, it has to send each of those as fast at it can, one at a time. Well, it takes 960 microseconds to send each note, so by the time that the 5 piano notes are sent, it's already 4800 microseconds later than the downbeat. By the time that the bass note gets sent, it's 7680 microseconds later than the downbeat when it was supposed to sound. We've lost the illusion of "instantaneous" because now our ear can start to hear that the bass note sounds "behind" the first piano note instead of "with it". Quite simply, we tried to pump too much data out of 1 midi bus all at the same instance (ie, the same downbeat). This problem is known as "Midi Data Clog" or "Midi Delays". Technically, we can say that midi doesn't have enough **bandwidth** to perform what we want well enough to satisfy our ears.

What if we had 3 independent MIDI busses (outputs). Well, we can hook the piano module to one output and tell the sequencer to send only the 5 piano notes to that output. We can connect the drum box to another output and tell the sequencer to send the kick and hihat notes through that output. We can connect the bass module and tell the sequencer to send only the bass note through that output.

Now, the sequencer doesn't have to send all of the data over a single bus. The sequencer can send data out those 3 busses simultaneously. In other words, the bass note can be output while the piano notes are being output. As a result, that bass note really does sound at the same time that the first piano note does. (It's true that those 5 piano notes are still sent out one at a time. After all, they're going out of one midi bus, so the 5th note is still 4800 microseconds behind the 1st. But, at least the bass note doesn't have to wait until all of those 5 piano notes are sent before it gets sent. Neither do the kick and hihat notes have to wait because they have their own bus, too. So, things end up sounding much more instantaneous, even if they aren't technically, and the illusion is all that matters to a human).

Of course, a program would have to be specifically written to access the 3 busses of the interface, so obviously you need software (ie, drivers) that utilize a particular devices' busses. If you attempted to use a program not written to control this device, at best, only one of the outputs would work. In other words, it would be no better than an ordinary interface. Most professional sequencers offer support for multiple busses, allowing you to set an individual track to output of any one of the numerous MIDI OUT connectors.
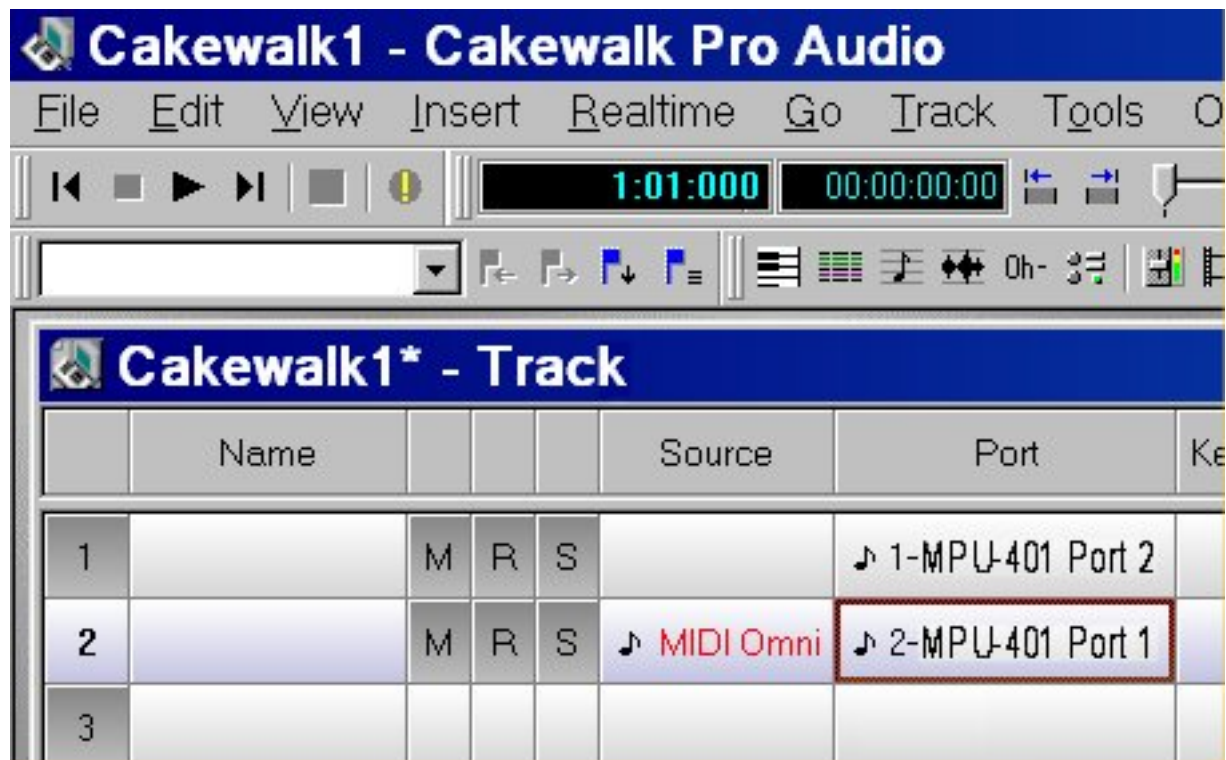
Assume that you have a MIDI interface that has two independent busses (ie, 32 MIDI channels). The first bus happens to be named "MPU-401 Port 1" and the second bus is named "MPU-401 Port 2". (These names are determined by the manufacturer of the interface, and set when you install the interface's driver). Above, we have assigned the first track of Cakewalk to output of the second port, and the second track to output of the first port. This means that even if we use MIDI channel 1 for both tracks, they will actually be different channels. After all, track one is using MIDI channel 1 on port 2. Track two is using MIDI channel 1 on port 1, which is an entirely different channel.

If you're using software that can utilize the Windows MIDI Mapper (ie, any Windows program that uses MCI), then you've already got the ability to adapt your software to use multiple busses. Simply

install more than one MIDI interface card (or a card with multiple busses), and route various MIDI channels of the MIDI Mapper to different drivers (ie, different outputs). You end up chopping up 16 channels into more than one output (ie, you don't take advantage of the fact that you could now have 16 MIDI channels per each device), but you get more efficient output for those 16 channels. In Windows 95, you open the Control Panel's MultiMedia notebook to the "MIDI" page, and use the "Advanced Configuration" to route MIDI channels to different driver/devices. See MIDI connections and computer setup for details of setting up multi-port cards under Windows 95.

Here are the numbers for all defined MIDI controllers. The first 32 numbers (0 to 31) are coarse adjustments for various parameters. The numbers from 32 to 63 are their respective fine adjustments (ie, add 32 to the coarse adjust number to get the fine adjust number, except for the General Purpose Sliders which have no fine adjust equivalents). For example, the coarse adjustment for Channel Volume is controller number 7. The fine adjustment for Channel Volume is controller number 39 (7 +32). Many devices use only the coarse adjustments, and ignore the fine adjustments.

See Registered and Non-Registered Parameter Number for more detailed information about those controllers.

For more detailed, technical information about MIDI controllers, see MIDI controllers.

Click upon any individual controller name for detailed, technical information about that controller (ie, what it does, what its allowable data range is, etc).

0   **Bank Select (coarse)**
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**
95	**Phaser Level**
96	**Data Button increment**
97	**Data Button decrement**
98	**Non-registered Parameter (fine)**
99	**Non-registered Parameter (coarse)**
100	**Registered Parameter (fine)**
101	**Registered Parameter (coarse)**
120	**All Sound Off**
121	**All Controllers Off**
122	**Local Keyboard (on/off)**
123	**All Notes Off**
124	**Omni Mode Off**
125	**Omni Mode On**
126	**Mono Operation**
127	**Poly Operation**

The following software is Windows 95/98/NT/2000/XP stuff that I've written. Some of these utilities require Microsoft's Visual C 4.0 runtime library which is a file named MSVCRT40.DLL. You very likely already have this file in either your Windows or System directory, but if you need to obtain it, it's freely available, and the download is provided below (under "Miscellaneous").

For most of the following programs, you'll be downloading one file. This file is the install program. When you run this install program, it will allow you to choose some options, and then it will install the final set of files upon your computer. After installation, you can delete this (install) program that you've downloaded (or keep it somewhere in case you have to reinstall the software later).

MIDI Utilities

MIDI File Disassembler/Assembler

A program to convert MIDI files into readable text files that you can view and edit with any text editor. This also goes the other way, converting the text file back into a new MIDI file with any changes you made. But it's more than that. It features much more processing of the MIDI data, allowing you to filter out any particular types of data, or filter by MIDI channel or note range, or remap note numbers, or scale volume, or delay tracks, etc. In order to use this program, you must first download and install the MidiFile DLL and GenMidi DLL.

MIDI Patch Lister

A program that allows you to create lists of all of the patches upon all of your MIDI instruments. These lists are then referenced by programs such as the MIDI File Disassembler. In order to use this program, you must first download and install the GenMidi DLL.

MIDI File Converter

A program to convert MIDI File Format files between Types 0, 1, or 2. See About MIDI files for a more indepth explanation of the 3 Types. In order to use this program, you must first download and install the MidiFile DLL.

MIDI Input Viewer

A program that displays the MIDI messages at your computer's MIDI In. (It also can be used as a "software MIDI Thru" utility. Useful for checking the output of MIDI controllers, or verifying a sound module's response. In order to use this program, you

must first download and install the GenMidi DLL.

MIDI Device Info Tool

A program that lists information about all of the MIDI devices in your system. Useful for checking device driver versions, and device capabilities.

WAVE (digital audio) utilities

Aiff To Wave

A program to convert Audio Interchange File Format (ie, AIFF -- a digital audio file format often used on the Mac) to WAVE file format (ie, the format mostly used on Intel-based computers).

Wave To Aiff

A program to convert WAVE file format to Audio Interchange File Format (ie, AIFF). This is the complementary utility to the Aiff to Wave utility above.

Patch Editors

Roland A-33 Patch Editor

A program to edit the parameters/setup of a Roland A-33 keyboard controller. Now you can setup the A-33 without needing to blindly poke at buttons upon its front panel. In order to use this program, you must first download and install the MidiFile DLL and GenMidi DLL.

Roland A-30 Patch Editor

A program to edit the parameters/setup of a Roland A-30 keyboard controller. Now you can setup the A-30 without needing to blindly poke at buttons upon its front panel. In order to use this program, you must first download and install the MidiFile DLL and GenMidi DLL.

Rap10 Playback Panel

A Windows 95/98/ME (not NT/2000/XP) program to control some aspects of the Roland RAP-10 audio card's playback. This can control the GM module's master volume, tuning, and transpose, as well as the muting of various MIDI channels. It also controls digital audio tracks' volume, tuning, transpose, chorus and reverb levels, and brightness. It controls reverb/delay and chorus units as well. (NOTE: For using the RAP-10's built-in sound module under NT/2000/XP, use an MPU-401 driver, and my SC-7 Playback Panel with that driver's MIDI Out).

RAP-10 Recording Panel

A Windows 95/98/ME (not NT/2000/XP) program to control some aspects of the Roland RAP-10 audio card's recording. This can control the microphone and aux input levels, turn the monitor on or off, and set bit resolution and sample rate.

SC-7 Playback Panel

A program to control some aspects of the Roland SC-7 MIDI module's playback. This can control the GM module's master volume, tuning, and transpose, as well as the muting of various MIDI channels. It controls reverb/delay and chorus units as well.

Miscellaneous

IDF Editor

Microsoft's program to create an Instrument Definition File (IDF) for Windows 95/98/ME. This is essentially Win95's version of the Patch and Drum Note remapping of Win3.1's MIDI Manager. An IDF allows you to remap the General MIDI Patch set and Drum Note assignments for a MIDI sound source that is not GM compliant. In this way, software that is designed to select patches and play drums upon a GM sound source will play the proper instrumentation and drum sounds upon a non-GM unit (ie, via the IDF you create for that unit). This is a zip archive, and you should use a utility such as WinZip to unzip it.

Microsoft Visual C runtime DLL

MSVCRT40.DLL is needed by most of my Windows utilities. Download this only if you don't already have that file in your Windows or System directory. This file is zipped, and you should use a utility such as WinZip to unzip it.

Developer's software

The following software is mostly of interest only to other developers:

These packages are archived in the ZIP format, and you should use the command to recreate directories when unzipping. Use a utility that preserves Windows Long File Names.

winmm.zip

> Microsoft's online book that gives more detailed information about Windows MIDI and digital audio API. For programmers. This is a bit older information, and doesn't include some of the newer functions. It also doesn't give nearly as much depth as my [Windows MIDI and Digital Audio Programming tutorial](#).

[GenMidi DLL](#)

> A Win32 dynamic link library that you can use with your applications to add "Instrument Definition" support (ie, whereby the user can maintain a database of Bank and Patch names, and select among those, rather than needing to deal with Bank Controller and Program Change numbers). Think of the "Instrument Definition" portion of CakeWalk, and you'll get an idea what I'm talking about. Examples for using the DLL in C are included. Be sure that when you unzip this archive, you restore the directory structure saved within the archive. You also need an unzip utility that preserves Windows Long File Names. Also available is the [MIDI Patch Lister](#) utility for creating databases. Freely redistributable with your application.

[MidiFile DLL](#)

> A Win32 dynamic link library that you can use with your applications to simplify reading and writing MIDI File Format. Examples for reading and writing in both C and C++ are included. Freely redistributable with your application. Be sure that when you unzip this archive, you restore the directory structure saved within the archive.

Electronic Arts is a company that deserves credit for helping make life easier for both programmers and end users. By establishing **Interchange Format Files** (ie, IFF) and releasing the documentation for such, as well as C source code for reading and writing IFF type of files, Electronic Arts has helped make it easier for programmers to develop "backward compatible" and "extensible" file formats. IFF also helps developers write programs that easily read data files created with each others' IFF compliant software, even if there is no business relationship between the developers. In a nutshell, IFF helps minimize problems such as new versions of a particular program having trouble reading data files produced by older versions, or needing a new file format everytime a new version needs to store additional information. It also encourages standardized file formats that aren't tied to a particular product. All of this is good for endusers because it means that their valuable data isn't locked into some proprietary standard that can't be used with a wide variety of hardware and software. Above all else, endusers don't want their work to be held hostage by a single, corporate entity over whom the enduser has no direct control, but that's exactly what happens whenever an enduser saves his data using a program that produces a proprietary, unpublished file format. IFF helps to break this needlessly proprietary stranglehold that developers have exerted upon endusers' works.

An IFF file is a set of data that is in a form that many, unrelated programs can read. An IFF file should not have anything in it that was intended specifically for just one, particular program. If a program must save some "personal" (ie, proprietary) data in an IFF file, it must be saved in a manner which allows another program to "skip over" this data. There are several different types of IFF files. ILBM and GIFF files store picture data. SMUS files store musical scores. WAVE and AIFF files store sampled sounds. Each of these files must start with an ID which indicates that it is indeed an IFF file, followed by an ID that indicates which type of file. So what is an ID? An ID is four, printable ascii characters (ie, 8-bit bytes). If you use a file viewer (capable of displaying each byte as an ascii character) to look at an IFF file, you will notice that every so often you will see 4 "readable" characters in a row. These 4 characters are an ID. Every IFF file must start with one of the following 3 IDs. (I've enclosed each ID in single quotes).

```
'FORM'
```

```
'LIST'
```

```
'CAT '
```

If the first 4 chars (bytes) in a file are not one of these, then it is not an IFF file. These IDs are referred to as **group ID**s in EA literature because each is like a "master ID" after which there may follow more IDs (ie, chunks) that are grouped under that master ID.

Note that the last character in the 'CAT ' ID is a blank space (ie, ascii 32).

After this group ID, there is an UNSIGNED LONG (ie, 32-bit binary value) that indicates how many bytes are in the entire file. This count does not include the 4 byte group ID, nor this ULONG. This ULONG is useful if you wish to load the rest of the file into memory to examine it. After this ULONG, there is an ID that indicates which type of IFF file this is. As mentioned earlier, "ILBM", "WAVE", and "AIFF" are 3 types of IFF files. There are many more, and programmers are always

inventing new types for lack of better things to do. Here is the beginning of a typical ILBM file.

'FORM'	OK. This really is an IFF file because it has one of the 3 defined group IDs.
13000	There are 13000 more bytes after this ULONG.
'ILBM'	It is an ILBM (picture) file.

All IFF files start with something similiar to the above, 12 byte "header", except that instead of 'FORM', the group ID can be 'LIST' or 'CAT '. Of course, the ULONG size and file type ID may be different in various files, but nevertheless, a 12 byte header always appears at the beginning of an IFF file. For example, here's an example AIFF header:

'FORM'	OK. This really is an IFF file because it has one of the 3 defined group IDs.
4000	There are 4000 more bytes after this ULONG.
'AIFF'	It is an AIFF (digital audio) file.

What you find after the header depends on which type it is (ie, From here on, an ILBM will be different than an AIFF).

One thing that all IFF files do have in common after the group ID, byte count, and type ID, is that data is organized into chunks. OK, more jargon. What's a chunk? A chunk consists of an ID, a ULONG that tells how many bytes of data are in the chunk, and then all those data bytes. For example, here is a CMAP chunk (which would be found in an ILBM file).

'CMAP'	This is the 4 byte chunk ID.
6	This tells how many data bytes are in the chunk (ie, This is the chunkSize).
0,0,0,1,1,4	Here are the 6 data bytes.

Notice that the chunk size doesn't include the 4 byte ID or the ULONG for the chunk Size.

So, all IFF files are made up of several chunks (ie, groups of data). Each group of data starts with a convenient ID (so that a program can ascertain what kind of data is in the chunk) and a ULONG size (so that a program can ascertain how many bytes of data are in the chunk). There are a few other details to note. A chunk cannot have an odd number of data bytes (such as 3). If necessary, an extra zero byte must be written to make an even number of data bytes. The chunk Size doesn't include this extra byte. So for example, if you want to write 3 bytes in a CMAP chunk, it would look like this:

'CMAP'	
3	Note that chunk Size is 3.
0,1,33,0	Note that there is an extra zero byte.

The reason for this extra "pad byte" for odd-sized chunks has to do with Motorola's 68000 CPU

requiring that LONGs be aligned to even memory addresses. IFF files were first used on 68000 based computers, and padding out odd-sized chunks made it easier to load and parse an IFF file on such a computer (ie, if you load the entire file into a single block of RAM starting upon an even address, all of the chunk IDs and Sizes will conveniently fall upon even memory addresses).

In the preceding example, the group ID was 'FORM'. There are 2 other group IDs as well. A 'CAT ' is a collection of many different FORMs all stuck together consecutively in 1 IFF file. For example, if you had an animation with 6 sound effects, you might save the animation frames in an ANIM FORM, and you might save the sound effects in several AIFF FORMs (one per sound effect). You could save the animation and sound in 7 separate files. The ANIM file would start this way:

FORM	
120000	Whatever the size happens to be (this is expressed in 32 bits).
ANIM	

Each AIFF file would start this way:

FORM	
8000	whatever size.
AIFF	

If the user wanted to copy the data to another disk, he would have to copy 7 files. On the other hand, you could save all the data in one CAT file.

CAT	
4+120008+8008+2028+...	The total size of the ANIM and the 6 AIFF files.
' '	Type of CAT. 4 spaces for the type ID means "a grab bag" of IFF FORMs are going to be inside of this CAT. If it just so happened that all of the enclosed FORMs were 1 type, such as ILBM, then this type ID would be 'ILBM'.

FORM	
120000	
ANIM	
	...all the chunks in the ANIM file placed here. (Note: ANIMs have imbedded ILBM FORMs. The guy who devised the ANIM type of IFF file broke the rules by mistake, and nobody caught his error until it was too late).

FORM	
8000	

AIFF

	...all the chunks in the first sound effect here.

FORM	
2020	
AIFF	
	...all the chunks in the second sound effect here.

	...etc. for the other 4 sound effects.

To further complicate matters, there are LISTs. LISTs are a lot like CATs except that there is an optional, additional group ID associated with LISTs. That ID is a PROP. LISTs can have imbedded PROPS just like an ILBM can have an imbedded CMAP chunk. A PROP header looks very much like a FORM header in that you must follow it with a type ID. For example, here is an ILBM PROP with a CMAP in it.

PROP	Here's a PROP.
4+14	Here's how many bytes follow in the PROP.
ILBM	It's an ILBM PROP.
'CMAP'	Here's a CMAP chunk inside of this ILBM PROP.
6	There are 6 bytes following in this CMAP chunk.
0,0,0,1,1,4	

LISTs are meant to encompass similiar FORMs (i.e. several AIFF files stuck together). Often, when you have similiar FORMs stuck together, some of the chunks in the individual FORMs are the same. For example, assume that we have 2 AIFF sound effects. AIFF FORMs can have a NAME chunk which contains the ascii string that is the name of the sound effect. Also assume that both sounds are called "car crash". With a CAT, we end up having 2 identical NAME chunks in each AIFF FORM like so:

CAT	We put the 2 files into 1 CAT.
4+1008+508	
AIFF	It's a CAT of several AIFF FORMs.

FORM	Here's the start of the first sound effect file.
1000	
AIFF	
	...other chunks may be inserted here.
NAME	Here's the name chunk for the 1st sound effect.

9

'car crash',0

...other chunks may be inserted here.

FORM Here's the start of the 2nd sound effect file.

500

AIFF

...other chunks may be inserted here.

NAME Here's the name chunk for the 2nd sound effect. Look familiar?

9

'car crash',0

...other chunks may be inserted here.

With a LIST, we can have PROPs. A PROP is group ID that allows us to place chunks that pertain to all the FORMs in the LIST. So, we can rip out the NAME chunks inside both AIFF FORMs and replace it with one NAME chunk inside of a PROP.

LIST Notice that we use a LIST instead of a CAT.

4+30+990+490+...

AIFF

PROP Here's where we put chunks intended for ALL the subsequent FORMS; inside a PROP.

22

AIFF Type of PROP.

NAME Here's the name chunk inside of the PROP.

9

'car crash',0

FORM Here's the start of the first sound effect file.

982 Size is 18 bytes less because no NAME chunk here.

AIFF

...other chunks may be inserted here, but no NAME chunk needed.

FORM Here's the start of the 2nd sound effect file.

482

AIFF

...other chunks may be inserted here, but no NAME needed for this guy either.

Notice that the PROP group ID is followed by a type ID (in this case AIFF). This means that the PROP only affects any AIFF FORMs. If you were to sneak in an SMUS FORM at the end, the NAME chunk would not apply to it. Also, if you included a NAME chunk in one of the AIFF FORMs, it would override the PROP. For example, assume that you have a LIST containing 10 AIFF FORMs. All but 1 of them is named "Snare Hit". You can store a NAME chunk in a PROP AIFF for "Snare Hit". Then, in the one AIFF FORM whose name is not "Snare Hit", you can include a NAME chunk to override the NAME chunk in the PROP.

It should be noted that you can take several LISTs and squash them together inside of a CAT or another LIST. Personally, I have never seen a data file with this level of nesting, and doubt that it would be of much use.

In the above examples, psuedo code was used to represent the headers. Let's look at how a hex file viewer might display the actual contents of an IFF file (in hex bytes). First, an IFF header for a FORM AIFF, psuedo code.

FORM

4096

AIFF

Now here's a view of the actual data file.

46 4F 52 4D	FORM
00 00 10 00	hex 00001000, or 4096 decimal
41 49 46 46	AIFF

Note that the ULONG byte count is stored in Big Endian order (ie, the Most Significant Byte is first, and the Least Significant Byte is last). This is how the Motorola 680x0 stores long values in memory (ie, the opposite order of Intel 80x86). IFF files use Big Endian order for all 16-bit (ie, SHORT) and 32-bit (ie, LONG) values.

Microsoft decided that IFF was a good idea, but since Windows is traditionally tethered to Intel CPUs, a version of IFF was needed which stored LONG or SHORT values in Little Endian order. So, MS decided to create some new group IDs. MS took the FORM ID and created a Little Endian version of it known as RIFF. For example, the WAVE file format has a RIFF group ID. All of the SHORT and LONG values in the file are stored in Little Endian order. Let's take a look at an example header for a WAVE file. Assume that there are 258 bytes of data after the byte count.

52 49 46 46	RIFF
02 01 00 00	hex 00000102, or 258 decimal
57 41 56 45	WAVE

Note that the ULONG byte count is stored in Little Endian order (ie, the Least Significant Byte is first, and the Most Significant Byte is last). Good old backwards-thinking Intel.

Now, there's some real justification for creating a RIFF group ID, if you're working with an Intel CPU. But Microsoft couldn't stop there. True to their "not made here, so if we're going to accept it, we have to inflict our brutish, unneeded brand upon it" mentality, Microsoft created another group ID called RIFX. What's an RIFX file? It's simply a FORM with RIFX replacing the FORM ID. So, if you want to turn a FORM AIFF into a RIFX AIFF, you just change the first 4 bytes to RIFX. Needless to say, nobody has ever used the RIFX group ID, and it will undoubtably suffer a justifiably ignoble disappearance.

Just like everyone else, programmers make mistakes. As mentioned before, the Amiga's ANIM file format was a mistake. It puts FORM headers inside of a FORM group ID. That's not supposed to happen. You can put FORM headers inside of a CAT or LIST, but not another FORM. A mistake was also made with the MIDI file format. The programmer who devised it didn't put a proper IFF header on the file. It should be:

FORM	group ID. Indicates an IFF file that contains one type of data.
3000	whatever size the file happens to be.
MIDI	type of data. What follows will be chunks as defined by the MIDI type of IFF file.

But the programmer omitted the FORM group ID, and simply put the MThd chunk first. So, a MIDI file starts as so:

MThd	Chunk ID.
6	size of MThd chunk.

Another deviation from the standard occurs with padding out odd-sized chunks with an extra byte. Some programmers didn't bother doing this when devising new IFF type files, and occasionally, one will come across some specification for a new IFF type that allows odd-sized chunks.

Unfortunately, these programmers released their work based upon these aberrations before getting that work reviewed by other programmers who might have offered good reasons why the aberrations should be corrected. It makes it that much harder for software to read and write files if it has to deal with aberrations of the IFF standard. There's no reason for that, particularly when a strict adherence to the standard sacrifices almost nothing in the way of quality and efficiency over an aberration. But try to tell that to a paranoid programmer who thinks that if he shows anyone what he's doing before his product is shrink-wrapped, someone will steal his soul... well, IFF does give the computer industry a means for resolving needless hassles with data file formats, and it has worked very successfully in a number of instances, although occasionally people don't always use the standard wisely, or don't quite grasp EA's altruistic notion that there is no good reason why a file format should ever be proprietary or unpublished. (I urge consumers to avoid products where that is the case).

Here are answers to questions about MIDI files. Questions are:

[What's a MIDI file?](#)
[How is a MIDI file different than a WAVE or MP3 file?](#)
[Why does a particular MIDI file play incorrectly?](#)
[What's an RMID file?](#)
[How do I play a Macintosh MIDI file on an IBM PC?](#)
[How do I transfer a hardware sequencer's patterns to my computer for use with my sequencer software?](#)
[How do I create an audio CD from a MIDI file (or convert a MIDI file to a WAVE file)?](#)

What's a MIDI file?

A MIDI file is a data file. It stores information, just like a text (ie, ASCII) file may store the text of a story or newspaper article, but a MIDI file contains musical information. Specifically, a MIDI file stores MIDI data -- the data (ie, commands) that musical instruments transmit between each other to control such things as playing notes and adjusting an instrument's sound in various ways.

MIDI is binary data, and a MIDI file is therefore a binary file. You can't load a MIDI file into a text editor such as Notepad and view it. (Well, you can, but it will look like gibberish, since the data is not ASCII, ie, text. Of course, you can use my [MIDI File Disassembler/Assembler utility](#) to convert a MIDI file to readable text, edit it in Notepad, and then convert it back to a MIDI file using the same software).

The MIDI file format was devised to be able to store any kind of MIDI message, including System Exclusive messages, along with a timestamp for each MIDI message. A timestamp is simply the time when the message was generated. Using the timestamps, a sequencer can playback all of the MIDI messages within the MIDI file at the same relative times as when the messages were originally generated. In other words, a sequencer can playback all of the note messages, and other MIDI messages, with the original "musical rhythms". A MIDI file can also store other information relating to a musical performance, such as tempo, key signature, and time signature. A MIDI file is therefore a generic, standardized file format designed to store a "musical performance", and is used by many sequencers. A MIDI file even has provisions for storing the names of tracks in a sequencer, and other sequencer settings.

MIDI files are not specific to any particular computer platform or product.

There are 3 different "Types" (sometimes called "Formats") of MIDI files. A Type 0 file contains only one track, and all of the MIDI messages (ie, the entire performance) are placed in that one track, even if this represents many musical parts upon different MIDI channels. A Type 1 file separates each musical part upon its own track. Both Type 0 and 1 store one "song", "pattern", or musical performance. A Type 2 file, which is extremely rare, is akin to a collection of Type 0 files all crammed into one MIDI file. Type 2 is used to store a collection of songs or patterns, for example,

numerous drum beats. (If you need to convert a MIDI file to the various Types, use my Midi File Conversion utility).

Besides sequencers, other software and hardware devices may use MIDI files. For example, a patch editor may store an instrument's patch settings in a MIDI file, by storing System Exclusive messages (received from the instrument) within the MIDI file. In this case, the patch editor may not care about the timestamp associated with each SysEx message.

How is a MIDI file different than a WAVE or MP3 file?

A MIDI file stores MIDI messages, which are commands that tell a musical device what to do in order to make music. For example, there is a MIDI message that tells a device to play a particular note. There is another MIDI message that tells a device to change its current "sound" to a particular patch or instrument. Etc. The MIDI file also stores timestamps, and other information that a sequencer needs to play some "musical performance" by transmitting all of the MIDI messages in the file to all MIDI devices. In other words, a MIDI file contains hundreds (to thousands) of instructions that tell one or more sound modules (either external ones connected to your sequencer's MIDI Out, or sound modules built into your computer's sound card) how to reproduce every single, individual note and nuance of a musical performance.

A WAVE file stores a digital audio waveform. This data must be played back upon a device with a Digital To Analog Converter (ie, DAC) such as a MIDI sampler (ie, the AKAI S1000 for example) or a computer sound card's DAC. There are no timestamps, or other information concerning musical rhythms or tempo stored in a WAVE file. There is only digital audio data. Typically, a WAVE file is used to store a looped, single-pitched waveform (which a sampler transposes and plays back over a range of MIDI note numbers), or a short, non-looped percussive sound or sound effect, or often a digital audio recording (ie, stereo mixdown) of some musical performance stored upon your hard drive.

The only way to record and store a musical performance within a WAVE file is to digitize the audio output of all instruments while they play that performance. The result will be a typically large WAVE file that represents the digitized "sound" of all instruments playing the musical piece in realtime. The act of doing such is often referred to as "Hard disk recording" because the WAVE data usually has to be recorded directly to a large Hard Drive while the DAC digitizes the performance.

A MIDI file allows easy editing of the individual musical parts, because each part is usually assigned to its own MIDI channel, and it's easy to separate that part's MIDI data from the other parts' MIDI data, based upon the MIDI channel in each MIDI message. On the other hand, you can't easily separate the digital audio data of one instrument from the digital audio data of another instrument, if the two were digitized simultaneously into one WAVE file. For editing of individual musical parts, each part should be digitized and stored in its own WAVE file. Due to the difficulties in separating musical parts from a WAVE file, a conversion from WAVE format to MIDI format is not too feasible, although the reverse is not a problem.

One benefit of a WAVE file is that its "sound" is not usually dependent upon the playback device. A WAVE file should sound the same upon different equipment (with reasonably similiar specs). On the other hand, a MIDI file can sound considerably different upon different MIDI gear. This is because the MIDI file doesn't dictate what means an instrument uses to produce its sound. The MIDI file may sound very different upon an instrument that uses FM synthesis than it will sound upon an instrument using "wavetable synthesis" (ie, looped, digital audio waveforms in ROM). But, standards such as *General MIDI* seek to alleviate some of the discrepancies between MIDI devices.

An MP3 file, like a WAVE file, stores digital audio data. So a MIDI file and an MP3 file are different in exactly the same way that a MIDI file and a WAVE file are different. Indeed, a WAVE and MP3 file are two different ways of storing the exact same type of data. The primary difference between a WAVE and MP3 file is that the latter uses compression to squeeze the data down in size, resulting in a typically much smaller file size.

Why does a particular MIDI file sound so bizarre when played back upon my equipment? It sounds like "jungle bunnies from Hell" are performing. And every time that I play the file, it sounds different depending upon what other MIDI file I played previously. Sometimes, certain parts don't sound at all, and other times they do.

A MIDI file may sound odd upon your equipment for one of two reasons.

First, the MIDI file may not have the proper "MIDI setup events" at the beginning of the file. If these events are missing, then the MIDI file won't properly setup your equipment with the necessary instrumentation, and panning and volume (ie, mix), etc, that is necessary to play the MIDI arrangement as it was intended by the creator (henceforth referred to as the author) of the MIDI file.

By "MIDI setup events", I'm referring to certain MIDI messages. For one thing, the MIDI file may not contain MIDI Program Change messages at the start of the arrangement to setup each MIDI sound module to play the desired Patches. The net result is that what was supposed to be a Piano part, for example, may end up being played upon a module that is currently setup with a Tuba patch. Having a Tuba play the Piano part probably isn't going to sound good.

Furthermore, the author may have neglected to put some important controller events at the start of each track. There are controller events to set volume, panning, reverb and chorus levels, modulation (ie, vibrato amount), etc. For example, if volume controllers are omitted, then the file will playback with whatever volumes all of your sound modules are currently set to. If a previously played MIDI

file has "faded out" the volume of a particular module, then that module may not even be sounding the part that it is supposed to be playing. Other clues that these important events may be missing is if parts inexplicably play with vibrato, odd panning, out of tune, or with the sustain pedal held.

Very often, an author forgets to place these important "setup events" at the start of the arrangement. There are two ways to fix such a MIDI file. You can manually edit it, inserting the proper setup events for your equipment. I'll describe this process below. Or, you can do it the easy way by using my MIDI File Cleanup utility, available upon this web site.

If your sound modules understand the MIDI *Reset All Controllers* message, I recommend inserting this at the very start of each track. This will instruct your module to set all of the MIDI controllers it understands (and usually Pitch Wheel and Aftertouch) to default values.

NOTE: You only need to put these setup events upon tracks that are assigned to different MIDI channels. For example, if both track 1 and 2 are assigned to the same MIDI channel, then you need to put the setup events in only one of those two tracks.

Also, insert a Bank Select controller with a value of zero, and then insert an appropriate Program Change event for each track. For example, if the name of the track is "Harpsichord", and you're using a General MIDI module, insert a Program Change for program #7 (or 6 if your software counts the first program as 0 instead of 1). (See the General MIDI article for a listing of GM programs). Make sure that the Program Change event occurs after the Bank Select. For the Drum tracks, I recommend a Bank Select controller of 0, and a Program Change of 0. Again, you only need the setup events on one drum track if all drum tracks are assigned to the same MIDI channel.

If your sound module doesn't recognize *Reset All Controllers*, then I recommend putting in the following controllers with these values: Volume (controller #7) at a value of 100, Pan (10) at 64, Expression (11) at 127, Mod Wheel (1) at 0, Hold Pedal (64) at 0, Effects Level (91) at 64 (assuming that it controls Reverb, otherwise set it at 0), and Chorus Level (93) at 0. For modules that recognize any of the following controllers, you may wish to also add that respective controller: Balance (8) at 64, Soft Pedal (67) at 0, Portamento (65) at 0, and Legato Pedal (68) at 0. To reset Pitch Wheel Range to +/- 2 steps, insert the following 3 controllers in this order: NRPN MSB (98) at 0, NRPN LSB (99) at 0, and Data Entry Slider (6) at 2. Finally, you should also reset the Pitch Wheel to 0, and Channel Pressure to 0 (as well as Key Pressure if your module implements that).

Alternately, some sequencers have a feature to automatically spit out default setup events for each track before playing any MIDI file. This may cure some mix problems, but you'll probably still need to adjust instrumentation (ie, insert Program Change).

On the other hand, the author may have included setup events at the start of the MIDI file, but your MIDI equipment's Patch set and MIDI channel assignments may be different than the ones used by the author. One very annoying ramification of this is when different MIDI channels are used for drum parts. The net result is that you often end up hearing instrumental parts (ie, piano, guitar, etc) played on drum sounds, and drum notes being played as piano/guitar/etc notes. Also, if parts are being played in odd octaves (ie, the part sounds like it's a few octaves too high or low), this is another

ramification of conflicts between the equipment which the author used and your own equipment.

The General MIDI specification is designed to get everyone to setup and use their equipment in a compatible way. For example, all drum parts are always upon MIDI channel 10, and all drum sound modules should be set to this channel. The GM spec details a lot of ways in which gear should be setup, in order to eliminate as many trivial incompatibilities as possible. If the author uses GM-compliant sound modules to create the MIDI file and remembers to put the proper MIDI setup events in the MIDI file, and you play that file with some GM-compliant sound modules, you'll hear the proper instrumentation and mix.

NOTE: If a MIDI file seems to turn off the Drum Kit permanently in a Roland card (ie, it seems to disable the Drum Kit for all MIDI files that you play afterwards), then the problem is likely due to a Bank Select controller and/or Program Change in one of the drum tracks. If a Roland card is asked to switch its Drum Kit to a bank/program that doesn't exist in the card, sometimes the Roland card will disable the drum kit. Simply edit the file to change that Bank Select controller and/or Program Change event to a value understood by the Roland card, for example, set the value to bank 0 and program 1.

What's an RMID file (ie, it has a filename extension of .rmi)? How do I extract the MIDI data from it? My sequencer won't read it.

An RMID file is simply a MIDI file that has some extra bytes prepended to it. If you remove those extra bytes, then you'll end up with a standard MIDI file. To edit the RMID file, you need to use a program that edits binary (ie, not text) files to chop off those extra bytes (ie, characters) at the start of the file. Such a program is usually referred to as a "hex (or binary) editor. Do not use a text editor such as Notepad.

Load the RIFF file into the hex editor, and chop of all bytes before you see the 4 ascii characters of "M', 'T', 'h', and 'd' in that order. (ie, The hex editor should have a split-window display were it shows the value of each byte, and then displays its ascii character on the other side of the window. Look at the ascii character display). Do not chop off the 'M', 'T', 'h', 'd' (as that is the start of the embedded MIDI file) -- just all characters before those four.

A free alternative is to use my MIDI File Disassembler/Assembler utility to convert the MIDI file to readable text. This will automatically strip out the RMID header. Then, you can immediately use the same utility to convert the text file back to a MIDI file.

How can I play, on my IBM PC, a MIDI file that was made on a Macintosh? My sequencer won't

read it.

A MIDI file made on a Macintosh may likely have 128 extra bytes prepended to it (ie, a Mac Binary Header). If you remove those first 128 bytes (ie, all the bytes before you see the first 4 characters of 'M', 'T', 'h', and 'd') then you'll end up with a standard MIDI file. Use a program that edits binary (ie, not text) files to chop off the first 128 bytes (ie, characters).

A free alternative is to use my MIDI File Disassembler/Assembler utility to convert the MIDI file to readable text. This should automatically strip out the Mac header. Then, you can immediately use the same utility to convert the text file back to a MIDI file.

Is there some way to take patterns stored in a proprietary format on some disk for my hardware sequencer (or stored in its internal memory) and transfer them to my computer so that I can edit those sequences with some computer sequencer software such as CakeWalk? My computer can't seem to read the disks that my hardware sequencer uses, and some proprietary data dump to the computer is no help in getting CakeWalk to recognize the data as musical sequences.

If your hardware sequencer outputs MIDI Start/Stop and MIDI clock messages (or can sync to those), then it's very easy. You'll need some sequencer software on your computer capable of syncing to MIDI Clock (or which generates MIDI Start/Stop and Clock messages if your hardware sequencer can alternately sync to MIDI). For example, CakeWalk is such a program (and I'll refer to it below).

Assuming that your hardware sequencer outputs MIDI Start/Stop and MIDI Clock -- first, connect the MIDI Out of the hardware sequencer into the computer's MIDI In. Next, set CakeWalk's sync to MIDI instead of Internal (ie, set the computer sequencer software to sync to incoming MIDI Clocks). Then, set CakeWalk's Settings->Channel Table so that each MIDI channel is being recorded upon a separate track (ie, set MIDI channel 1 to track 1, MIDI Channel 2 to track 2, etc). Make sure that you disable any quantize of data being recorded, or set the quantize to a very small note value. Then, you need to get CakeWalk ready to record.

Now simply press the Play button on the hardware sequencer, and let it play all of its MIDI tracks into CakeWalk. (ie, CakeWalk should kick into record when you press the hardware sequencer's Play button if the sync is working). For all CakeWalk knows, it's merely recording the MIDI output of a bunch of daisy-chained synths being played by numerous musicians live. CakeWalk doesn't need to know that it's recording the output of another sequencer. In other words, you're transferring the data by playing it back, and simultaneously recording it, in real-time.

Of course, this doesn't transfer such things as track names, key and time signatures, tempo settings, and other display settings that your hardware sequencer may support. It also separates the MIDI data by channel assignment which may be different than the way that the tracks were arranged upon your hardware sequencer. (For example, if your hardware sequencer had 3 separate tracks all containing data upon MIDI channel 1, these 3 will be merged into 1 CakeWalk track). But it captures all of the MIDI data with the proper timing, and that's the important thing.

To speed up the transfer process, you could increase the tempo on your hardware sequencer. But you don't want to increase the tempo too much as this transfer method inevitably introduces slight deviations in the timing of individual events, and that is made worse by faster tempos. (Hopefully, your ear won't hear those deviations, but if any timing sounds too odd, then slow down the tempo when you transfer the file).

If your hardware sequencer supports syncing to MIDI Clock, you can transfer a MIDI file back by reversing the above setup (ie, setting CakeWalk to output MIDI Start/Stop and Clocks, and set the hardware sequencer to sync to MIDI. Connect the computer's MIDI Out to the hardware sequencer's MIDI In, press record on the hardware sequencer, and then play the MIDI file in CakeWalk).

This transfer method isn't quite as precise nor convenient than if you use a hardware sequencer that writes standard MIDI files upon floppy disks that can be read by your computer (or have a computer program that can accept a dump from your hardware sequencer and convert it to standard MIDI format). But it works between any 2 sequencers.

How do I create an audio CD from a MIDI file (or convert a MIDI file to a WAVE file)?

An audio CD (designed to be played in standalone CD players) contains tracks of digital audio (in WAVE file format). There is also some "control information" on the CD that tells the CD player how many musical tracks are on the CD, and in what order to play them. So, to create an audio CD, you need software that is specifically written to do that, such as Adaptec's "Easy CD Creator" (which allows you to create various types of CD's, among them, audio CD's) or Nero Express (or Burning ROM).

Easy CD Creator, or Nero Express, are typical of programs that allow you to create an audio CD. The software lets you pick out various WAVE files (or sometimes MP3, or other digital audio formats -- but not MIDI files) and drop them into a "layout". The layout simply determines in what order the WAVE files will become musical tracks on the CD. Then, you tell the software to burn an audio CD

using those WAVE files.

So, the major task is first converting your MIDI file to a WAVE file so that you can subsequently use Easy CD Creator to create an audio CD. To do this, you need a program capable of recording a WAVE file, such as "Cool Edit" or "Sound Forge". And you need a MIDI player program such as Cakewalk or Cubase or even something as simple as the Windows Media Player. (If you're using a program that can play MIDI tracks while it is also recording a digital audio track, such as Cakewalk Pro Audio, then you can do everything with just the one program).

To convert a MIDI file to a WAVE file, follow these steps:

1. Attach the audio out of your MIDI sound module(s) to the line in jack of your sound card. If your sound module is built into the sound card, simply run the sound card's "line out" (or "speaker out") back into its own "line in" (or "microphone").

2. Open the MIDI file in the MIDI player program (ie, Windows Media Player).

3. Run the digital audio recording program. Set it to record a stereo, 16-bit, 44Khz audio track. Also, select your sound card's input as the recording source.

4. Put the digital audio recording program into record. (Before doing this, you may want to first set the recording level by playing the MIDI file while you adjust the recording program's record volume).

5. Quickly flip back to the MIDI player program and put it into play. Now let the MIDI file play back while the digital audio recording program is recording the sound of your sound module playing that MIDI file.

6. When the playback ends, stop the digital audio recording. Trim off any excess silence at the start and end of the audio track, and then save it in WAVE file format. Now you're ready to add this WAVE file to your CD software's "layout".

In lieu of using the above procedure to create a WAVE mix from a MIDI file, there are also some programs that can directly convert a MIDI file to a WAVE file. Such a program uses its own internal set of waveforms to mathematically create the WAVE mixdown (rather than using the sound of your own sound modules). So, if you don't like the program's built-in sounds, then you'll either have to find a new waveform set (if the program supports that -- some programs support loading SoundFonts), or try another such program.

Recommended reading:

What is MIDI?
What is a sequencer?
General MIDI

With the advent of computer audio cards that can record and playback digital audio, computer software has appeared which turns a computer into a device that can not only record and playback that digital audio, but also present that data in a way that makes it easy for musicians to view and edit it. The data can be graphically displayed upon a computer monitor, and can be manipulated with the mouse, for example. Furthermore, computer software can also perform its own manipulations of that data, yielding effects such as delay, transposition, chorus, compression, etc, sometimes even in real-time (ie, while the audio data is playing back). The wide range of computer audio products also means that a computer digital audio system can be tailored to many budgets. And being that computers are typically more easily upgradable than dedicated digital audio units (for example, adding a second hard drive to accomodate more digital audio tracks), and can do other things besides digital audio work, they are often ultimately more versatile and cost effective than dedicated digital audio units. In short, with good audio hardware and software, computers make very good digital audio workstations.

Examples of software that supports both digital audio recording, as well as MIDI, are CakeWalk Pro Audio, PG Music's Power Tracks, Steinberg's Cubase, etc. Examples of programs that specialize in digital audio recording (and may have a more powerful and easier feature set for digital audio editing than the sequencers) are Cool Edit, Sound Forge, SAW, Samplitude, etc.

Digital Audio Recording

A typical computer system works with digital audio in the following way. First, to record digital audio, you need a card with an Analog to Digital Converter (ADC) circuitry on it. This ADC is attached to the Line In (and Mic In) jack of your audio card, and converts the incoming analog audio to a digital signal that computer software can store on your hard drive, visually display on the computer's monitor, mathematically manipulate in order to add effects or process the sound, etc. (When I say "incoming analog audio", I'm referring to whatever you're pumping into the Line In or Mic In of your sound card, for example, the output from a mixing console, or the audio output of an electronic instrument, or the sound of some acoustic instrument or voice being feed through a microphone plugged into the sound card's Mic In, etc). While the incoming analog audio is being recorded, the ADC is creating many, many digital values in its conversion to a digital audio representation of what is being recorded. Think of it as analogous to a cassette recorder. While you're recording some analog audio to a cassette tape, the tape is constantly passing over the record head. So the longer the passage of music you record, the more cassette tape you use to record that analog audio signal onto the tape. So too with the conversion to digital audio. The longer the passage of music you record (ie, digitize), the more digital values are created, and these values must be stored for later playback.

Where are these digital values stored? Well, as your sound card creates each value, that data is passed to the card's software driver which then passes that data to the software program managing the recording process. Such software might be CakeWalk recording a digital audio track, or SAW, or Samplitude, or Windows Sound Recorder, or any other program capable of initiating and managing

the recording of digital audio. Whereas the program may temporarily accumulate those digital audio values in the computer's RAM, those values will eventually have to be stored upon some fixed medium for permanence. That medium is your computer's hard drive. (For this reason, sometimes people refer to the process of recording digital audio to a hard drive as "Hard Disk Recording". Henceforth, I will abbreviate "hard drive" as HD). Usually, how this works is that the program accumulates a "block" of data in RAM (while the sound card is digitizing the incoming analog audio), for example 4,000 data values, and then writes these 4,000 values into a file on your HD. (It's a lot more efficient to write 4,000 values at once to your hard drive, than it is to write those 4,000 values one after the other separately). All of these 4,000 values go into one file. Usually, the format for how the data is arranged within this file follows the WAVE file format. (But there are other formats that may also be used by programs to store digital audio values. For example, AIFF is often used on the Macintosh. AU format is used on Sun computers. MP3 is a popular format nowadays because it compresses the data's size. Etc. Any of these formats could be used on any computer, but WAVE is considered the standard on a Windows-based PC). Now, if the recording process is still going on, the software program will collect 4,000 more values in RAM, and then write them out to the same WAVE file on the HD (without erasing the previously stored 4,000 values -- ie, the values accumulate within the WAVE file, so that it now has 8,000 values in it). This process continues until the musician halts the recording process. (ie, If you let recording continue long enough, it will eventually fill up your HD with digital audio values in one, big WAVE file). At that point, the WAVE file is complete, and contains all of the digital audio values representing the analog audio recorded.

So, digital audio recorded by most PC programs is predominantly stored in a WAVE file on your HD, and that file is created while the digital audio is being created/recorded.

<hr>

Digital Audio Playback

In order to subsequently playback this digital audio (ie, WAVE file), you need a card with a Digital To Analog Converter (DAC) circuitry on it. Needless to say, most sound cards have both an ADC and a DAC so that the card can both record and play digital audio. This DAC is attached to the Line Out jack of your audio card, and converts the digital audio values back into the original analog audio (that was initially recorded during the recording process). This analog audio can then be routed to a mixer, or speakers, or headphones so that you can hear the recreation of what was originally recorded. You need software to manage the playback of digital audio, and not surprisingly, the same program that was used to manage the recording process usually can also manage the playback. For example, CakeWalk can playback the digital audio track that it recorded. The playback process is almost an exact reverse of the recording process. The program reads a block of digital audio data from the WAVE file on the HD. For example, CalkWalk may read the first 4,000 data values. (It's more efficient to read 4,000 values off of the HD at once, than it is to read those 4,000 values one after the other separately). Then, the program passes each one of these values to the card's driver which feeds it to the card's DAC. The program then reads the next 4,000 values from the WAVE file, and plays those back as described. In other words, the sound card is recreating the original analog audio while the program is reading the digital audio values off of the HD and passing them back to

the card. This continues until the program has played all of the values in the WAVE file (or until the musician interrupts the playback).

Data processing during playback

Some programs can optionally perform some mathematical manipulation of the digital audio values immediately before the data is passed to the card's driver (ie, during playback). Such manipulations may be to add effects such as reverb, chorus, delay, etc. Programs that do such realtime processing are SAW and Samplitude.

Some programs also can process the digital audio to do some valuable things such as "time-stretching" and "pitch shift" (although these are often too computationally complex for today's computer to do during playback. So, this processing usually has to be applied before playback, and may be "destructive" in that it permanently alters the digital audio data).

Time-stretching: When you play a one-shot (ie, non-looped) waveform, it lasts only so long (ie, in terms of time). For example, maybe you've got digital audio tracks of a piece of music whose duration is 2 minutes. Sometimes, people need to adjust the length of time over which the waveform plays. For example, maybe the producer of a movie says "I want this piece of music to last exactly 2 minutes and 3 seconds in order to fit with this filmed scene I have which is this long". Well, if you had a MIDI track, you'd just slow the tempo a little in order to make the music last that extra 3 seconds. The music would sound exactly the same, but at a slightly slower tempo. OK, so how do you "slow down" the digital audio tracks? Well, you could reduce the playback rate a little. But, if you've ever used a sampler, you'll notice that when you play a waveform at a rate different than it was recorded, this changes the character of the waveform itself. You don't just hear a different tempo, you hear different pitch, vibrato, tremulo, tone, etc. So, time-stretching was devised to take a waveform, analyze it, and change its length without (hopefully) changing its characteristics. The net result is that you get the same effect that you had by changing the tempo of the MIDI track; ie, merely a change in tempo/duration rather than a change in the characteristics of the waveform. (Nevertheless, there is always some potential for a change in timbre when time-stretching algorithms are applied to a waveform).

Pitch shift: This changes the note's pitch without altering the playback rate (which would alter other characteristics of the waveform). So, if you sampled the middle C note of a piano at 44KHz, but when you play it back at 44KHz, you really want to hear a D note, you could apply pitch shift to it to create a waveform whose "root" (ie, the pitch you get when you playback at the same sample rate as when recorded) is D. If you take a musical performance with "rhythms" in it, and apply pitch shift, you should get a different pitch, but retain the same rhythms (ie, tempo).

Virtual Tracks

Although most sound cards have only 2 discrete digital audio channels (ie, stereo digital audio capabilities), many programs such as CakeWalk support recording and playing many more tracks of digital audio. And yet, CakeWalk seemingly plays more than 2 digital audio tracks using such a card. How is this done? It is accomplished through the use of "virtual tracks". What this means is that the program allows you to record as many digital audio tracks as you like, mono and/or stereo. The only limitation is your HD space, plus how many tracks can be read off of your HD at the required rate of playback. (You can record only 2 mono tracks, or 1 stereo track <u>at one time</u>, due to the card's limit of only 2 channels. But, you can do as many iterations of the recording process as you wish to yield many more than 2 tracks. It's on playback that the concept of virtual tracks works). For example, you could record 4 mono tracks, plus 3 stereo tracks (if you had a fast enough HD -- doubtful on slow IDE stuff). You can set the individual panning for each of the mono tracks. Then, upon playback (ie, when CakeWalk reads the data for all of those digital audio tracks from their WAVE files), CakeWalk itself mathematically mixes all of the digital audio tracks into one stereo digital audio mix, and outputs this mix to the sound card's stereo DAC. In other words, CakeWalk itself functions as a sort of "digital mixer", mixing many mono and stereo tracks together (during playback) into one stereo digital audio track. So, using any sound card with a stereo digital audio DAC, you can actually record as many tracks as you like. CakeWalk virtualizes the card so that it appears to have many digital audio tracks, mono and/or stereo. Most pro programs that work with digital audio support the concept of "virtual tracks".

There are a few drawbacks to this scheme though. First, the more tracks that you record, the more that you have to back off on the individual volume of each track. Why? Because when all of the tracks are mathematically summed together by CakeWalk, if at any point the sum exceeds a 16-bit value, you'll get clipping. The more tracks that you sum, the more likely you are to get clipping -- unless you back off on the individual track volumes more with each added track. It doesn't matter if your card has 18-bit (ie, Tahiti) or 20-bit (ie, Pinnacle) DACs. CakeWalk itself performs the sum into a 16-bit mix, so there is an inherent 16-bit limitation to CakeWalk's output (or any other program that is limited to 16-bit digital audio -- some digital audio programs offer greater than 16-bit resolution for their internal mixing, such as 24-bit or 32-bit. These programs don't require you to back off on the individual wave volumes so much. If you're using a lot of virtual tracks, use a program with at least 24-bit resolution for its internal mixing). So the more tracks you record, the less dynamic range you get for each individual track as you turn down the individual volume to avoid clipping the mix. I doubt that you'd want to mix more than 4 mono virtual tracks to one card. (On the plus side, if you get more digital audio cards, you can then split up virtual tracks among them since CakeWalk can use more than 1 card simultaneously. But remember that most ISA cards require at least 1 DMA channel for playback, and 1 for full-duplex recording, and a PC has only 3 16-bit DMA channels, so you're pretty much limited to 2 ISA sound cards -- unless you get a Tahiti, Fiji, or Pinnacle which use a proprietary, non-DMA method of output, or a PCI card as those do not use motherboard DMA). Of course, you could get one of those cards with 8 digital audio channels like a DAL V8, Antex SoundCard, EMagic Audiowerk8, or DigiDesign Session 8, which usually have their own accompanying program to support better than 16-bit DACs and perhaps better throughput.

Secondly, you may lose some of the individual hardware control of the card when CakeWalk takes over the task of digitally mixing the output. For example, CakeWalk has to operate a Roland RAP-10 in its stereo mode, so you lose the RAP's individual control over each track's reverb/delay and chorus (effective when the RAP-10 channels are used as 2 Mono channels). In other words, most programs

take a generic approach to virtual tracks which may not support some of the more esoteric functions of certain cards. So, it's important that if you get a card that has something more than just a simple, stereo 16-bit DAC, you make sure that you have software support for that additional functionality lest you want it to be wasted by a program that treats your card as if it were a simple, stereo 16-bit DAC.

Hard Drive requirements

With digital audio tracks recorded to a HD, usually the limiting factor in how many virtual tracks can be simultaneously played, and how well the audio sounds, is the speed at which data can be read/written to your HD. A slow HD (more than 12ms access time) is usually the limiting factor in how many virtual tracks can be used. Also, you need a HD that has no problems with thermal recalibration (ie, lengthy adjustments the HD may make during reading/writing that can cause a delay in accessing the drive). These delays may cause a program to fail to feed a continuous stream of digital audio values to/from the sound card, and you'll then hear "glitches" in the audio. Newer HD designs have tended to minimize this problem. Select a good HD for digital audio. It may be even more critical than differences in sound card designs. If you go with an EIDE HD, make sure that you get one that supports bus mastering (in Mode 4) or at least DMA/33, and you use it with a motherboard that has a PCI EIDE controller and drivers that support EIDE bus mastering. (Win95's Service Pack 2 added such support to Win95). Otherwise, SCSI has an advantage in that most SCSI controllers support bus mastering, scatter-gather lists, and other features that make for efficient HD I/O, and which are supported by most operating systems' drivers.

Furthermore, since digital audio requires a constant stream of values throughout the recording process (unlike with MIDI), you need a large HD if you want to record long digital audio tracks. You'll use up 5 MEG of HD space for every minute of a 16-bit mono digital audio track recorded at 44.1Khz (and 10 MEG for a stereo track). In other words, recording a CD-quality digital stereo track that is 5 minutes long will use up 50 MEG of your HD.

Audio card requirements

Of course, this is not to say that the quality of your sound card isn't important too. If you've got a card with a cheap DAC or ADC, you're going to get noisy digital audio. (ie, The result will typically sound like a "graininess" to the audio or even "tape hiss". On the other hand, problems with the HD usually manifest in horrid distortion or weird noises such as pops and clicks). To get nicely recorded digital audio, and as many simultaneous virtual tracks as possible, you'll want both a fast, large HD with efficient controller I/O, and a card with a good DAC and ADC.

Unless you've got a card that has digital I/O so that you can run a digital connection right to a DAT deck, for example, you'll also want a card that has a clean (ie, low Thd distortion) and quiet (ie, low signal-to-noise ratio) audio output stage.

For more information about digital audio cards themselves, see [Digital Audio Cards](#).

Introduction

An "Algorithmic Composition program" is software that has the capacity to create a complete musical arrangement on your behalf. Such a program typically allows you to hear (ie, play) the arrangement that it creates, and even edit the arrangement somewhat. Therefore, such a program is a specialized breed of sequencer software.

The program composes all of the individual musical parts (for example, drums, bass guitar, piano, etc, parts). You need specify only what chord changes are in the arrangement (ie, what chord plays upon which musical bars and beats), what instrumentation you would like, and what "style" of music you wish.

Entering chord changes

Styles of music

The program usually offers a variety rock, jazz, country, waltz, rap, latin, etc, "styles" of music.

MIDI/Audio cards for a PC

The Old Days

Soon after the IBM "personal computer" (ie, PC) first appeared in the consumer market in 1984, companies began to make cards that plugged inside of the computer in order to play and record music/ audio/speech (since the IBM PC didn't have that capability built in). Some of the cards dealt only with digital audio playback and recording. Some other cards dealt with only MIDI input and output. Some cards even did both. (Remember that digital audio and MIDI are two entirely different things, although both typically serve the purpose of recording and playing musical performances).

At the time, the only operating system much in use for the IBM PC was called **MS-DOS**. It was Microsoft's precursor to Windows, and it was a text-based (ie, no graphical interface like we have now) operating system. The MS-DOS operating system was very primitive when it came to supporting audio/MIDI hardware. In fact, all that it supported was the internal PC speaker. If an MS-DOS program wanted to use any other audio hardware, such as a MIDI interface like the Roland MPU-401, or a sound card like the Creative Labs' Sound Blaster (ie, back then, these were the two most popular add-on cards for MIDI, and digital audio, respectively), the program had to manipulate the audio hardware directly, controlling it in order to make sound. If a particular program didn't directly support your particular audio card, then you wouldn't get sound from that program.

Because the very first, widely accepted MIDI interface for a PC was the Roland MPU-401, almost all MS-DOS MIDI programs directly supported this card. Because the very first, widely accepted Sound Card for a PC was the Creative Labs Sound Blaster, almost all MS-DOS game programs directly supported this card. (MS-DOS MIDI programs didn't support the SB because MIDI users weren't at all interested in the awful FM synthesis and very poor MIDI handling of the early SB. Likewise, MS-DOS game programs didn't support the MPU-401 because game players weren't interested in buying expensive external MIDI sound modules just to hear their game music. Furthermore, game programs needed to play digitized voice and sound effects on a card with a Digital to Analog converter, or DAC, which the SB had and the MPU-401 didn't. So, it was almost impossible to find audio hardware that could be used with both game software and music software).

Lots of companies making MIDI interfaces subsequently offered MPU-401 hardware compatibility so that their products would work with MS-DOS MIDI software that directly manipulated an MPU-401. For example, MusicQuest made the MQX-16. Likewise, lots of companies making Sound Cards offered Sound Blaster hardware compatibility so that their products would work with MS-DOS game software that directly manipulated an SB. For example, MediaVision made the PAS-16.

But typically, if you wanted both MIDI and digital audio support in your computer, then you had to not only buy two different cards to use with your software, but one of the cards had to be 100% hardware compatible with an MPU-401 and the other card had to be 100% hardware compatible with an SB. If you bought an MPU-401 clone from some company other than Roland, or an SB clone from some company other than Creative Labs, you often ran into hardware compatibility problems which

made the MS-DOS software not work with the card. So, in the old days, hardware incompatibilies were very common. Also, the market was prevented from innovating beyond the MPU-401 and SB "standards" because the hardware was tied to the software and vice versa. It was a catch-22 situation. The hardware manufacturers couldn't innovate because then their new hardware designs wouldn't work with any software. And the software programmers couldn't innovate because the hardware manufacturers were too reluctant to invest in making new hardware that didn't already have a lot of software support. (Consumers shunned hardware that wasn't supported by a lot of software, for good reason -- you can't do much with hardware that has little software support). So consumers were limited to buying hardware that had only the features and performance of an MPU-401 or SB, and software that supported only one or the other.

Windows to the rescue

Some years later, the Windows operating system was released. This time, Microsoft realized that some standards were needed for digital audio and MIDI support right in the operating system itself. Windows has specific functions in it to do various things with audio and MIDI. For example, it has a function to play a WAVE (ie, digitized audio) file on a sound card's Digital-to-audio Converter (ie, DAC). It has a function to send a MIDI message out of a MIDI interface's MIDI OUT jack. Etc. This group of audio functions is known as Windows *Media Control Interface* (ie, MCI). (In fact, Windows MCI also includes functions for video, as well as audio operation of CD-ROM drives. But as MIDI/ audio enthusiasts, we don't care about that). So rather than manipulate the audio hardware directly, a Windows program will instead call these audio functions in Windows. For example, if a Windows program wants to send a MIDI message out of a MIDI interface's MIDI OUT, rather than directly manipulating the card's MIDI OUT port like an MS-DOS program would do, the Windows program passes that MIDI message to the Windows function that deals with outputting MIDI messages.

So what does Windows do with that MIDI message? Well, if your sound card or MIDI Interface has a Windows MCI driver for it (and virtually all audio hardware ships with a Windows MCI "driver"), and the card itself has MIDI playback support, then Windows will be able to pass that MIDI message to the card's driver which will actually send the MIDI message out the card's MIDI OUT jack (or maybe play the MIDI message on a built-in MIDI sound module).

NOTE: If you can hear the Windows system sounds upon your audio card, then you definitely already have it setup with a Windows MCI driver (and a card that supports digital audio).

In this way, a Windows program (including Windows game programs) can support any audio hardware that ships with an MCI driver. The Windows program doesn't care what audio hardware design is used; MPU-401, Sound Blaster, or anything else. If the program needs to do MIDI playback, all that is required is audio hardware that actually can do MIDI playback, and its Windows MCI driver. If the program needs to do digital audio playback, all that is required is audio hardware that actually can play digital audio, and its Windows MCI driver. For example, CakeWalk Pro Audio does its MIDI and digital audio playback and recording via Windows MCI functions. Therefore, CakeWalk can use the digital audio and/or MIDI features of either some fancy high-end audio hardware or some cheap "game card". (Actually, CakeWalk is smart enough to use MCI drivers itself, rather than just via Windows MCI functions, and can therefore use more than one card

simultaneously for MIDI play/record. Other Windows programs strictly go through Windows MCI functions, and therefore their access to audio hardware is determined by the settings in Window's "MultiMedia" or "Sound" Control Panel utility, or Win3.1's MIDI Mapper).

So what Microsoft did was essentially make hardware/software incompatibilities a thing of the past. Consumers no longer have to worry about whether a particular software program works with their hardware, and vice versa. They are free to buy any software and use it with any hardware, and be confident that it should work. (This is not to say that all cards are created equal. Sure, a game card can be used to record a digital audio track with Cakewalk, just like a high-end card can. But the game card will typically create a track with a lot more audible "hiss" in it, and maybe roll off more of the high and low frequencies). Hardware manufacturers and software programmers were free to innovate, and since then, we have moved well beyond the MPU-401's MIDI limits and the SB's digital audio limits.

The trade-off

There's always a price to be paid. Sure, now consumers no longer have to ask "Is this hardware compatible with an MPU-401 or Sound Blaster?", nor "Does this software program require MPU-401 or Sound Blaster hardware compatibility?". The hardware incompatibility issues are pretty much irrelevant today, and today's software no longer requires specific audio hardware (although now it may require specific driver support, as I'll discuss below). Things have definitely improved there.

But, we have a new level of software support that we have to worry about, namely, driver support. Whereas it is completely irrelevant today whether the digital audio card you're considering is 100% hardware compatible with an SB, or the MIDI interface you're considering is 100% hardware compatible with an MPU-401, you do have to consider whether the card has good, complete driver support. If it doesn't, you could find yourself stuck with a useless piece of hardware, just like the folks back in the old days were sometimes stuck with useless hardware when it wasn't 100% compatible with an MPU-401 or SB, or they couldn't get software that supported their hardware.

As mentioned, virtually all audio cards made since the late 80's ship with a Windows MCI driver that offers the basic functionality of MIDI play/record and/or digital audio play/record. But over the years, Microsoft has added new features to the Windows operating system which drivers may utilize to give added functionality above the basic features. For example, some cards offer mixing capabilities that any advanced program can tap into (ie, if the program is likewise written for these new Windows features). For example, you may be able to pop-up a graphical "mixer" in Cakewalk to adjust the microphone recording level or speaker output (ie, Master Volume), and mute inputs/ outputs, and control other aspects of the card. With older or "low quality" drivers, you may have no such features, or you may have to use some special software that ships with the card to adjust these settings. (ie, You can't fully control such settings on the card via software, and/or from one program like Cakewalk).

Another example is that some drivers are "multi-client" (or "multi-instance") whereas others are "single-client". Multi-client drivers allow more than one simultaneously running program to use the hardware. With a single-client driver, if you're already running one program that is using the audio

hardware, and you try to simultaneously run a second program which wants to use that audio hardware, the second program will pop up an error message saying that "The device is busy". You have to close down the first program in order to run the second program. This can be annoying. (On the other hand, a properly written Windows program should avoid this error even with a single-client driver. Unfortunately, too many audio/MIDI programmers do not know how to write such software. If you encounter a program that gives you this error, send the following URL in an email to the tech support for that product: http://www.borg.com/~jglatt/tech/share.htm).

Microsoft has developed some new functions especially useful for new Windows software. These new functions are called DirectX, and they offer extra flexibility that MCI doesn't such as realtime mixing of many channels of digital audio. In particular, DirectSound is applicable to cards that play digital audio, and DirectMusic is applicable to MIDI. A Windows program has to be specially written to use these new functions, and many new Windows games now use DirectX for video and sound. So, if you have an MCI driver for your card, does that mean that you have DirectSound support, and that a program using such will work with your card? Well, yes and no. To take advantage of DirectX, you need an MCI driver for your card which also has some extra support for DirectSound/ DirectMusic in it. If your sound card driver doesn't directly support DirectSound/DirectMusic, Windows can "manipulate" things such that a program using DirectSound/DirectMusic will be able to use your sound card, but there may be speed penalties involved, or some sound features may not work very well. Or the sound may even not work at all if the driver is really questionable. In conclusion, you really do want DirectSound and/or DirectMusic support in your card's driver if you plan to use programs that use DirectX (ie, mostly new Windows games at this point, but audio/MIDI software using DirectSound/DirectMusic may likely be on its way. You may be especially interested in DirectX support if you're using a software synthesizer -- a program running on your computer which emulates a MIDI synth. Software synths tend to work best with programs and drivers that directly support DirectX. And some of the audio programs that use "DirectX plug-ins" to add new features to the program may also benefit from drivers that support DirectX).

But by far the greatest source of problems will be related to Windows XP. Why? Windows XP is a very different operating system than Windows 3.1, Windows 95, Windows 98, and Windows Millennium (ME). Those latter 4 operating systems can all use the same driver. **But a driver written for any of those 4 will <u>not</u> work for Windows XP**. Windows XP is based upon the operating systems of Windows NT and Windows 2000, and it uses a completely different type of Windows driver than does Windows 3.1/95/98/ME. Now it is true that a driver written for Windows NT or Windows 2000 will very likely work for Windows XP. (So, if you can't find an XP driver for your hardware, look for NT or 2000 drivers). But, many sound cards do <u>not</u> ship with a Windows driver that supports NT, 2000, and XP. Those cards ship with a driver that supports only Windows 3.1, 95, 98, and ME. Windows NT/2000/XP drivers are more complicated to write, and a lot of programmers do not know how to write them (whereas 3.1/95/98/ME drivers are "old hat"). So, many music companies have very spotty NT/2000/XP driver support, if any support at all. Before you buy some audio/MIDI hardware to use with XP, make sure that you can get driver support for XP (or at least NT/2000).

For all of the above reasons, I want to stress that it's important to choose a sound card from a manufacturer who is good about providing updated drivers. You may be able to save a few bucks by buying a cheap sound card from "Joe's Sound Card Company", but if you can't get uptodate drivers

for it, it may never function as well as a card with good driver support. Years back, I used to have to tell people that they were stuck buying Creative Labs cards due to hardware/software compatibility issues. Now that Windows MCI has made the hardware issues irrelevant, is that no longer the case? Sadly, no. To be perfectly frank, as perhaps the largest music company making computer audio cards, Creative Labs can afford to hire numerous, decent programmers, and so CL's driver support is typically a lot better than most other companies. And today, driver support is what really matters. A case in point: when I moved to Windows 2000, I couldn't get my Roland RAP-10, or expensive Mediator MIDI interface, or Turtle Beach audio card, or some other multi-port MIDI interfaces working. (These were all considered "better" than Creative Labs cards back when I purchased them). Why? Because they never had Windows NT/2000/XP drivers. Sure, they had drivers for 3.1/95/98/ ME, but remember that those drivers will not work for NT/2000/XP. And the manufacturers never wanted to spend the money hiring developers to provide updated support for hardware that had been around a few years and was no longer selling (or selling very much). But what did work on Windows 2000? -- a Sound Blaster 16, because Creative Labs did release updated drivers for it. And so history repeats itself.

Now, a lot of companies like to "promise" future support, but you should take this with a big grain of salt. For example, one company rep I spoke with mentioned that they expected to have NT drivers available for their product within "a couple months". It is now 3 years since that conversation and they still have no such NT drivers (even though their web site still actually claims "Drivers for WinNT are being worked on". I don't think so). My rule of thumb is that, if appropriate drivers are not available within one month of release of the hardware, the odds are unlikely that you're ever going to see the driver support you want. It is unlikely that a competent, "inhouse" programmer would be more than one month behind the actual public release of the hardware, so the company has either hired someone questionable, or they've "farmed out" the work to an outside contractor. In either case, my experience is that such a situation forebodes a questionable commitment to driver updates. So don't hold out on any promise of future driver support longer than one month. (And wait a month before you buy if you're waiting on some "promise").

So, the important questions to ask about driver support for audio hardware is:

1. If you're using Windows NT, 2000, or XP (or plan to soon) -- "Does this card ship with a driver that works under Windows XP?". (If you want to get really technical, you can ask if it ships with a Windows NT/2000/XP kernel mode driver or miniport driver. If the person doesn't know what you're talking about, then you may be talking to someone who can't give you a definitive answer as to whether the card really does have drivers that work under XP). If the answer is no, then the card won't work at all under NT/2000/XP.
2. "Does the driver for this card support the Windows mixer API?" (pronounced "aye-pee-eye", like the individual letters). If the answer is no, then you can't use the standard Windows Volume control (in the task bar) or mixer software with it, and probably can't control such settings from other software. You had better hope that the card at least ships with some special software that lets you do things like adjust the volumes of various inputs/outputs, etc.
3. "Is the driver for this card multi-client?" If the answer is no, then you can run only one program at a time which uses the card.
4. "If this card does digital audio playback/recording, does its driver implement DirectSound functionality? If this card does MIDI playback/recording, does its driver implement

DirectMusic functionality?". If the answer is no, then you'll find that the card doesn't work with game software that requires DirectX sound support, or works with very limited sound. And if music software arrives with DirectX support, you may also wish that your driver supported DirectX too.

Different cards for different needs

Now, this is not to say that, just because MCI allows a Windows program to perform MIDI and/or digital audio with most all cards on the market nowadays, you'll get the same performance from all. As mentioned, a game card can be used to record a digital audio track with Cakewalk, just like a high-end card can. But the game card will typically create a track with a lot more audible "hiss" in it, and maybe roll off more of the high and low frequencies. A cheap card may have only a 16-bit DAC (and ADC), rather than 24-bit, and so you may hear more "distortion" as you mix numerous audio tracks in realtime (for example, if you take a program that can record numerous, separate "tracks" of digital audio, such as Cakewalk or Cool Edit Pro, and then play all of them back simultaneously. Also, software synths, and/or DirectX software that does realtime wave mixing, can benefit from 24-bit, rather than 16-bit, DACs. Essentially, any time you're mixing more than 2 digital audio waveforms in realtime, you can benefit from higher resolution than 16-bit DACs. But playing back only stereo, 2 track audio shouldn't make any audible difference between 16-bit and 24-bit hardware).

Furthermore, you should be aware that some cards don't fully support (in hardware) all that is possible under Windows MCI operation. For example, some cards don't offer *full duplex* operation. Full duplex means that when you're recording a digital audio track, you can simultaneously hear previously recorded digital audio tracks being played back. Although you can use CakeWalk with a half duplex card, you won't hear previously recorded digital audio tracks when recording a new track, even though that's what CakeWalk <u>could</u> do with a full duplex card under Windows MCI. (But most all cards nowadays are full duplex). As another example, Digital Audio Labs CardD only supports digital audio playback and recording. It doesn't handle MIDI at all. CakeWalk could use a CardD to play/record digital audio tracks. But CakeWalk can't play/record MIDI tracks on a CardD (ie, you'd have to use a separate card for MIDI, or use a "Software Synth" as described below).

In other words, although MCI allows Windows programs to support a wide variety of cards, not all cards are created equal, and therefore there still are performance differences and limitations that you have to consider.

Finally, it should be noted that not everyone wants the same thing. For example, if you find the sounds and flexibility (ie, controller routing, patch programming, etc.) of the internal sound modules built into computer sound cards too limited, and instead prefer external MIDI modules, then you'll probably want a card without a built-in sound module, and a solid MIDI interface built in. On the other hand, if you play games, you'll very likely want a card with digital audio playback, and may not care about a MIDI interface at all.

In conclusion, you have to know what you want to do with audio/MIDI, and what kind of performance you require, before you can pick the card for you. If you just want a card to use with game software, then it's very likely that the "best" card for you is different than the "best" card for

someone who wants to master a CD of his own compositions as played back on his audio card. And if you're a guy who wants to do both, then you may have to decide whether to invest in 2 cards, or find the best single-card compromise. Finally, since driver support is so important to the ultimate performance of a card under modern operating systems, you'll want to buy from a company that offers good, periodic driver updates.

MIDI Interfaces

If you're going to do any MIDI work, you definitely need hardware that supports MIDI input and output. I refer to hardware that deals only with the transfer of MIDI data between a computer and some external MIDI units as a "MIDI Interface". (Yeah, it's sort a misnomer since the last "I" in "MIDI" itself means "Interface"). MIDI interfaces usually have the MIDI IN and OUT DIN jacks right on the hardware itself. Some MIDI Interfaces take the form of a card that plugs into an ISA or PCI slot inside of your computer. Others take the form of an external box that attaches to the computer's serial (COM) or parallel (printer) or USB port. A MIDI Interface does not produce sound by itself. It needs some other, external MIDI sound module connected to it's MIDI IN and OUT jacks.

Because MIDI Interfaces are designed for only MIDI work, sometimes they have special features that appeal to professional musicians who need optimum MIDI performance. For example, some interfaces have multiple MIDI busses to allow MIDI data to be input/output more efficiently to many external devices. (See the article Multiple MIDI outputs for more information about such). Some interfaces offer SMPTE (and other forms of) synchronization. Some interfaces have very fast hardware-buffering of input/output (moreso than cards that aren't dedicated to handling MIDI). Without such buffering, MIDI playback can really hamper a fast computer, and cause it to perhaps slow down when it tries to simultaneously do other things. The absense of such buffering may even result in MIDI playback with weird timing fluctuations (ie, the rhythm of the music sounds "off").

Examples of MIDI Interfaces in ISA card format are Roland's MPU-401 (or MPU-IPC), the SuperMPU, MusicQuest MQX-16 and MQX-32, and certain Voyetra cards. Of these, the MQX-32 and SuperMPU each have 2 MIDI busses, as well as SMPTE sync. These are all older cards that are either out of production or not supported. There have not really been many new MIDI interfaces in the form of an ISA/PCI card. Most MIDI interfaces sold nowadays are external boxes that attach to the serial, parallel, or USB port, with USB being the most popular and best supported now.

Note: The original MusicQuest MQX-32 offered two independent MIDI busses, but the second bus (as well as SMPTE support) was disabled in Uart mode, and that's the mode that most all software runs an MPU-401 compatible in, including Windows software and MPU-401 drivers. MusicQuest also released the MQX-32M (which had 2 independent MIDI inputs) in 1989. A few months later in April 1989, MusicQuest, realizing that most software was running MPU-401 compatibles in Uart mode, redesigned the MQX-32 and MQX-32M to allow the second MIDI bus and SMPTE features to be used in Uart mode. I'm not sure of the firmware version that offered these changes, but the important thing to note is that only the redesigned MQX-32 offers multiple busses and SMPTE under Windows.

Examples of MIDI Interfaces that attach to the parallel port are Midiman's Portman series (comes in

4, 2, or 1 buss) and Mediator's MP-128S (8 MIDI busses, SMPTE). Some serial interfaces are Mediator's MS-124 (multiple busses) or MS-101. Examples of USB interfaces are Edirol's UM-880 (8 MIDI busses, IN and OUT), UM-2 (2 MIDI busses), UM-1S (1 MIDI bus), and MidiMan's MidiSport series (comes in 8, 4, 2, or 1 MIDI buss). Parallel, serial, and especially USB port interfaces are very useful for laptops running Windows. Some of the USB port interfaces are quite nice. They have multiple MIDI outputs (ie, busses) and even SMPTE sync. I'd avoid the serial port interfaces unless you know that your computer has at least a 16550 chip in it. Less capable chips won't be able to keep up with the MIDI baud rate required by the interface. And forget about serial port interfaces with multiple outputs. There's still a single serial buss (from your computer) feeding the interface. And my experience with parallel port interfaces is that they can sometimes not work with notebooks (although they seem to do fine with most desktop PC parallel ports). Most companies seem to have moved to making USB interfaces, and so the parallel and serial interfaces may not come with driver support for XP (whereas a lot of the USB interfaces do support NT/2000/XP. Edirol seems to have pretty good driver support, as do Midiman). USB is the way to go now, and can work with both PCs and Macs (given driver support).

Some external MIDI modules have a connector to which the module can be directly connected to a computer serial (ie, COM) port. So you don't need a computer MIDI interface at all for such a module. Besides eliminating the need for a MIDI interface, the module typically also has MIDI jacks to which you can attach more external MIDI units which can then pass MIDI data to and from the computer via this module directly attached to the computer. (ie, A MIDI keyboard controller attached to the module's MIDI In sends its MIDI messages along to the computer's COM port for input into the computer. Other sound modules daisy-chained to the module's MIDI out receive MIDI messages going out of the COM port). In other words, such a module functions as a MIDI interface by itself. Examples of such modules are the numerous Roland SC line sold through Edirol, as well as many Roland pro units such as the JV line. These units can even be attached to a Mac serial port (with the proper cable). Furthermore, the units can be attached directly to a keyboard controller, and used without the computer. They even have a headphone jack, making them ideal for playing MIDI scores on notebooks. Of course, you do need a serial MIDI driver for your operating system. (Windows 3.1/95/98/ME drivers, as well as some Windows NT drivers which work on 2000/XP, are available for Roland modules on Edirol's web site). The Roland JV-5030 can be directly attached to a USB port and function as a MIDI interface, as can some of the new Edirol models. USB will likely become the preferred port for MIDI sound modules that attach directly to some computer port.

Digital Audio Cards

If you're going to do any digital audio work, then you need audio hardware with a Digital to Analog Converter (ie, DAC) to play digital audio, and an Analog to Digital Converter (ie, ADC) to record digital audio (ie, digitize an audio signal running into the card's Microphone or Line Input jacks). I refer to such a card as a "Digital Audio Card" or "Audio Card".

The typical (ie, most supported) use of digital audio is for game sound effects and voice (ie, sound effects and voice are usually WAVE files played on the card's DAC), and also recording/playing a finished musical mixdown (such as creating a WAVE or MP3 mixdown of a song). See the Digital Audio on a computer FAQ for further information.

Most digital audio hardware for a computer takes the form of a PCI card (that plugs inside of your computer and has jacks on the back of your computer for speaker output, microphone input, etc). But some digital audio hardware comes in external packages such as Roland's Audio Canvas UA-100 which connects to a USB bus. (Parallel and serial ports are too slow for digital audio transfer in realtime, so products are not available for those ports).

Most all sound cards on the market today offer digital audio playback/record. Examples of cards that offer digital audio playback/record are Event's Darla, Gina, or Layla cards, Digital Audio Labs CardDeluxe or V8, Aardvark Direct Pro 24/96, EMagic Audiowerk8, etc, as well as "game cards" like the SBLive or Audigy, or Santa Cruz, or most anything that you'll see on the shelves of your typical computer store. The more professional cards may offer more than 2 discrete digital audio tracks. (But note that most programs, such as CakeWalk, offer "virtual digital audio tracks" in order to support more than 2 tracks of digital audio on cards with only 2 tracks, so extra digital audio tracks aren't necessary and aren't useful unless supported by software. They prove most useful when using software that can access multiple "devices" simultaneously such as Cakewalk Pro Audio. Then you can use a unique hardware audio "channel" for each sequencer track, and have Cakewalk not need to do any software mixing in realtime. In other words, Cakewalk doesn't need to implement virtual tracks because you have enough actual hardware tracks for all of your sequencer tracks. The hardware does the mixing. And that can help reduce any "artifacts" such as distortion upon playback). The Event, DAL, and Aardvark units are geared for serious semi-pro use (although all of the above non-gamer cards are good digital audio performers), whereas the game cards typically have lower signal-to-noise ratios (ie, produce more "hiss" upon recording) or poorer frequency response (ie, can roll off the lower and higher frequencies more).

There are some high-end ($$$) digital recording packages that feature both hardware and software (typically Windows 3.1/95/98/ME programs). Typically, these systems offer multiple digital audio tracks (usually discrete, with multiple inputs/outputs, digital I/O, SMPTE sync, etc) and features that make them full-fledged production studios comparable to using a standalone digital audio machine such as an ADAT. Such packages are CreamWare's TripleDat, DigiDesign's Session (and earlier Session 8), Spectral's Prisma, Studio Audio & Video's SADIE, Micro Technology Unlimited's Microsound Krystal, and Merging Technology's Pyramix. But for most people, I think that you'd be much better off using off-the-shelf software such as Cakewalk with a good digital audio card and MIDI interface.

Lately, many cards have been offering Digital I/O, meaning that the output of the card's DAC can be run directly to another digital recording medium such as a DAT recorder. In this way, you bypass the card's analog audio output stage (which isn't needed if you're going to create your master mix on a DAT deck anyway), and eliminate any extra noise from that section. Also, transferring digital audio between your computer and other devices that support digital I/O is quicker and results in bypassing the card's analog input stage which can also introduce noise. Most all of the serious audio cards have Digital I/O. Even some of the cheap cards, such as the SBLive or Audigy or Santa Cruz, have digital I/O, but sometimes they don't always support the DAT bit rate as well as the more professional cards do. For example, a cheap card like the Audigy may do something called "resampling" which can introduce distortion. On some cards, the digital I/O is an add-on board that you must buy. Some manufacturers, such as Zefiro, make relatively inexpensive, digital I/O only cards (ie, without the

analog audio stage -- ie, you can't actually hear the tracks playing until you hook up some analog stage to them). This is useful if you already have another card that can play digital audio. You use that card just to monitor the audio tracks. But you use the Zefiro to record the tracks, and then play them during the final DAT mixdown. There is also a digital I/O only version of the CardD.

Digital Audio and MIDI support in one card

Most sound cards have both digital audio support, as well as MIDI support. The card may have a built-in hardware MIDI sound module which will play any MIDI data that a program sends to the card. This is akin to attaching one external MIDI sound module to a MIDI Interface card. Such a card actually can produce MIDI generated sound all by itself, unlike a MIDI Interface. So, the one card offers a "complete" audio/MIDI system capable of being used by itself to produce a musical project.

Most all built-in modules nowadays support the General MIDI standard. This means that the module is multi-timbral and can therefore play entire, complex MIDI arrangements. (See the article entitled What's Multitimbral? for more details). Some cards even have built-in hardware effects such as digital reverb, delay, and chorus to enhance the sounds on the card. (Ideally, you want these effects to also be applicable to the digital audio playback, as well as the built-in MIDI sound module. But on some cards, this may not be the case).

One advantage to having such a sound card as opposed to just a MIDI Interface, is that you can easily use the sound card to compose and score music without needing to run MIDI and audio cables to external MIDI sound modules and effects devices, and twiddling knobs on a mixer. You just run the stereo outputs of the card to a tape/DAT deck, and monitor with amp/speakers or headphones (since most cards have a headphone jack too). Many Windows sequencer programs such as CakeWalk offer a software "mixer" that can adjust the panning, volume, reverb level, etc, of each playing "instrument" via MIDI messages that are (hopefully) understood by the card's driver. So, you can setup a finished, stereo "mix" without moving from your computer.

NOTE: Many "game cards" have a poor MIDI implementation that doesn't support MIDI control over certain sound parameters. For example, maybe a better sound card lets you add vibrato to an instrument using MOD Wheel MIDI messages, whereas a cheap card won't do anything with that message.

Some cards also allow you to replace the waveforms used in the GM sound module. You load some WAVE file into RAM on the card itself, set loop points, and can play the wave back polyphonically as an "instrument". In other words, these cards are starting to be "MIDI samplers" (albeit without the bells and whistles of fancy VCAs, VCFs, multi-sampling, etc). Of course, you need a software program in order to choose which waves to load and how to play them back (ie, map them out to particular MIDI note numbers if supported, set looping, etc). Support software for such, which often is released with important features not yet implemented, makes this one of those features that you may want to wait for it to mature. (Such software also tends to be written for Windows 3.1/95/98/ ME, but not Windows NT/2000/XP which are much stricter about allowing access to hardware by a program). Many cards offer the ability to install lots of RAM on the card itself which can translate to loading lots of waveforms and creating very rich-sounding patches or unusual custom patchsets

which sound totally unlike the original patch set in ROM. Companies are releasing patchsets that are fairly extensive and nice-sounding, which you can further modify, such as Creative Labs' 8 MEG EMU Patchset. Of course, uploading that much data to the card does take a little while (less than a minute) but only has to be done once whenever you turn on your computer. If you do get a card with this feature, you'll want one that supports "downloadable soundfonts" (ie, DLS), preferably version 2.0 of the specification.

Most sound cards with MIDI support also contain a MIDI Interface (for external MIDI modules) too, because after all, the internal sound module is usually controlled by MIDI messages anyway, and it's not much more circuitry to allow the MIDI messages to be sent on out to external modules. On the other hand, most sound cards are designed more with MultiMedia use in mind rather than MIDI. For this reason, they don't have actual MIDI IN and OUT DIN jacks on the card itself. Instead, the cards have a joystick port into which you connect a special box that contains the actual MIDI IN and OUT DIN jacks. (The box should also contain an optoisolator on the MIDI input). Most companies refer to this box as a "MIDI Adapter Cable" (or sometimes a "MIDI Adapter", or even by the inaccurate designation of "MIDI Cable" which invites confusion with a simple MIDI cable that connects any two MIDI devices together). Since the Sound Blaster was the most common card with this type of "connect the MIDI Adapter to the joystick port" arrangement, most manufacturers have adopted the same pinout (15-pin) as the SB for their joystick ports. In this way, any MIDI Adapter Cable for the SB will work with any other Sound Card.

And although most MIDI Adapters are made to work with any SB compatible joystick port (also called a "gameport"), that doesn't mean that they're all alike. Some adapters have an actual box with surface-mounted female DIN jacks, including even a MIDI THRU jack. You then connect MIDI cords between the Adapter and your MIDI module. The box may even have its own joystick connector so that after you plug the adapter into the card's joystick port, you can then plug a joystick into the adapter's joystick connector. Thus, you don't have to disconnect the adapter every time that you want to use a joystick. (You can use the joystick at the same time that MIDI is going through the card since the joystick uses different pins than the MIDI in/out). An example of such an adapter is Mediator's JAM52/M. Other, cheaper adapters simply chop a MIDI cable in half, solder the bare ends to a connector that plugs into the card's joystick port, and you plug the dangling male DIN connectors directly into your MIDI module's MIDI jacks. Obviously such an arrangement isn't as versatile (nor dependable since cables tend to break) as the box with the surface-mounted jacks. An example of these cheaper adapters is Mediator's MG-2. Many other companies make these MIDI adapters.

There is one thing to know if you're looking for a sound card that offers MIDI support (in addition to digital audio). Many of the new cards don't have a hardware MIDI sound module on the card itself. Rather, the card uses a "Software Synth" for MIDI playback. To cut costs, the card may use the "Microsoft GS Wavetable SW Synth" that is included with the Windows operating system. (The "SW" stands for "Software"). What is a Software Synth? It is part of the card's (software) driver. A MIDI program such as Cakewalk sends the driver a MIDI message to play a C note with a Piano sound, for example, and the driver translates that one MIDI message into a stream of digital audio values which represent the sound of a piano playing a C note. The driver likewise translates other MIDI messages as it receives them. It uses its own set of digital audio waveforms for its built-in "patches" and mathematically mixes them into a stereo, digital audio "mixdown". This digital audio mixdown is played on the card's DAC (ie, digital audio section) immediately (ie, while the driver is

doing the translation). In effect, a Software Synth translates MIDI to WAVE, in realtime (ie, while receiving the MIDI messages). This is an inefficient approach toward MIDI playback that is much more CPU intensive for your computer (than use of a real hardware MIDI sound module built into the card). And since the software synth typically needs to load its "soundfont" of waveforms into your computer's system RAM, it eats up your RAM. You'll likely find that, if you need to do any work with MIDI, you'll have to supplement such a sound card with a MIDI interface. One telltale sign that a card's built-in MIDI support is limited to only a Software Synth is if it doesn't support MIDI recording (but rather, only playback).

There are many cards on the market that support both digital audio and MIDI. A few of them are the Creative Labs SBLive (older) or Audigy (newer), Turtle Beach Santa Cruz (sometimes branded as a "Videologic SoundFury"), Acoustic Edge, Lynx Studio Technology LynxONE, the Terratec DMX 6fire LT, etc.

Audio on the Motherboard

Some computers have audio chips directly soldered onto the motherboard, and therefore don't need a separate sound card. In particular, notebook computers usually have motherboard audio because ISA/PCI cards cannot be put into a notebook, and notebooks are cramped for space.

These chips typically offer basic, digital audio recording/playback of the quality of a typical "game card". Besides having a DAC and ADC so that you can record and play digital audio (and play game sound effects and voice), these chips also tend to have at least support for a Software Synth for MIDI playback. A good chip should even support MIDI IN and OUT through a MIDI Adapter attached to a joystick port. But in notebooks without a joystick port, you typically have to get an external MIDI Interface for MIDI support beyond a Software Synth.

I've never seen motherboard audio using a chip that offers much better specs than your typical game card. So, for people looking for higher quality (ie, musicians), motherboard audio is usually to be avoided.

Personally, I don't like motherboard audio because you can't upgrade it easily. If you want to buy and install a new audio card, you can't remove that chip. Therefore you have to setup your new card such that it doesn't conflict with the settings of that onboard chip. Typically, the computer's BIOS has settings to disable the onboard chip, but besides being a waste (ie, at least you can sell an old audio card when you buy a new one, or reuse it in another computer -- not so with these embedded chips), I've discovered that PnP issues and other BIOS bugs can result in problems. Sometimes you can't get that onboard chip to stop using valuable resources (ie, IRQ lines) that you want for your new card, or otherwise can't make it completely "disappear". It lingers like body odor, causing you grief.

I think that an audio card is a better deal than motherboard audio, unless you're absolutely sure that you want a typical game card quality and don't ever want to upgrade for the lifetime of that motherboard. An exception to this is a notebook computer since you can't install a separate ISA or PCI soundcard, and notebooks aren't noted for their upgradability anyway.

What's best for me?

As mentioned, that depends upon what you want to do with audio/MIDI, and what kind of performance you require. Let's examine a few test cases:

I want only a MIDI Interface. I'm going to use only external MIDI sound modules (ie, I like the sound and features of dedicated MIDI modules better than the internal sound modules on Sound Cards). I want to use Windows MIDI programs such as CakeWalk. I'm not going to need digital audio tracks (ie, I'm dealing only with MIDI music or using a dedicated external digital audio unit such as an ADAT). I don't care about game audio support.

A very good choice are the USB port devices with multiple MIDI busses and SMPTE. See the listing of USB interfaces. They offer lots of flexibility and can handle a large, professional MIDI system that needs to sync to other equipment well.

Note that some external MIDI modules also have a connector allowing you to attach the device directly to the computer's serial (COM) port, and more recently, the USB port. Examples are most of Roland's Sound Canvas and JV lines. These external units have MIDI IN and OUT jacks to which you can attach further units. So, in effect, they're like serial port MIDI Interfaces with a built-in GM module. See serial (COM) port modules.

Like the guy above, I'm going to use only external MIDI sound modules with Windows MIDI programs such as CakeWalk. But I want to record digital audio tracks too. I also don't care about game audio support.

If you want top performance, you could buy 2 items; an audio card that is made especially for clean digital audio such as the DAL CardD or V8, Aardvark Direct Pro 24/96, the Event Gina, Layla, or Darla, or other high-end digital audio cards, and a MIDI Interface as mentioned in the preceding answer. Configure your software (and Windows) to use the former for digital audio and the latter for MIDI.

Or, you can buy a sound card that has decent digital audio (although not quite as good as the above digital audio cards) plus at least a built-in MIDI interface for both output and input, such as the Turtle Beach Santa Cruz. This is actually a fairly good all-around card. Even the SB Audigy isn't bad, except for if you want to use its digital I/O with a DAT machine.

Stay away from the no-name sound cards as typically sold in large computer retail chains. They typically use the sort of chip that you'd find used for motherboard audio.

I'm interested in a card only for games, and I want something that's compatible with all games.

If you're a game player, it's mostly all about drivers, specifically DirectX support. You definitely want a card that supports DirectSound, and supports it well. If you want to play it safe, buy a Creative Labs Audigy. If you do, you probably won't be left high and dry without any driver support. Don't buy "Joe's Sound Card" unless there's a web site where you can download driver updates for it, or you're sure it comes with the DirectSound drivers (for your version of the Windows operating system -- XP needs an NT/2000/XP driver) you need in the box. You also want to pay attention to the details of the driver's DirectSound support -- whether it supports "3-D sound modeling" and various effects algorithms, and especially how well it supports realtime mixing of "channels" of audio.

Whether a card uses a Software Synth for MIDI playback is pretty much irrelevant to you, since most games nowadays do not use MIDI at all. (For game music, they use CD audio tracks, which is digital audio. And of course, sound effects and voice are digital audio too).

What may be more important to you is whether the card supports "surround sound" (ie, has 4 independent speaker outputs for "front" and "back" speakers).

I'm interested in a card for games which will work well with all of my game programs. But, I also want to use it for music projects, so it has to have high quality digital audio and MIDI performance, and a great sounding internal GM module. In other words, I want something that sounds better than Creative Labs' line, but is every bit as supported by game programs.

Now you've got a problem. You need a card that is designed for both gamers and musicians, and these are very different markets. You'll likely have to make compromises (ie, maybe give up surround sound for gaming, or accept worse S/N ratios and frequency response for music work).

I have a notebook computer.

Since you can't put ISA/PCI cards into notebooks, a lot of the aforementioned cards are not useful here. As mentioned, a USB port MIDI Interface is one solution for MIDI recording/playback. Of course, you then need an external MIDI module. See the listing of USB interfaces.

External MIDI units that directly attach to a [serial (COM) port](#) or USB port are quite useful here, as they eliminate the extra box (and are hence more portable).

If you want to improve upon the digital audio specs of the motherboard audio on your notebook, you may want to look into the Roland Audio Canvas UA-100, or if you want both digital audio and MIDI, check out some of the new USB Edirol models that support both digital audio recording/playback and MIDI.

I need a card that works under Windows XP.

Do not buy anything that doesn't specifically come with a Windows XP driver, or at least has a Windows NT or Windows 2000 driver for it (which <u>should</u> work). Windows 95, 98, SE, ME, or Windows 3.1 drivers will <u>not</u> work with Windows XP.

This article answers questions about connecting MIDI gear (particularly with regard to a MIDI setup that includes a computer), and setting up computer MIDI hardware/software.

When it comes to solving problems with an entire setup (ie, more than one, self-contained piece of equipment), you need to learn how to do something called "trouble-shooting". What this involves is simply going through your entire setup, one item at time, <u>isolating</u> that individual piece of equipment and checking that it is operating as you expect.

If you suspect that a particular item is the source of your problems, try to remove just that one item from your setup. Replace it with a suitable, substitute item (or nothing at all if your system can operate without that one item). If your problems disappear, then you've found the "bad" item, and can concentrate upon trying to "fix" this one item. Every single detachable item should be regarded as a separate item to be checked, including <u>all</u> cords and cables, power strips, and even each program you're using on your computer.

Don't go running around the room, checking items at random. Start with one item at the end of your MIDI daisy-chain and follow the MIDI connections through to the last item. As you move from item to item, remember to check the MIDI cable connecting the 2 items. Replace it with another MIDI cable and see if that makes any difference. Look for obvious mistakes on each item such as forgetting to turn the power on, forgetting to turn the volume up, connecting a MIDI OUT jack to another MIDI OUT jack or MIDI IN jack to another MIDI IN jack (MIDI OUT or THRU jacks always connect to MIDI IN jacks, and vice versa), forgetting to make an essential connection (such as the power cord or audio cable), etc. Check for loose connections. Plug a pair of headphones directly into the output of a sound module if you suspect a problem with your mixer. Play the sound module from its own local keyboard if you suspect a problem with the MIDI output of your sequencer.

When it comes to problems involving computer sound cards or MIDI interfaces, or getting computer software working with sound cards or MIDI interfaces, problems can even be more convoluted. If you're having a problem with an internal IBM PC card not working, then you should first check for any hardware conflicts in your system. Read the article ["Resolving hardware conflicts"](#) for more information.

Another, even more common problem concerns software drivers. The fact of the matter is that programmers <u>do</u> write buggy software, and there's a chance that any problem which looks hardware-related may actually be due to some bug in the card's driver. Check with the card's manufacturer that you have the latest drivers. Ask if there have been any problems reported that may be applicable to your own setup.

And definitely don't rule out the possibility that you may have configured the driver's setup (ie, "Properties" in Microsoft-speak) incorrectly. If you use Windows 95, read the FAQ ["MIDI/Audio under Win95"](#).

Questions in this FAQ are:

Why do MIDI IN jacks connect to MIDI OUT jacks? It makes more sense to connect a MIDI IN jack to a MIDI IN jack. After all, they're both labeled IN. Same goes for MIDI OUT to MIDI OUT.

The accepted way actually makes a lot of sense. Think about it. You want MIDI data to go <u>out</u> of your controller and <u>in</u> to your sound module. After all, you wouldn't connect the audio out jack of your sound module to the outputs of your mixer, would you? No, you connect the audio output to an audio (mixer) input. And then you connect the mixer outputs to the inputs of your amplifier. And then you connect the amp's speaker outputs to the speaker inputs. Same thing with MIDI. Think of MIDI data as "flowing" in the same way that audio signals "flow" through your audio system.

I connected the MIDI OUT of my keyboard controller to the MIDI IN of my computer sound card. The card has a built-in sound module. So why am I not hearing any sound from the

card when I play the controller?

The built-in sound module on a typical computer card is not internally connected to the card's MIDI IN. Neither is the MIDI OUT jack of your sound card internally connected to its own MIDI IN. Therefore, the MIDI data (from your keyboard controller) which goes into your card's MIDI IN jack does not automatically get sent to the card's built-in module nor external modules attached to the card's MIDI OUT. (ie, The MIDI data goes into the computer OK. It just isn't simultaneously sent to MIDI OUT nor the card's built-in sound module). You need to specifically route the MIDI IN jack to the card's built-in sound module or its MIDI OUT.

The internal module is considered an entirely separate "device" from the MIDI IN or OUT jacks on the card. (The exception is with early Roland cards, or with cards that can host a "daughterboard" card such as a Roland SCD-10/SCD-15. In this case, the module is internally attached to the MIDI OUT, and is not a separate device). So, if you plug an external controller into the MIDI IN of your card, you won't automatically be able to play the built-in sound module on your card. You need to run some software program that provides a "software THRU switch" (usually referred as "MIDI Echo" in programs). In other words, the program reads the MIDI data coming into the card's MIDI IN, and immediately resends that data to the card's internal sound module or MIDI OUT. At that point, the built-in sound module "sees" the MIDI data from the controller. Yeah, it's a roundabout way of getting the sound module to see data from the card's MIDI IN, but if the sound module was connected to both the MIDI IN <u>and</u> MIDI OUT jacks, then it would get very confused if both a sequencer (sending data to to the module) and a controller (sending data to MIDI IN) were both sending MIDI data simultaneously.

Some fancier MIDI interfaces have hardware support for MIDI Thru. Usually, this is disabled by default. Typically this is turned on and off by software supplied with the card. This is a lot more efficient (ie, you hear less of a delay between pressing a controller key and hearing the sound) than a software MIDI Thru.

I put a fancy daughterboard in my Creative Labs' card, but I still hear the same old cheesy sounds as I did with the card's built-in module. Did I get ripped off?"

Yes, but prior to the daughterboard purchase.

The problem here is that you haven't reconfigured your operating system (or your MIDI software) to use the daughterboard instead of the Sound Blaster's built-in module. Hence, you're still hearing that old built-in module. The built-in module is considered to be a separate "device" apart from the daughterboard. The daughterboard is actually attached to the MIDI OUT of the card, so you need to go into your operating system's (or your MIDI software's) MIDI settings and select the Sound Blaster's "MIDI Output" as opposed to "WaveTable Synth" or "FM Synth" (or something to that effect, which refers to the card's built-in module). Different operating systems have different ways to do this. In Win95, you use the Control Panel's MultiMedia notebook, open to the MIDI page, and select the SB's MIDI Out as the "Single Instrument". (Or, you could go to Custom Configuration, and

divide up the MIDI channels between "MIDI output", ie, the daughterboard, and "WaveTable Synth", ie the built-in sound module, if you wish to use both sound sources together). Under another OS, you may have to modify some configuration file to indicate that the MIDI Out should be used as the destination for MIDI playback. If your sequencer uses Windows MCI drivers directly, make sure that you've assigned tracks to the card's "MIDI Out" rather than its "WaveTable Synth" (ie, the sequencer will no doubt have its own built-in MIDI setup screen).

How do I setup my software so that its "software mixer" patch names will match the sounds (ie, patches) on my sound module? If I select the "Fretless Bass" patch (#35) on my Yamaha PSR500 via the Yamaha's own control panel, I get the proper sound. But if I use my MIDI software's "mixer" panel to select a "Fretless Bass" patch, it switches to an accordian patch.

The problem here is that your Yamaha doesn't have a General MIDI patch set, but that's what your software "mixer panel" assumes your module has. So, when you select some "Fretless Bass" patch using your software, the software sends a MIDI Program Change to where the General MIDI "Fretless Bass" patch is located (ie, patch #36). But, your Yamaha has an accordian patch in that location instead (ie, #36). You need to have your computer remap the GM patch set to where those respective patches are really located on your Yamaha. (For example, if your Yamaha's "Fretless Bass" patch is #12, you need to tell your computer that when you select "Fretless Bass" on your software mixer, it should send a Program Change to patch 12 instead of 36. You'll need to go through all 128 GM patches, and select the correct patch number on your Yamaha for each, or the closest patches that you can find). Or, if your MIDI software supports it, you need to tell your software what are the real names of all of the patches on your sound module (in the correct order from patch #1 to the last patch), so that the software mixer will display those patch names (instead of the GM Patch set).

Let's consider the second option since that is more flexible. (ie, You aren't stuck with using only the GM patch names). Some MIDI software has its own "patch naming" features. CakeWalk has a feature whereby you can enter the name of each patch on your sound module. You list these names in the correct order (ie, from patch #1 to the last patch). For example, you can create a listing of all patches on your EMU Proteus sound module, specifying that patch #1 is called "Tuba Surprise" (or whatever), patch #2 is called "Deep Bass", etc. Then you can apply this patch set to a particular track. Now when you use the software's mixer panel, it uses the patch names you specified (rather than the GM patch set names), and will select the proper patches. Usually, the software allows you to create an individual set of patch names for each one of your sound modules, and then select any set for any given sequencer track. So track 1 can display the patch names of your Proteus (when you use the software mixer), and yet track 2 may display the patch names of your JV-90, and track 3 may display a standard GM patch set for your SCC-1 card. (ie, Your tracks can have different patch sets applied to them, which is very useful if you're using MIDI sound modules that have different patch sets, as above).

If your software doesn't support various sets of patch names (ie, you're stuck using the GM patch set names, and therefore when you select a particular patch name, it may be sending the wrong Program Change to your non-GM modules), you can have Windows do a remapping of those MIDI Program Changes at the driver stage. You'll still be stuck using the GM patch names, but at least you can reroute the program changes to respective patches on your non-GM module. (ie, If you've got a "Fretless Bass" patch on your module for example, at least you can setup your software so that when you select the GM "Fretless Bass" patch name on the software mixer, it actually switches to such a patch on your module).

Windows does this patch remapping right before the MIDI data is sent out of your computer. In this way, your software doesn't even need to know that Windows is remapping all of the Program Change events. And this remapping is effective for all of your software (that doesn't bypass Windows high level MCI API), including the Windows Media Player and any Windows multimedia titles.

For Windows 3.X, you use the MIDI Mapper. For Windows 95/98, you instead use an Instrument Definition File (IDF) made with the IDFedit utility and applied within the MultiMedia notebook's MIDI page. (Win95 IDF's also allow you to apply other translations such as transposing notes, and remapping the GM drum sounds to different keys. See my tutorial about using IDFEdit).

The drawback to letting Windows do the remapping is that you'll have to apply remapping by MIDI channel. (Win95's IDF allows remapping for the 16 channels of each MIDI card, which is more flexible than Win3.1's MIDI Mapper's limit of 16 MIDI channels total). This is less flexible than being able to apply an instrument definition to each sequencer track. With the sequencer, you can have a track outputting data on MIDI channel 1 use one patch set for a given song, and then in a different song, use another patch set for that same track outputting on channel 1. With Windows, you can apply one remapping to channel 1, and if you want to change it, you've got to hassle with MIDI Mapper or Win95's MultiMedia notebook instead of easily reconfiguring right within your sequencer software). Another drawback is that Windows remapping is limited to one bank (ie, one set of 128 patches) whereas the renaming features of many software programs allow you to specify several banks of patches and easily select them from the software mixer.

NOTE: Don't use Win3.1's MIDI Mapper's patch remapping (or Win95 IDF's) in conjunction with the built-in patch naming features of your sequencer. You don't want to remap your custom patch sets. The exception to this is if your sequencer completely bypasses MIDI Mapper (or Win95's MultiMedia MIDI setup). Such a sequencer would use the "low level" MIDI API of Windows. In this case, you can still setup the MIDI Mapper (or IDF) for the benefit of sequencers and multimedia software that uses the "high level" MCI API of Windows. But then, don't select the MIDI Mapper as the output driver for your software that has its own patch naming features.

I ran a MIDI program. It was using my sound card (or MIDI interface) just fine. Then, I wanted to run another program that used the same sound card, but I didn't want to end the first program. (ie, I wanted both programs running

simultaneously). The second program said that my device was "already in use", and wouldn't work with the sound card. What's up?

This is a limitation of your card's driver. The driver simply doesn't allow more than one program at a time to use it. You're just going to have to run only one MIDI program at a time. (Yes, it's a hassle).

Some drivers are written such that they do allow more than one program to use the driver simultaneously. (ie, The driver doesn't use "global data". It's fully re-entrant). Such a driver is referred to as "multi-client" (or "multi-instance"). If you have one for your sound card or interface, you won't see the above problem. But until Windows gets a real "MIDI Manager", you still have to be careful not to cause two programs to be simultaneously doing MIDI output (or simultaneous MIDI input). In that case, one program may mess up the output (or input) of the other. You can have both programs loaded simultaneously, but only operate one at a time, as you're switching between them. (ie, Don't hit the play button on a sequencer, leave it running, and then flip to another program and do something which causes that second program to do MIDI output simultaneously).

Can I have two sound cards (or MIDI interfaces) in my computer? If so, how can I get a software program to use them both simultaneously?

Maybe. Of course, both cards must be set to use a different IRQ #, and port address. (If they are Plug-and-Play cards, hopefully they are intelligent enough to cooperatively use different resources).

If both cards are of the same model/type, and therefore use the exact same driver file, you had better hope that the driver is written to support more than one of the cards installed. It may need to be "multi-client", as explained above.

If the cards are different, then they'll likely have different drivers. Install both drivers and then use Win95's Control Panel's MultiMedia notebook (ie, the MIDI page), to do a Custom Configuration as described in How do I install/setup an audio/MIDI driver?. (For Win3.1, you'll use the MIDI Mapper). You're still stuck with a 16 MIDI channel limitation in software (even though you really have 32 MIDI channels support in hardware). But at least you can divide up those 16 channels between the 2 cards (and that helps reduce bandwidth problems).

On the other hand, if you have software that can query and directly use all installed MIDI drivers (as opposed to software that only uses the default MultiMedia settings), then it likely will support all available MIDI inputs and outputs fully. For example, with two installed drivers, CakeWalk will recognize two MIDI ports (each with 16 channels IN and OUT).

I have a MasterTrax MIDI interface card. It has 16 MIDI OUT jacks (ie, multiple busses). Each jack supports 16 MIDI

channels (256 channels total). How do I use more than one output? My external Sound Canvas is connected to one MIDI OUT of the midi interface. If I want to use a separate drum machine, or my Korg M1 as extra sound modules, how do I get my software to route MIDI to those specific modules (ie, the various MIDI OUTs)?

The external Sound Canvas is connected to one of the 16 available MIDI OUT jacks on your interface, and since the Sound Canvas is multitimbral, it's going to hog all 16 MIDI channels (of one jack) for itself. So obviously, since you've got 15 other ports available, you'll want to attach your other gear to those ports. Attach one MIDI device per port as long as you've got enough ports. (ie, Each MIDI module attaches to a different MIDI OUT jack on your interface).

(As a side note, it is possible to tell the Sound Canvas to ignore certain MIDI channels. You do this by sending it MIDI System Exclusive messages. See the article Roland Audio Card FAQ for details. So, you could daisy-chain all your equipment to one MIDI OUT jack on the interface, and then divide up the 16 MIDI channels between all your gear by telling each module to only recognize a smaller, unique set of MIDI channels. But this is a waste of your other MIDI OUT ports).

Your real question is "how do I get my MIDI software to recognize, and direct its MIDI data to various ports on my MIDI Interface"? This is entirely a software issue. (Yeah, you can start shivering in fright now. You know that software issues pertaining to particular pieces of hardware means that we're talking about DRIVER SUPPORT, CONFIGURING YOUR DRIVER, and CONFIGURING YOUR APPLICATION SOFTWARE. Yep, this is going to be painful, especially if you're dealing with companies that make poor drivers and/or inflexible applications. Grab your ankles and grimace).

Hopefully, the Windows driver with your Interface is designed such that it tells Windows that there is more than one MIDI output. (If it doesn't, toss away the hardware. Without decent driver support, it's useless). Assuming Win95, open the Control Panel's *MultiMedia* notebook. Turn to the **MIDI** page. Look at the list of driver names under "Single Instrument". What do you see there? Is there more than one item there (ie, indicating that there is more than one MIDI output available)? For example, for your MasterTrax brand interface, maybe you'll see several items called something like "MasterTrax Output 1", "MasterTrax Output 2", etc. (I'm making up these names. The names are determined by what strings were in the .INF file that shipped with your driver, which you used when you installed the driver. This INF file is just a regular text file that you can read in a text editor. Think of it as a CONFIG.SYS for the sound card alone). If you see multiple items, you're cooking with gas. Your driver has successfully "installed" several "MIDI outputs (ie, devices)" with Windows. For more information on installing and setting up Win95 drivers, see "How do I install/setup an audio/MIDI driver?".

Now, you need to use Windows software that queries and can access all of the MIDI devices installed on your system. Typically, the software will allow you to choose which MIDI data goes to which MIDI output, and which input supplies incoming MIDI. For example, CakeWalk (ie, its *Options ->*

MIDI Devices menu item) will present a list of all installed MIDI devices, and you can choose which ones you want the software to access. Then, you can route each CakeWalk track to whichever output (of the ones you enabled) you want that track to be sent.

So what if your software doesn't query and use all installed MIDI devices, nor allow you to somehow route the MIDI data between those outputs? Well, that means that the software is written to only use one MIDI output at a time. Which MIDI output is that? Well, go back to the Control Panel **MultiMedia** "MIDI" page. Did you select "Single Instrument" or "Custom Configuration"? If you selected "Single Instrument", then the output which is used is the one whose name appears in the box immediately below "Single Instrument". For example, maybe you've selected the output "MasterTrax Output 1" which is the first MIDI OUT jack on the interface. Maybe you've connected your Sound Canvas to that jack. The result is that all MIDI data (output by your software) will be sent to the Sound Canvas. If you'd like your software to use another output, scroll through the list of outputs below, and select the desired one. You have to do this every time that you want the software to use a different output, and the software can only use 1 output at any given time.

If you selected "Custom Configuration", this allows you to divide up 16 MIDI channels among several outputs. Never mind that your MIDI interface is capable of handling 256 channels among its 16 outputs. You're going to have to stick to a channel limitation of 16, and divide those up between available outputs. For example, maybe for MIDI channels 1 to 5, you'll select the output "MasterTrax Output 1" which presumably is the first MIDI OUT jack on the interface. Maybe you've connected your Sound Canvas to that jack. The result is that all MIDI data (output by your software) on channels 1 to 5 will be sent to the Sound Canvas. Maybe for MIDI channels 6 to 9 and 11 to 16, you'll select the output "MasterTrax Output 2" which is the second MIDI OUT jack on the interface. Maybe you've connected your Korg M1 to that jack. The result is that all MIDI data (output by your software) on channels 6 to 9 and 11 to 16 will be sent to the M1. Maybe for MIDI channel 10, you'll select the output "MasterTrax Output 3" which is the third MIDI OUT jack on the interface. Maybe you've connected your drum box to that jack. The result is that all MIDI data (output by your software) on channel 10 will be sent to the drum box. Win95's MIDI "Custom Configuration" setup is basically Win3.1's MIDI Manager (without the "Patch remapping" feature -- this feature is now assumed by Win95's Instrument Definition Files, or IDF's). With Custom Configuration, you're still limited to 16 channels (as you would be by daisy-chaining all of your gear to only one MIDI OUT jack on your interface), but at least you're using more than one MIDI OUT jack on your interface (which helps to relieve some problems with MIDI bandwidth).

But of course, the best solution is using software that recognizes and uses the multiple MIDI devices (outputs) on your interface.

I have a PC with Win3.1, Windows sequencing software, a controller keyboard, and a MIDI interface for my computer.

Whenever I try to record MIDI (from my controller), some (but not all) data gets lost randomly. I've tried various controllers, and various sequencer programs to no avail. I did not use

MIDI mapper.

MIDI playback works perfectly.

Is this a hardware problem with my MIDI interface?

I suspect that you're right as to hardware deficiencies being part of your problem, but I also think that your real problem may be due to something called "interrupt latency" (which is sort of dependent upon both software and hardware). In a nutshell, all that means is that your computer isn't running the sound card driver's interrupt handler often enough that the driver is grabbing all of the MIDI data from the card's MIDI IN port. That data is only available for a limited time on the card's MIDI IN port, and if the driver code ISN'T run (by your computer's CPU) in time to grab that byte and pass it off to your software, then the data byte is lost forever when the next incoming MIDI data byte arrives.

Solutions (to be tried in the following order):

1. Make sure that you have the latest drivers for your MIDI interface. OK, this isn't directly related to the problem of "interrupt latency", but it's worth checking out first because it's a lot easier to replace a driver than to screw around with hardware.
2. Make sure that you use a MIDI interface that has a sufficiently large hardware buffer on its MIDI input. It should be able to store upto at least 64 incoming MIDI bytes that are waiting to be read by the driver. In this way, the driver can take its time reading the next MIDI data byte without worrying that another incoming MIDI data byte will "overwrite" the preceding data byte. All Roland cards have buffered MIDI inputs, some with over 1000 bytes in order to better handle large System Exclusive messages. Many other "high-end" cards likely have good hardware-buffering too, but I'm mostly familiar with Roland stuff.

   It sounds like your MIDI interface has no, or anemic, hardware buffering on its MIDI input.
3. If the above doesn't cure the problem, then try to reduce the load on your CPU so that it can devote more time to running the sound card driver's interrupt handler. In particular, remove anything extra that is interrupt-driven. Turn off any "software blinkie lights" (if possible) and other "animated" graphics that your sequencer may be doing while it's recording. (ie, You don't want the sequencer to be using CPU cycles drawing some scrolling display during MIDI recording). Don't move the mouse during recording. If you're trying to do digital audio playback while recording MIDI data, be aware that digital audio playback/recording is extremely I/O intensive (and perhaps CPU intensive if your card doesn't use DMA to stream audio data to/from the card's Digital To Analog Converter or Analog To Digital Converter, as some cheap cards may not). Don't play/record digital audio tracks during MIDI recording if your MIDI recording is getting screwed up. Reduce your video display (ie, less colors and/or less resolution) to make sure that the CPU is not spending too much time drawing graphics.
4. If the above doesn't work, then get a faster computer system (ie, faster CPU with a chipset that more efficiently handles I/O). Kill the problem with brute force speed, because after all, the real problem is that, somehow, your system isn't working fast enough to handle incoming

MIDI data at MIDI baud rate, as far-fetched as that may seem. (ie, Most computer systems run at <u>much</u> faster rates than MIDI baud rate).

There is one footnote to be added here. If you're using a Creative Labs' AWE64 or AWE32, and having problems with garbled MIDI Thru (ie, the MIDI data from your controller keyboard is recorded into the computer OK but when simultaneously sent to MIDI OUT or the AWE built-in sound module, you get weird MIDI response), this is due to a bug in the chipset used in some CL cards. This is particularly noticeable with Roland keyboard controllers used with an Awe32/64. (In fact, Roland discovered the bug in CL's chipset). You need to send the card back to CL for servicing. Contact CL tech support.

I have this program which displays a graphical piano keyboard. When I click the mouse on a "key", the program outputs a MIDI message to my sound card. I have another program which is a sequencer program. It can record MIDI messages coming into the computer. How can I get the first program to send its MIDI data to the sequencer so that I can record my "playing" (ie, I can use the computer as a controller at the same time that I'm recording my performance).

If the two programs aren't designed to use some sort of scheme to internally pass MIDI data between themselves, then you need to rig up some connection between MIDI input and MIDI output. That could be as simple as just connecting the computer interface's MIDI OUT to its MIDI IN with a MIDI cable.

Alternately, if both programs are Windows software, you can use software called "MIDI Router" by Zoltan Janosy, or Hubi's "Loopback", or MidiOx. These programs feature a special MIDI device driver that you install (just as you would any other Windows audio/MIDI driver) which makes a software "MIDI connection" between any Windows program outputting MIDI data and any Windows program inputting MIDI data. (It connects the first program's MIDI OUT to the second program's MIDI IN). Note that this isn't a perfect solution. For one thing, you may now have trouble using a MIDI program that does simultaneous MIDI input and output (ie, because now it's connected to itself, and feeding back upon itself). In that case, you'll have to disable MIDI Router's function whenever you don't need it. Secondly, this extra software layer does slow down MIDI input and output.

Why is my MIDI sound module playing only some of the MIDI channels? I played a MIDI file that has tracks on MIDI channels 1 to 16, but I only heard the tracks on channels 1 to 8.

There are two possibilities here. First, does your sound module support all 16 MIDI channels simultaneously? Some older models do not. For example, a Roland D-70 only has 5 "Parts", which means that it can play only 5 MIDI channels simultaneously. (See my article on [Roland sound module architecture](#) for more details concerning Roland modules). Some sound cards, particularly early 8-bit Sound Blasters and their ilk, also didn't recognize all 16 MIDI channels. There is only one official MIDI specification, and it specifies 16 MIDI channels. But, Creative Labs generally makes sound cards for game players, not musicians, so the cards typically aren't as "full-featured" as cards for musicians. Game players buy cheap sound cards, and that means CL had to "cut corners" on that "esoteric MIDI stuff". One way that CL cut corners was by not supporting all 16 MIDI channels. Early CL cards had very limited polyphony, so it wasn't as if anyone was going to do complex MIDI arrangements on them anyway. So, two "sub-standards" were devised known as "base level" and "extended level" MIDI. "Base level" supports only something like 8 MIDI channels and really limited polyphony, and discards any MIDI events on the remaining MIDI channels. "Extended level" supports slightly more channels and polyphony. Both are for cheesy sound cards. "Base level" is for really shitty, now-obsolete sound cards. "Extended level" is for the slightly less shitty, but equally obsolete cards (ie, you know, essentially the same basic shit design repackaged in a box with the word "Pro" appended to the name of the sound card).

If your sound card or module <u>does</u> recognize all 16 MIDI channels (and you've checked that it is in fact setup to do so -- try connecting a controller directly to it and sending messages on one of the troublesome MIDI channels), the problem could be software related. If you're using "Custom Configuration" in Win95's MultiMedia MIDI setup, make sure that you've got the MIDI channels set to the intended device. Also, check your sequencer software's MIDI configuration. Maybe it's your sequencer program that is setup for "base level" or "extended level" MIDI (and therefore the program itself is discarding MIDI data on those extra MIDI channels). Check your sequencer's "MIDI setup" options, and look for something that says "base level" or "extended level". Turn that off. Instead select full MIDI support, sometimes indicated by the label "General MIDI". Incidentally, if your software <u>does</u> screw around with this "base level" and "extended level" crap, this is a good indication that you've got "toy" music software there, which was designed to be used with "toy" audio cards. Professional music software does <u>not</u> bother with "base level" and "extended level".

I've heard that MIDI (note) delays are due to daisy-chaining MIDI sound modules. Each module's THRU jack delays the MIDI signal before it gets passed to the next module, and you can hear the delay after about 3 modules are daisy-chained together. Is this true?

No, but this is such a popular myth that it has been elevated to the status of "urban MIDI legend". The amount of time that it takes the MIDI signal to pass from a module's MIDI IN jack to a <u>properly configured</u> MIDI THRU jack is a matter of a few microseconds. You would need to daisy-chain <u>many</u> (ie, certainly more than 30) modules before you could even begin to ascertain any kind of delay.

How did this myth get started? Well, in the early days, MIDI modules were very limited in polyphony. They weren't even multi-timbral, so you usually needed many, many modules in order to play a MIDI arrangement with lots of musical parts and heavy "note density" (ie, lots of notes playing upon the same beat). This was a really expensive proposition, so in the early days, musicians tended to do simpler arrangements with typically small, limited MIDI setups. They didn't tax the MIDI bus with lots of notes and therefore, didn't notice the limitations of "MIDI bandwidth". (For a more indepth discussion of MIDI bandwidth, read the article [Multiple MIDI outputs](#) now). Later on, as MIDI sound modules got cheaper, with more polyphony, musicians started daisy-chaining more modules together. They also started making more complex MIDI music to use all of this additional polyphony. And that's when MIDI bandwidth reared its ugly head. The musicians failed to recognize that they were taxing the MIDI bus with denser arrangements now. Instead, they mistakenly assumed that the delays that they were hearing must be due to daisy-chaining MIDI modules. Hence, the "MIDI THRU Delay Theory" was born. It's incorrect.

Now, note above that I've emphasized <u>properly configured</u>. Some manufacturers foolishly do not follow the MIDI specification properly. The MIDI THRU jack is supposed to be "directly coupled" to the MIDI IN jack. Instead, some manufacturers process the MIDI IN, and THEN send it out the MIDI THRU jack. (A telltale sign that this is likely being done is a module that has no dedicated MIDI THRU jack. Rather, it has a "software switchable" MIDI OUT/THRU jack). This "processed output" likely will introduce more delay than a direct coupled MIDI THRU -- a LOT more delay. But we're talking about an aberration of the MIDI spec. My advice is to avoid using that THRU jack, or only buy gear that follows the MIDI spec precisely. I tend to buy Roland stuff because the company is religious about not screwing around with the MIDI specification. (Roland is one of the few companies that even go so far as to implement Active Sense).

A processed MIDI THRU is a very, very, very, very bad thing. It shouldn't be necessary, and if you got it and don't need it, you can't get rid of it. (If you're daisy-chaining so much gear that you need to worry about "cleaning up" the MIDI signal along the path, then you should <u>definitelyDo not solve the problem with processed MIDI THRU signals.</u> That introduces more problems than it ever solves). Avoid it. Avoid products that do this. Stick with products that follow the MIDI spec. We've been using it for a long time. It works. There are much better solutions to connecting gear than a processed MIDI THRU.

I'd like to use my AKAI S-1000 MIDI sampler to record digital audio to my computer HD. Can I do that?

You can't do "direct-to-disk" recording to your computer's HD with an external MIDI sampler. The wave data transfers that such MIDI samplers do is not a real-time transfer. MIDI Sample Dump Standard sends data far too slowly over the MIDI cable to be able to store it as fast as it's being recorded. It's a much, much slower transfer than playing back the data. Even SMDI (ie, the SCSI version of SDS) isn't real-time (although much faster than SDS). Besides, few samplers are designed to do recording at the same time that they're doing a dump of a waveform to another device (the Peavey SX being an exception). Usually samplers require a direct link to storage medium in order to record in real-time to that medium (ie, the HD has to be directly connected to the sampler).

You'd need a direct hardware connection to your computer's HD. This may be possible using a SCSI controller in your computer and SCSI HD, plus a sampler that can record directly to a SCSI drive, but probably your sampler's file format will be different than what your computer requires (ie, FAT or NTFS format for most PCs). And, you couldn't control the recording with PC software. You'd have to do so from your sampler's front panel. So, it won't be practical anyway.

Of course, you can record digital audio on your MIDI sampler, as much as its RAM will allow, and then later transfer that data to the PC via SDS or SMDI protocol. You need a PC program such as "sdxsend" (http://alfred.uib.no/People/midi/midi_prog.html), which works with SDS capable samplers. If you've got an akai sampler, you can use floppies to transfer your samples using "Akaidisk v2.0" (http://www.cs.ruu.nl/~jules/Akai/) which reads/writes AKAI format floppies on a PC floppy drive. Then, you can edit the waveforms on the PC with PC wave-editing software such as SoundForge (assuming that it will read the data format of "sdxsend" or "Akaidisk") and later send the waveform back to the sampler. Obviously, this isn't real-time control.

You'll need a computer card with an ADC in order to do direct-to-disk recording to your PC's HD. For example, a DAL CardD, Roland RAP-10, TB Monterrey or Tahiti or Tropez, etc, are cards with an ADC.

Another alternative is to use a DAT recorder to record your digital audio tracks. Then, you can transfer the waveforms to the PC (saving as WAVE files) using a card with digital audio I/O such as the CardD+ (or the digital audio I/O only version of it, http://www.digitalaudio.com) or the Zefiro Acoustics ZA1 or ZA2 (http://archive.uwp.edu/zefiro) or Turtle Beach Fiji or Pinnacle or other cards with digital I/O. (Even the Awe64 Gold now has digital I/O). You can edit the waveforms with PC software as above, and then digitally transfer it back to the DAT or burn a CD using software such as Adaptec's Easy CD Creator with a capable CD writer drive.

I have an AWE32 (ie, built-in wavetable and built-in FM synth) to which I've added a Yamaha DB50XG daughterboard. I have 3 separate output devices for MIDI playback listed although I only have one sound card. Why? Also, how should I setup MIDI channels between these MIDI output devices? Do I use one channel per output?

You actually have 3 separate output devices (all on one card). You have the *SB FM Synth*. That supports 16 MIDI channels. (But since its sound quality and polyphony is extremely poor, you likely won't be using it much except for music on really old game software). You have the *SB WaveTable Synth*. That supports 16 more MIDI channels. And you have the *SB MIDI Out* (to which the DB50 is internally attached). That supports another 16 MIDI channels. So, each one of the components is its own output device, not internally daisy-chained to the other 2 components, and therefore has its full 16 MIDI channels which it does not need to share with the other devices. (You have a total of 16+16 +16 MIDI channels). That's why you have 3 outputs listed -- because even though they're on the same card, they each have their own set of 16 MIDI channels.

MIDI channel #1 on your AWE32's WaveTable Synth is <u>not</u> the same as MIDI channel #1 on your DB50. Why? Because these two devices are not daisy-chained together. (If they were, <u>then</u> they'd be using the same MIDI channel 1). They each have their own set of 16 MIDI channels to work with. So, if you set one CakeWalk track to output to the AWE32 WaveTable Synth on MIDI channel 1, and you set another track to output to the SB MIDI Out (ie, the DB50) on MIDI channel 1, then these are going to two different places (ie, outputs). They're not the same MIDI channel 1. One is on the AWE32 WaveTable output, and one is on the MIDI OUT output.

So there's no reason whatsoever for you to divide up MIDI channels between devices, <u>assuming that you're using a sequencer program which supports outputting to all 3 devices simultaneously</u>, such as CakeWalk. If you have a sequencer that supports only 1 output device at a time, and therefore only 16 MIDI channels total, <u>then</u> you'll need to use Win95's Advanced MIDI setup or Windows 3.1's MIDI Mapper, and divide up the 16 channels among your output devices. (I'd recommend using 6 channels for the AWE32 WaveTable, and 6 for the DB50, and forget the FM synth).

Of course, if you attach external MIDI modules to the AWE32's MIDI Out, then you're effectively daisy-chaining these to the DB50, since the daughterboard is internally attached to the AWE32's MIDI Out. Now, you've got to share the MIDI Out's 16 channels between the DB50 and other external modules. (Setup each module to ignore certain channels. The DB50 can be set to ignore channels via System Exclusive messages). No, you can't reroute some of WaveTable Synth's or the FM Synth's 16 channels to the MIDI Out (and therefore have more than 16 channels going to MIDI Out). The MIDI Out has only its own 16 channels with which to work, and these must be divided among all modules attached to MIDI Out, including the DB50.

I have a General MIDI sound card to play my musical sequences. I always use a different MIDI channel for each track in my sequencer. Is this unnecessary?

That depends upon what is on each track.

You'll likely need to have each "instrument" in your arrangement upon a separate MIDI channel. For example, don't put the piano <u>and</u> the sax parts upon the same MIDI channel (unless the parts don't occur during the same musical bars. I'll explain that later). Why? Because a single MIDI channel can't <u>simultaneously</u> play 2 different patches. A patch is usually analogous to one instrument sound. (On some modules, a patch can be setup to divide the MIDI note range between 2 or more instrument sounds, ie, a "split". But on General MIDI devices, such as what you have, most patches feature only a single instrument sound across the entire MIDI note range. The GM patch named "Bass + Lead" is an exception. Notes lower than middle C play a bass guitar sound, whereas notes above C play a synth lead sound. So you can play two instrument sounds with that one patch. But that's an exception, and the other GM patches feature only 1 instrument sound across the entire MIDI note range).

But that's not necessarily to say that every sequencer track will be on a different channel. Maybe you'll have 2 tracks that are both using the same piano sound (ie, same patch) on the same module,

and have the same settings (ie, the same panning, volume, etc). Maybe one track is some background piano part, and you decided to put the piano solo upon a separate track (so that it is easier to edit separately from the background piano notes). In that case, you'll likely have the tracks set to the same MIDI channel. After all, even though you're using 2 sequencer tracks, they are both playing the same part -- the piano part.

Of course, if you have 2 instrumental parts that don't occur during the same musical bars, then you can put both parts upon the same MIDI channel. For example, let's say that you have a trumpet solo during bars 1 to 16. Then you have a sax solo during bars 16 to 32. At no time do any trumpet solo notes sound while the sax solo notes are sounding (and vice versa). Although you may choose to record the solos to separate sequencer tracks, you can have both solos set to play upon the same MIDI channel and the same sound module. You would have a MIDI Program Change at the start of the trumpet solo. It would change to the trumpet patch. Then at the beginning of bar 16, after all of the trumpet notes have stopped but before any sax notes have started, you would have a MIDI Program Change to switch to the sax patch. Using the same MIDI channel for instruments whose parts do not occur upon the same musical bars (ie, the parts don't overlap) is a good way to conserve your MIDI channels for musical parts that <u>do</u> overlap.

Sometimes you do have to use more than 1 MIDI channel for a certain instrument if you're doing some stereo effect. For example, let's say that you want to have all of the piano notes played with the left hand panned to the left, and all of notes played with the right hand panned to the right. You'll need to record the notes for each hand upon a separate MIDI channel. Why? Because not only is a single MIDI channel limited to playing only one patch, it is also limited to only one pan position. That means that you can pan notes on MIDI channel 1 to the left, or to the right, but not both. You can't have some notes panned to the right and other notes simultaneously panned to the left. You'll need to separate the left and right hand parts to 2 separate MIDI channels, and pan one channel to the right and the other channel to the left. (Alternately, if your sound module offers "stereo patches", you may wish to do this sort of stereo imaging in the patch settings of your sound module, as opposed to "faking it" via MIDI panning).

But, mostly, you'll have each instrument's part on just one track (with its own, unique MIDI channel). Especially if your arrangement doesn't use more than 16 patches, you've got it covered with only 16 MIDI channels, and you can safely dedicate a unique MIDI channel to each instrument (ie, patch).

I have a MIDI interface that attaches to my parallel port. It isn't working and the driver says that there's a hardware problem. The interface works upon another computer I tried.

First, get into your BIOS setup, and check whether you have the parallel port set to "Enhanced" or "Bi-directional" modes. Some parallel port MIDI interfaces do not support these two modes, and will only work when the parallel port is in a basic, uni-directional mode. (Of course, changing this may then affect your printer operation if it shares the port).

Secondly, if you have a printer already using the first parallel port (ie, LPT1), it may be that the

printer's driver is not allowing the MIDI interface's driver to use the IRQ (ie, #7). If you have a second parallel port (ie, LPT2), then use that (and make sure that you let the MIDI driver know that it should use IRQ 5 -- but make sure that you don't have another sound card set to that, since many SB cards use this IRQ by default). If you don't have another parallel port, you may have to try removing your printer driver to see if that resolves some conflict.

I have a MIDI interface that attaches to my serial port. It isn't working and the driver says that there's a hardware problem. The interface works upon another computer I tried.

Do you actually have free resources associated with the COM ports? Note that COM1 and COM3 both share IRQ 4, and COM2 and COM4 share IRQ 3. So, if you have two devices that require use of the IRQ, you can't set them to COM1 and COM3 respectively as that would force them to both try to simultaneously use IRQ 4. Ditto with COM2 and COM4. If you have a serial mouse plugged into serial port 1, and you have a modem plugged into serial port 2 (an internal ISA card will also typically be set to use the second COM port and its IRQ), then you've already used those IRQ's associated with the COM ports. You won't be able to also use your serial port MIDI interface with a serial mouse and modem already in your computer.

Of course, you'll want to make sure that your serial port isn't disabled in your BIOS. Also, make sure that your serial port uses a chip (such as the 16550) that supports the 38KHz baud rate that serial MIDI interfaces require.

To what extent are piano (ie, sustain) pedals supported in MIDI? Does General MIDI cover this issue? If I have a controller that includes at least a sustain pedal, does it output some kind of "pedal down" and "pedal up" message? If so, how consistently will sound modules respond? I'd guess that when long arpeggios are performed, either the oldest or newest notes will get cut off after the limitation on voices is surpassed, but which is it, and how much of a problem is this?

MIDI has many "controller functions" to adjust such things as a patch's volume, pan, portamento time, modulation (which could be setup to implement a vibrato effect, or tremulo effect, or other effects), reverb amount, chorus amount, and many other functions. (For a list of defined controllers, see Defined MIDI Controllers). There is even a controller that specifically functions as a sustain pedal. (ie, When "on", it causes the module to "hold" the sustain portion of its volume envelope even after receiving a note-off. When "off", the envelope proceeds to its release portion as usual when a note-off is received. So yes, it does implement a "pedal up" and "pedal down" simulation of a piano sustain pedal). There are two other specific controllers to simulate the "soft pedal" and the "sustenuto

pedal". These controllers are all part of the MIDI spec itself.

The GM spec simply states that a module <u>must</u> support particular controllers, in a particular way. Think of GM as a <u>minimum</u> standard for what portion of the MIDI spec a module must support, and how it must support that portion of the spec. GM is just a minimum level of support -- a minimal "setup" such as 128 specific patches and 47 specific drum sounds that a module must have in order to be at least GM compliant. The reason for GM is to ensure that all modules have a minimal, standard setup so that MIDI files that adhere to this minimum standard can playback easily and properly upon all GM equipment.

One of the controllers that GM mandates support for is the sustain pedal, so virtually every module supports that function. (The soft and sustenuto pedals are not part of the GM spec, and are rarely supported, so you'd have to specifically shop for a module that supports such, ie. goes beyond GM support, if you want those other 2 pedals).

Most modern sound modules support many of the above functions (ie, adjustment of volume, pan, reverb level, sustain on/off, etc), although how extensive the module's support is usually varies according to price and model. Many modules offer an assignable pedal input. That is to say that you can assign the pedal to control any of the above functions (or maybe just a limited set of functions). This pedal will also typically generate a MIDI controller message. Typically, it will use a "generic" controller number, since you can assign the pedal to control any one of a variety of functions. Most modules also have a pedal jack that is hard-wired to that sustain pedal function. This will generate that defined controller number for the sustain pedal (ie, controller #64). Virtually all modules support this particular controller, and will respond in a similiar (if not identical) manner.

Most modules implement dynamic voice allocation, which means that as voices are all used up (because you're sustaining notes) and you play more notes, previously played notes are "stolen" (ie, a previously played note is muted so that a new note may sound). This is no standard algorithm for dynamic voice allocation. Some modules have a more intelligent scheme than others, and will try to drop the "least important" notes first. (For example, one module may discern that you're sustaining the 4 notes C-E-G plus another C an octave above the first. Even though the E may have been the last played note, maybe the module will steal a C note because it's a doubling of another playing C. Some modules try to avoid stealing the highest and lowest sounding notes, as your ear tends to be most sensitive to these dropping out). So, it may be possible that you'll prefer one module's dynamic voice allocation to another. To be sure, the more voices that a module has, the less likely you'll run into a "problem" with voice allocation.

Personally, I think that 32 voices is more than enough to simulate a piano well (but of course, with a multitimbral module, you may be playing other patches simultaneously, and spreading out voices among numerous patches).

Whenever I depress the sustain pedal, all sounds on my module no longer sustain when I release the key. When I release the pedal, the sound sustains. In other words, my

pedal works exactly the opposite to the way that a real pedal works. What have I done wrong?

You bought a pedal that was made for a keyboard that had a reverse polarity to yours. Some modules want "open circuit" switches and some want "closed circuit" switches. You can tear open the pedal and rewire it, assuming that your module doesn't have some feature to reverse the polarity (ie, many new units have that function selectable by the user, or if you press down your pedal while turning on the unit, it may automatically adjust to using reverse polarity). But a better idea is to buy a pedal that has a polarity switch on it.

I want to connect several external MIDI modules to my sound card's one MIDI Out. I must setup each module so that it responds to MIDI channels that the other modules do not also respond to. How can I do this setup using MIDI messages (so that it can be done automatically at the start of a MIDI file's playback for example)?

There is no standard MIDI message to explicitly tell a sound module to ignore certain MIDI channels. Many sound modules do support their own System Exclusive message to do that. But that message will vary from manufacturer to manufacturer, as Sysex messages tend to do, and some modules may not offer any way to setup MIDI reception other than by manually using the module's control knobs. You'll have to check the manual for each module, and see if it offers such a Sysex message.

Now, I said that there's no MIDI message to <u>explicitly</u> tell a sound module to ignore MIDI channels, but in later revisions of the MIDI specification, the MIDI controller message, "Monophonic Operation" (ie, controller 126) is redefined to work in conjunction with the "Omni Off" Controller (ie, 124) message to allow you to set a MIDI module to respond to only a limited set of MIDI channels. One caveat with using this method is that the MIDI channels must be adjacent numbers. For example, you can get a module to respond to channels 6 through 10. But you can't tell it to respond to 6, 7, 8, and 10 (ie, skipping channel 9). Well, not via the Monophonic Controller message anyway.

There's another caveat. You must manually set the "Base Channel" of every module to be different. (Again, some modules let you select the Base Channel with a System Exclusive message). The "Base Channel" is the channel upon which messages such as Omni Off and Monophonic Operation must be sent in order for the module to recognize the message.

Here's how Monophonic Controller works in conjunction with Omni Off. (NOTE: The following discussion assumes a multi-timbral module. Monophonic Operation does not work the following way for modules that aren't multi-timbral).

Like all controllers, a Monophonic Operation Controller message has a data value associated with it.

If Omni is off (ie, you have sent the Omni Off Controller message already -- most multi-timbral modules automatically powerup in this state anyway), this value tells how many MIDI channels the module is expected to respond to. 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.

If Value is 0, what this means is that the module should play as many MIDI channels as it has multi-timbral "Parts". So, if the module can play 16 of its patches simultaneously, then it can respond to MIDI messages on all 16 channels.

If Value is not 0 (ie, 1 to 16), then that's how many MIDI channels the module is allowed to respond to. For example, a value of 1 would mean that the module would be able to respond to only 1 MIDI channel (and therefore play only 1 Part). If a module 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 module can play only 5 Patches simultaneously, and receives the value 8 in the Mono message, then the module will play 5 patches on MIDI channels 0 to 4 and ignore messages on channels 5 to 7. (Here we assume a base channel of 0).

Most multi-timbral modules allow you to specify a Base Channel. This will be the lowest MIDI channel that the module responds to. For example, if a Mono message specifies that the module is to respond to only 2 channels, and its Base Channel is 4, then the module responds to channels 4 and 5.

Of course, in order to be able to send an individual Mono message to each of your modules, each one has to be set to a different Base Channel. Otherwise, 2 modules with the same Base Channel would both respond to the same Mono message. And of course, you need each module set to a different Base Channel in order to also have each use its own, unique range of MIDI channels. After all, the Base Channel is the first channel that the module always responds to.

Getting a headache yet? As you can see, because of this Base Channel hassle, the above method of setting up a multi-timbral module's MIDI reception means that you likely have to setup each module manually anyway (unless the module also allows setting Base Channel via System Exclusive). It would have been nice to have a dedicated MIDI message to set channel reception which, like System Exclusive, didn't need to be sent upon a particular MIDI channel, and had two data values to set the desired lower and upper ranges for MIDI channels. But the MIDI designers didn't think of this at first, and this method of hacking the Monophonic Controller was the best, backward-compatible trick they could devise later on. Because of its convoluted and inflexible hassles, many multi-timbral modules don't even support the Monophonic Operation controller at all. So it's not even very much of a "standard" approach today. Why didn't the MIDI designers think to have a dedicated message to set MIDI channel reception right from the start? Well, it never occurred to them that it was important. They didn't have multi-timbral modules back then. And it's a weird concept to setup MIDI channel reception using MIDI messages. Think about it.

Most modules instead use a simple System Exclusive message to set MIDI channel reception, and usually this allows for picking MIDI channels in any order. For example, the Yamaha DB50XG daughterboard uses the SyxEx message (values in hexadecimal):

F0 43 10 4C 08 **Channel** 04 7F F7

where *Channel* is the desired MIDI channel that you want the DB50XG to <u>ignore</u>. (0 is the first MIDI channel). You would repeat this message for each channel that you wish ignored, substituting that channel number in the message. For examples with the Roland RAP-10, see <u>Roland Audio Card FAQ</u>.

But the drawback with SysEx messages is that everyone does it differently. There's really no easy, "works for all MIDI modules" way to setup a module's MIDI channel reception over MIDI.

In Windows 3.1 MIDI Mapper, what is the difference between the patch and key maps?

Patch maps are used to remap MIDI Program Change messages. For example, maybe a MIDI file assumes that the Grand Piano patch is #1 on your module. But maybe your module has an Oboe patch for #1 instead. Maybe the Grand Piano patch is #40 on your module. So when the MIDI file tells your module to change to Grand Piano, it will accidentally misinterpret this as a command to play the Oboe. You can use MIDI Mapper to fix this. Set MIDI Mapper's patch #1 to send a Program Change value of 40 instead. Now when the MIDI file sends a Program Change for patch #1, MIDI Mapper instead sends a Program Change for patch #40, and your module does in fact play that Grand Piano.

Most MIDI files nowadays expect a General MIDI patch set (ie, certain instruments in a certain order). If your module is not General MIDI, then you should use MIDI Mapper to remap a GM patch set to your module's patch numbers.

Key maps are used to remap MIDI Note messages. Usually, every different note number of a Drum Kit plays a different drum sound. For example, a middle C Note message sent to a General MIDI drum kit, will play a "High Bongo" drum. Now, maybe the drum kit in your module has a snare drum sound mapped to that middle C key. Maybe the High Bongo is assigned to note number 61. So, when a GM MIDI file tells your module's drum kit to play a High Bongo, it will accidentally misinterpret this as a command to play a snare. You can use MIDI Mapper to fix this. Set MIDI Mapper's key #60 (ie, middle C) to send a note number of 61 instead. Now when the MIDI file sends a Note message to play note 60, MIDI Mapper instead sends a message for note #61, and your module does in fact play that High Bongo.

Registered and Non-registered Parameter Number controllers

The MIDI Non-registered Parameter Number (abbreviated to NRPN) and Registered Parameter Number (abbreviated to RPN) controllers can be used to adjust and control a wide range of functions upon a MIDI device. These controllers are used in conjunction with the Data Button Decrement (#96), Data Button Increment (#97), and Data Entry Slider (#6 for coarse adjust, and #38 for fine adjust) controllers.

Non-registered Parameter Number

The Non-registered Parameter Number controllers are controller #98 for the "low bits", and controller #99 for the "high bits" of the parameter number. Here's how a manufacturer may use NRPN.

Assume that a manufacturer decides to allow a musician to adjust the **Filter Resonance** of a Voltage Controlled Filter using some MIDI controller messages. There is no standard MIDI controller defined for this purpose. One option that a manufacturer has is to use some Non-registered Parameter Number to control Filter Resonance. The available NRPN's are numbers 0 to 16,383. Just for the sake of illustration, let's say that the manufacturer chooses 14,000 as the parameter number for Filter Resonance (ie, the NRPN is 14,000). Note that a manufacturer is free to use parameter numbers for any desired purpose.

A Data Button Increment, Data Button Decrement, and/or Data Entry Slider controller will be used to actually change the Filter Resonance value. Each Data Button Increment message sent to the device will increase the Filter Resonance by 1. Each Data Button Decrement sent to the device will decrease the Filter Resonance by 1. Each Data Entry Slider will set the Filter Resonance to whatever value is specified in the Data Entry Slider MIDI message. But before these 3 controllers can adjust Resonance, first you must send the pair of NRPN controller messages which tell the device that it should regard those 3 controllers as affecting Filter Resonance. The parameter number in the NRPN messages will be 14,000 because that's what this manufacturer chose as the parameter number for Filter Resonance.

To create the NRPN message, you need to convert the 14,000 (decimal) to a binary representation. The reason is because MIDI can't transmit numbers larger than 127 within a message unless you "break up" the number into several "pieces of data", where each piece is less than 127, and the receiving device puts those pieces back together to recreate the original, large number. So, we need to convert the 14,000 to binary in order to figure out how to break it up into 2 pieces. 14,000 works out to be the following (16 digit) binary stream:

0011 0110 1011 0000

NOTE: The Windows 95 calculator, when in scientific mode, can be used to convert values between decimal, hexadecimal, and binary notation. But note that the calculator trims off leading zeros (ie, It will display only 14 or less of the 16 digits) of binary values. Make sure that you remember to prepend leading zeros so that you always have 16 binary digits.

In order to break this up into 2 pieces, we discard the first two zeros. (Due to the parameter number range being 0 to 16,383, the first two of the 16 binary digits will always be zero). We then take the next 7 digits, put a 0 in front of them, and recreate a decimal value from them:

0 11 0110 1

The above binary stream converts to 109 (6D in hexadecimal). These are the "high bits" of the parameter number.

We then take the remaining 7 digits, put a 0 in front of them, and recreate a second decimal value from them:

0 011 0000

The above binary stream converts to 48 (30 in hexadecimal). These are the "low bits" of the parameter number.

So, when we transmit the parameter number of 14,000 over MIDI, we break it up into 2 pieces; the decimal values 109 and 48 (where 109 is the "high bits" and 48 is the "low bits").

Now we need to send the pair of NRPN controller messages to the device. We send NRPN controller #99 (ie, the "coarse" controller) with our high bits of the parameter number:

176 99 109

NOTE: I'm expressing all values in decimal instead of hexadecimal. Also, I'm assuming a MIDI channel of 1 (ie, actually 0), so that's why I show a 176 above for the MIDI Controller Status. Of course, MIDI channel 2 would be a MIDI Status of 177. MIDI Channel 3 would be 178. Etc. If you're using a sequencer program to enter these messages, the program will probably figure out the status for you -- you only need to tell the sequencer that you want to enter a MIDI controller event, with a controller number of 99, and for the data value, enter the low bits of your desired parameter number (above, that's 109).

We then send the NRPN controller #98 (ie, the "fine" controller) with our low bits of the parameter number:

176 98 48

Incidentally, here are the above 2 messages shown as hexadecimal (which is the way that they're

listed in Roland literature):

B0 63 6D
B0 62 30

After sending these 2 messages, any subsequent Data Button Decrement, Data Button Increment, or Data Entry Slider message (on the same MIDI channel) will adjust the Filter Resonance. For example, let's use the Data Entry Slider controller to set Filter Resonance to the value 100:

176 6 100

Note that the manufacturer can define other NRPN numbers. For example, let's say that parameter number 14,001 is used to adjust the **Filter Envelope Release Time**. Now, if you want to use the Data Button Increment, Data Button Decrement, and Data Entry Slider messages to adjust Filter Envelope Release Time, you first have to send the pair of NRPN controller messages with a parameter number of 14,001. (And you have to break up the 14,001 into two pieces; the "high bits" and the "low bits", as we did above with the parameter number of 14,000). Of course, after you send the pair of NRPN controllers with this new parameter number, the Data Button Increment, Data Button Decrement, and Data Entry Slider will no longer adjust Filter Resonance, but instead, will adjust Filter Envelope Release Time. In other words, the Data Button Increment, Data Button Decrement, and Data Button Slider adjust whatever "setting" is associated (by the manufacturer) with the last received pair of NRPN controllers (and the parameter number specified by that pair). That's how these MIDI controllers are related.

CakeWalk Example

Here's the procedure for entering the above example of setting Filter Resonance to the value 100 using CakeWalk.

1. Select the desired track where you wish to insert the NRPN controller.
2. Go to the **Insert** menu and select **Series of Controllers**.
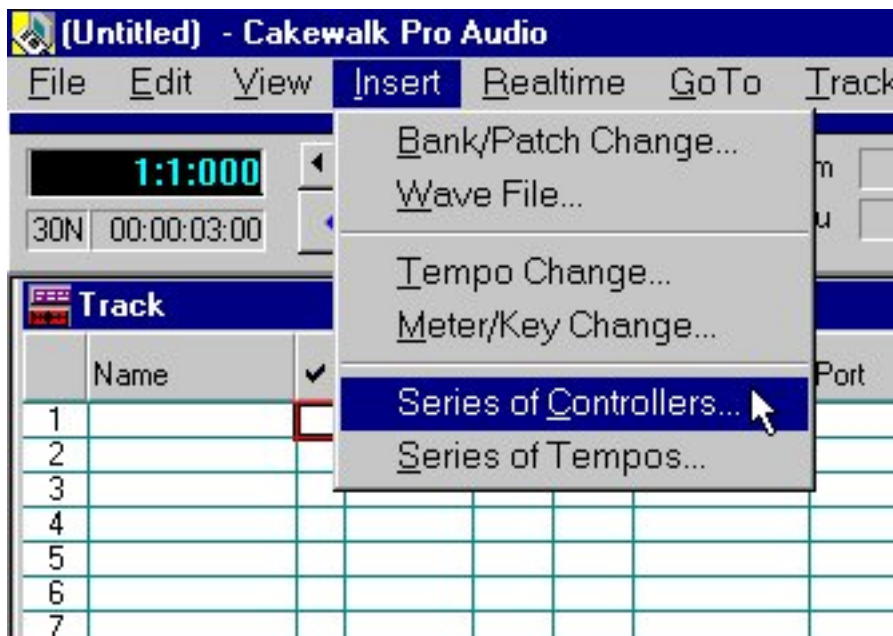
3. In the dialog that opens, select **NRPN (coarse)** controller.
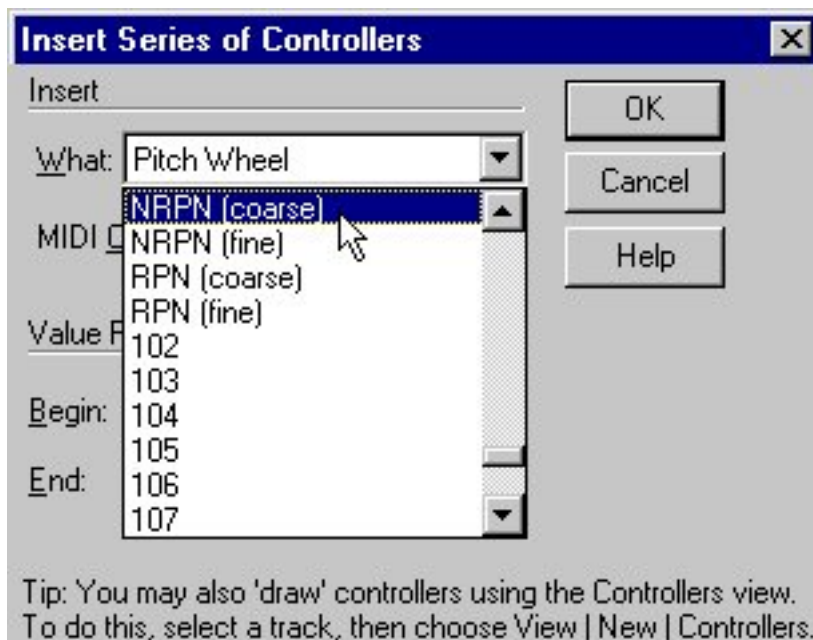
4. For the **Value Range End**, type in the "high bits" of the parameter number. For this example, that would be 109.
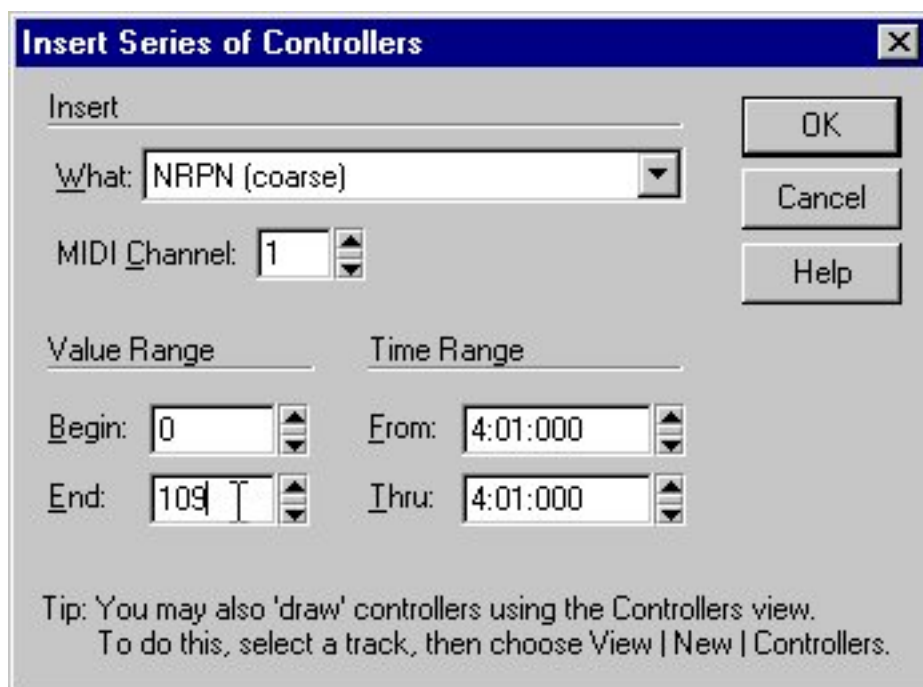
5. Set both the **From** and **Thru** times to the Measure, Beat, and Clock when you want the NRPN to be sent.
6. Click on the OK button.
7. Go to the **Insert** menu and select **Series of Controllers**.
8. In the dialog that opens, select **NRPN (fine)** controller.
9. For the **Value Range End**, type in the "low bits" of the parameter number. For this example, that would be 48.
10. Set both the **From** and **Thru** times to the Measure, Beat, and Clock when you want the NRPN to be sent.
11. Click on the OK button.
12. Go to the **Insert** menu and select **Series of Controllers**.
13. In the dialog that opens, select **Data Entry (coarse)** controller.
14. For the **Value Range End**, type in the value you wish. For this example, that would be 100. (Note that some MIDI devices may want to use both the coarse and fine Data Slider controllers if a parameter can be set to greater than the value 127. In that case, you break up the value into "high" and "low" parts just like with the NRPN coarse and fine controllers).
15. Set both the **From** and **Thru** times to the Measure, Beat, and Clock when you want the Data Slider to be sent. (Note that this must be later than the 2 preceding NRPN controller messages. If you enter this controller after the preceding 2, CakeWalk should send it after those other 2, even if you specify the same times for all 2 events. But, if you wish to be sure, you can choose a time that is at least one clock later than the preceding 2 events).
16. Click on the OK button.

Registered Parameter Numbers

Of course, there were some "settings" that nearly all manufacturers found themselves defining, such as Pitch Wheel Bend Range. For this reason, we also have a pair of controllers for these **defined**

parameter numbers. We use the Registered Parameter Number (RPN) controller messages for these defined parameter numbers. The RPN controller #101 specifies the "high bits" of the parameter number, and RPN controller #100 specifies the "low bits". There are (as far as I know) only a couple defined RPN's. Parameter # 0 adjusts Pitch Wheel Bend Range. Parameter # 1 adjusts Master Tuning. Of course, you break up these parameter numbers in the same manner as you did NRPN's, but you use RPN controllers 100 and 101 to transmit the parameter number (instead of the NRPN controllers 98 and 99). For example, here I tell the device that I want to adjust Pitch Wheel Bend Range, and then I use the Data Entry Slider to set it to 12 steps (ie, an octave up and down). This requires 3 events (ie, the two RPN controllers to tell the device to adjust parameter number 0 for Pitch Wheel Range, and the Data Entry Slider to set the value for Pitch Wheel Range).

176 101 0
176 100 0
176 6 12

Unlike with some other operating systems, in Windows, a program should refrain from directly reading and writing hardware ports on a sound card. Whenever possible, a program should instead call functions in the Windows operating system, which will do the actual hardware reading and writing for you (in conjunction with the sound card's Windows device driver). For writing data to the card, you pass that data to an operating system function that sends that data to the card's device driver, which in turn writes that data to the card. For reading data from the card, you call an operating system function that causes the driver to read some data from the card, perhaps placing that data into a buffer whose address you've specified. In this way, you'll create a Windows program that will:

1. continue to operate under versions of Windows which restrict access to hardware, in order to implement "crash protection" (for example, Windows NT/2000/XP). Typically, such versions allow drivers to operate in special "modes" that allow hardware access, but restrict programs from the same access.

2. work with a wide range of sound cards and MIDI interfaces. It's the driver that has the hardware specific code in it, not your program. The drivers for various hardware all look the same to the program and operating system. (ie, Every driver has the same set of functions in it, which are called by the operating system using one "standard" method for all. Therefore, your program automatically supports all such drivers and their hardware by virtue of calling Windows operating system functions).

So, is there a "standard" that all Windows sound/MIDI drivers are written to follow (ie, such that each driver has the same set of functions, to be called using the same method)? Yes. This is called Windows "Media Control Interface" (ie, MCI). The Windows MIDI Mapper, the MCI Sequencer Device (driver), and the MCI Wave Audio Device (driver) are 3 parts of MCI that are particularly relevant to sound/MIDI cards.

When a user buys a sound/MIDI card, he installs the Windows MCI driver that ships with it (for example, using Windows Control Panel's *Add new Hardware*. See the explanation of installing a driver under Win95 for more details). Virtually every PC sound/MIDI card ships with a Windows MCI driver for that card. You should never have to write a Windows driver for a card unless you're the manufacturer of that card. As a program developer, you don't care what particular brand/model card is being used (as long as it supports what you need to do). You don't even need to know the name of the driver, because you call it indirectly via functions in the Windows operating system.

Note: Windows programs that call these MIDI and digital audio functions in the Windows operating system should include the header MMSYSTEM.H (and MMREG.H for Win32), and also be linked with the library WINMM.LIB (or MMSYSTEM.DLL if a Win3.1 program).

Windows' lists of devices

In any computer, there can be more than one installed card capable of inputting or outputting MIDI data. Likewise, there can be more than one installed card capable of recording/playing digital audio data. (Needless to say, each card will have its own driver installed). For this reason, Windows maintains a list of all of the installed devices that support MIDI and Digital Audio. The list of devices

that support MIDI is a separate list from the list of devices that support Digital Audio. So, if one card happens to support both Digital Audio and MIDI, a name for it will appear in both lists. In fact, Windows differentiates between devices that support MIDI input and devices that support MIDI output, and maintains separate lists for each. So, if one card happens to support both MIDI input and output, a name for it will appear in both lists. Likewise, Windows maintains separate lists for devices that record Digital Audio as well as devices that play Digital Audio.

Therefore, Windows maintains 4 lists:

1. Devices that can input (or create) MIDI data.

2. Devices that can output MIDI data (or play it upon some sort of built-in sound module).

3. Devices that can record (or create) Digital Audio data.

4. Devices that can output (ie, usually play) Digital Audio data.

For a card that can input/output MIDI data, as well as record/play Digital Audio data, a name for it will appear in all 4 lists.

What name for a device appears in each list? Well, that is entirely up to the card's device driver. It is the card's device driver that tells Windows into which lists Windows should place the card's name, and what name to use for each list. And in fact, if the card has several different components that can logically fit into one list, then the device driver may tell Windows to put several component names into that list.

Let's take an example for illustration: A Creative Labs' AWE32 sound card. This card has the following components:

1. An ADC that can record stereo Digital Audio.

2. A DAC that can play stereo Digital Audio.

3. A MIDI In port that can input MIDI data.

4. A MIDI Out port that can output MIDI data.

5. An FM Synth sound module that can play MIDI data.

6. A Wavetable Synth sound module that can play MIDI data.

Notice that the last 3 components can all fit into the list of devices that support outputting or playing MIDI data. So, the AWE32 driver is going to tell Windows to put 3 names into that one list. For example, the driver may tell Windows to put the name "AWE32 MIDI Out" into the list. The driver

may tell Windows to also put the name "AWE32 FM Synth" into that same list. Finally, the driver may tell Windows to also put the name "AWE32 Wavetable Synth" into that same list. Although these 3 components are all on the same card, to Windows, they represent 3 individual devices that are capable of outputting/playing MIDI data, and Windows treats them as if they really are 3 separate devices. (Incidentally, there is no particular rules about naming the components. I just arbitrarily picked the above 3 names, although they do make sense. A manufacturer can put any names he wants into his driver for inclusion into those Windows lists).

Of course, the AWE32 driver will also tell Windows to put the name "AWE32 Digital Audio In" into the list of devices that support recording Digital Audio. The driver will also tell Windows to put the name "AWE32 Digital Audio Out" into the list of devices that support playing Digital Audio. Finally, the driver will also tell Windows to put the name "AWE32 MIDI In" into the list of devices that support recording or creating MIDI data.

Windows assigns a **Device ID** (ie, merely an unsigned long numeric value) to each name in its lists. The first device in each list gets a value of 0, and the subsequent devices have increasing values for their IDs. (ie, The second device in a list has a Device ID of 1, the third device has an ID of 2, etc).

So in our above example, the 3 components that the AWE32 device driver told Windows to add to the list of devices that support playing MIDI data ("AWE32 MIDI Out", "AWE32 FM Synth", and "AWE32 Wavetable Synth") will have Device IDs of 0, 1, and 2 respectively (assuming that they were added to the list in that order and are the first 3 names to be added to the list).

In fact, drivers for MIDI interfaces that have multiple MIDI Outputs typically add a separate name to the list for each MIDI Output, for example "MIDI Out #1 for Brand X card", "MIDI Out #2 for Brand X card", etc.

The one caveat to this is the "MIDI Mapper" device driver which is part of Windows. This doesn't appear in the list of devices that support outputting MIDI data, although the MIDI Mapper is indeed a device to which your program can output MIDI data. MIDI Mapper always has a Device ID of -1.

For a discussion of writing a program that properly shares MIDI inputs and outputs, see Managing MIDI Ports.

Low level API

The "low level" Windows functions are called that because they require that your program does a lot of the work of recording/playing MIDI and Digital Audio data, short of actually manipulating the sound card hardware.

For example, if you want to play a MIDI file (ie, play a musical sequence), then you have to load the MIDI data contained within that file, use a (Windows software) timer to determine when it is time to output each MIDI message, and when that time occurs, pass that message (ie, its data bytes) to a Windows function that actually outputs the bytes.

If you want to record a MIDI file, then you have to tell Windows to notify you when the card's driver has input the next MIDI message and to request Windows to pass you those MIDI data bytes, use a (Windows software) timer to timestamp that MIDI message, and write the bytes to disk (typically in MIDI File Format).

Because your program does most of the work, you have a lot of control over the playback/recording process, such as the ability to vary tempo, or perform filtering/mixing/transposing/etc of data on-the-fly.

For a more indepth discussion of writing a program that uses the Low level MIDI functions, see Low level MIDI API.

There are also low level functions for Digital Audio. Again, these require that you do a lot of work to record/play the digital audio data.

For example, if you want to play a WAVE file (ie, play digital audio), then you have to load the digital audio data contained within that file in "blocks" (ie, usually filling a buffer of 4K of data at a time), and feed it to a Windows function that actually outputs the data. In order to get a smooth, continuous playback of digital audio, you typically have to do double-buffering (ie, while the card is playing back one buffer of 4K data, you need to be simultaneously filling a second buffer with the next 4K of data).

If you want to record a WAVE file, then Windows has to feed you a continuous stream of those blocks of data, and you must write each block to disk (typically in WAVE File Format). In order to get a smooth, continuous recording of digital audio, you typically have to do double-buffering (ie, while the card is recording one buffer of 4K data, you need to be simultaneously writing the buffer containing the previous 4K of data to disk).

For a more indepth discussion of writing a program that uses the Low level Digital Audio functions, see Low level Digital Audio API.

High level API

The "high level" Windows functions are called that because they relieve your program of a lot of the burden of recording/playing MIDI and Digital Audio data.

For example, if you want to play a MIDI file (ie, play a musical sequence), then you merely call one Windows function, specifying the name of the MIDI file to be played back. Windows will open the MIDI file and load the MIDI data contained within that file, use a (Windows software) timer to determine when it is time to output each MIDI message, and actually output the bytes. In other words, Windows plays the MIDI file in the background while your program can go on to do other things simultaneously.

If you want to record a MIDI file, then you merely call one Windows function, specifying the name

of the MIDI file to be created. Again, Windows does all of the work of reading the incoming MIDI data, time-stamping each message, and writing the data to disk in MIDI File Format. (NOTE: At this time, Windows does not offer High level API support for MIDI recording).

But because your program doesn't do most of the work, you lose a lot of control over the playback/recording process, such as being able to perform filtering of data on-the-fly, etc. You're pretty much limited to being able to stop, start, pause, rewind, and fast-forward the playback.

For a more indepth discussion of writing a program that uses the High level MIDI functions, see High level MIDI API.

There is also a high level API for playing/recording WAVE files. Just like with the High level MIDI API, Windows does a lot of the work (ie, completely loads and plays the WAVE file in the background), but again, you lose a lot of control over the actual playback (for example being able to do on-the-fly mixing of digital audio -- to implement virtual tracks. For a more indepth discussion of virtual tracks, see the article Digital Audio on a computer).

For a more indepth discussion of writing a program that uses the High level Digital Audio functions, see High level Digital Audio API.

A new, "inbetween" MIDI API

A further option for MIDI sequencing (ie, the playback of musical excerpts using MIDI events) under Win95/98/ME/2000/XP and WinNT (4.X and above only) is to use the Stream API. This allows you to store a MIDI message in a special structure that contains a field that specifies the amount of time to delay before outputting the MIDI message. Windows does the actual timing of the MIDI playback. Unlike with the High level MIDI API, you still load the MIDI data and feed each message (or more likely a block of messages) to Windows (ie, so you have more control over the playback on a per-message basis), but you no longer have to time the playback. In other words, the Stream API is useful for playing back musical sequences where you still have a lot of control over individual events and can therefore perform various filtering and modification of MIDI data on-the-fly, but don't want to deal with timing issues (which is particularly aggravating under Win95/98/ME due to the timing discrepancies of 32-bit MultiMedia Timer callbacks).

Currently, the Stream API supports playback only (not recording -- ie, it can't supply you with time-stamped incoming MIDI messages). So, if you need to do MIDI recording, you'll have to use the Low level MIDI API and a MultiMedia Timer, or use DirectMusic.

For a more indepth discussion of writing a program that uses the MIDI Stream functions, see MIDI Stream API.

A new, fancier Digital Audio API

Many game programs use lots of digital audio for sound effects, and various waveforms often have to be mixed on-the-fly. For this purpose, Microsoft devised a new set of functions called "DirectSound" (ie, part of DirectX) under Win95/98/ME/2000/XP and WinNT (4.X and above only). These functions let the operating system take care of mixing several waveforms to a sound card's stereo DAC.

DirectX versions prior to 6.1 support playback, but not recording.

A card's device driver should have extra support to work well with the DirectSound API (although older drivers not directly supporting DirectSound may work with this API albeit very inefficiently and/or with some features disabled).

For a more indepth discussion of writing a program that uses the DirectSound functions, see DirectSound API.

A new, less archaic MIDI API

Many game programs also use MIDI for background music. For this purpose, Microsoft devised a new set of functions called "DirectMusic" (ie, part of DirectX) under Win95/98/ME/2000/XP and WinNT (4.X and above only). These functions are a bit newer than the older low level MIDI API, and address some of the oddities/discrepancies that plague the legacy MIDI API.

DirectX versions prior to 6.0 do not include DirectMusic, and versions prior to 6.1 support only playback.

A card's device driver should have extra support to work well with the DirectMusic API (although older drivers not directly supporting DirectMusic may work with this API albeit very inefficiently and/or with some features disabled).

For a more indepth discussion of writing a program that uses the DirectMusic functions, see DirectMusic API.

Miscellaneous Functions (ie, the Mixer API, and Aux API)

The Mixer API has functions to get a listing of all of the various inputs and outputs on a particular card (ie, how many Aux inputs, Line inputs, Mic inputs, Line outputs, etc, it has), and to adjust all of the volumes of these various components, as well as perform other operations such as muting inputs. This is a new API added to Win95/98/ME/2000/XP and WinNT (4.X and above only).

A card's device driver needs extra support to work with the Mixer API. Not all Windows drivers have this support. Win3.1 drivers do not.

For a more indepth discussion of writing a program that uses the Mixer functions, see Mixer API.

Older drivers should at least support the Aux API. Just like with the lists for devices that support MIDI input and output, and Digital Audio playback and recording, Windows maintains a list of all devices that have Aux outputs. (And again, a card's device driver instructs Windows to put some name into this list). The Aux functions allow a program to control the volume (and perhaps other settings) of each Aux output. Typically, audio from a CDROM or the audio output of a voice modem is connected to a sound card's Aux output. (ie, So, it's the volume of these that you're affecting).

For a more indepth discussion of writing a program that uses the Aux functions, see Aux API.

MultiMedia File functions

When programs use the low level MIDI and Digital Audio API, they typically have to open, read, and write MIDI and WAVE File Format files. To help parse Interchange File Format types of files (ie, files whose data is arranged into nested chunks), Windows has a MultiMedia File (ie, Mmio) API.

For a more indepth discussion of writing a program that uses the MultiMedia File API, see MultiMedia File API.

Misc Topics

The following online Windows book is available for download. It gives more information about the Windows MIDI and digital audio functions (as well as Aux API and MultiMedia Timer API). But note that this book dates back to Windows 3.1, so some newer features, APIs, and notes may be omitted. Use the tutorials on this site for more uptodate information.

Windows MultiMedia System Book

For a more indepth discussion of the history of why and how the Windows MIDI and digital audio functions came to be, read the introduction to the article Audio cards and MIDI Interfaces for a computer.

Introduction

Sometimes, the cause of internal IBM PC cards not working is the result of conflicts with IRQ (Interrupt Request Line), I/O (Input/Output) Addresses, or DMA (Direct Memory Access) channels. Two (or more) cards may be trying to simultaneously use the same IRQ, I/O address, or DMA channels. The net result is that the cards interfere with each others' operation **at the hardware level**, typically making the cards uncontrollable by drivers and software. These hardware conflicts need to be corrected before drivers and software programs will work with the cards.

Most motherboards also have built-in components that use IRQ, I/O addresses, and even DMA channels. For example, IRQ 7 is used by the parallel port upon most motherboards. So, you must make sure that your card isn't set to some resource that is also being used by a component upon your motherboard.

One good sign that you may have an IRQ conflict with your sound/MIDI card is when MIDI playback works fine, but MIDI recording doesn't. Many cards do not use the IRQ for playback (ie, polled output) but most all do use the IRQ for recording (ie, interrupt-driven input).

Plug and Play (PnP)

The very first IBM PC cards that came out were designed to be plugged into ISA bus slots. Later, PCI and AGP bus slots were added to motherboards for newer PCI and AGP cards.

The ISA bus is much more primitive than PCI and AGP, and therefore the configuration of ISA cards is less automatic and more troublesome. Typically, you have to manually set "jumpers" or DIP switches on an ISA card to determine the IRQ, I/O address, and DMA channels that you wish the card to use.

On the other hand, most all PCI and AGP cards have a feature called **Plug and Play** (ie, PnP) that lets the computer itself configure the cards to resolve all hardware conflicts. So you don't have to manually set any jumpers or DIP switches for PnP cards, nor inevitably worry about hardware conflicts... at least in theory. This assumes that the card wasn't cheaply made to support only a limited set of the possible IRQ numbers. (ie, Some manufacturers will make PnP sound cards which only allow a choice of the most common IRQs used by sound cards; 5, 7, or 9. They do this to save a few pennies. But this can be really bad if you happen to have a few other cards that only support these same few IRQs such that there aren't enough IRQs to dole out among the cards). Furthermore, some older motherboard BIOSes have bugs in their PnP code such that conflicts aren't properly resolved. (In this case, either update your BIOS, or if using an OS that supports PnP, such as Win95/98/2000, see if you can disable your BIOS handling of PnP and let the OS setup the cards' IRQs and base addresses). So, it's still possible (but rare) to see hardware conflicts with PnP cards.

Some companies have tried to make ISA cards that allege to be PnP, but ISA cards don't really support PnP. An ISA card that alleges to be PnP may or may not work properly with real PCI PnP cards. In my experience, they don't, and should be treated as non-PnP cards.

Hardware conflicts are very common if your sound/MIDI card is not PnP (or if you have other types of cards in your computer which are not PnP). Non-PnP cards may interfere with the operation of PnP cards (or vice versa) if you don't properly setup your computer's BIOS and Windows' Device Manager. So, unless your computer system has all PCI (and AGP) cards in it and does not have **any** ISA cards, then you should definitely check for hardware conflicts.

Checking for conflicts

For non-PnP cards, do physically inspect the cards to make sure that every one is set to a different IRQ and base address. I can't possibly stress this enough. If you're not sure what IRQ's the non-PnP cards in your system are using, then open up the case and take inventory. (I urge anyone who buys a pre-built system to insist, **when you buy the system**, that the vendor provide you with a list of any non-PnP devices in the system indicating which IRQs, base addresses, and DMA channels are used by those devices. Demand it. And then make sure you do get it). Apparently, there are lots of people who enjoy playing what I call "the IRQ guessing game". Without knowing what IRQs, base addresses, and DMA channels are in use in his system, this person will arbitrarily set the jumpers on a non-PnP card being added to his system, install the card, and then hope that when he boots up everything will magically work. It usually doesn't. In fact, by the time that the person has played this game for a suitably long time, monkeying around with drivers, he has really messed up his entire software installation. It isn't worth doing that when, if you have a list of what devices use which resources in your computer, changing or adding hardware to a PC is typically a very simple, quick, trouble-free process. I've done this a lot to my own system and others' systems. (You know all those horror stories about installing PC cards? They come from people playing the "IRQ guessing game").

If you have non-PnP cards, don't rely upon Win95 (or any other OS) telling you if you have an IRQ or base address conflict. The ISA bus does not support determining this, so OS's (and utilities such as MSD.EXE) have to perform a sort of "guessing game" of their own, assuming certain "standard and/or typical" IRQ settings, and then checking for anything that looks like a deviation from this.

In order to unequivocally determine what IRQs, base addresses, and DMA channels are in use on non-PnP devices, you need to know the jumper/DIP settings of your cards. This includes all of your ISA cards as well as any non-PnP devices on your motherboard. IRQ conflicts appear to be the biggest cause of conflicts. Base address and DMA conflicts aren't nearly as common (but if you have 2 or more of the same type of card in your system, for example, 2 MIDI/audio cards, the chances of base and DMA conflicts increase dramatically as many manufacturers of the same types of cards follow each others' "standards" for IRQ/address/DMA settings. For example, most MIDI cards use the Roland MPU-401 standard I/O address of 330). Here's a list of what I've found to be the most common IRQ assignments. I think that it's pretty safe to assume these, but beyond the standard devices (ie, clocks, keyboard, serial + parallel ports, HD and floppy controllers, math chip), check all of your other devices:

```
IRQ #
0       System timer
1       Keyboard
2       Cascade for second interrupt chip
3       Serial (COM) port 2 (often a modem is attached to this)
4       Serial (COM) port 1 (usually a serial mouse is attached to this)
5       Parallel port 2 (not often used)
6       Floppy controller
7       Parallel port 1 (usually a printer is attached to this)
8       Realtime clock
9       free (but some video cards may use this for EGA emulation)
10      free
11      free
12      free
13      Math chip
14      HD (IDE) controller
15      Second IDE controller (usually a CD-ROM is attached to this. This second
        IDE controller may even be on your sound card, such as an SB card)
```

Typically, sound cards are set to use IRQ 5. This is often a good choice. Usually, IRQs 10, 11, and 12 are safe to use (assuming other ISA cards aren't using such). If the card has a MPU-401 compatible MIDI port, it usually uses I/O addresses 330 and 331 (hex). The MIDI port may also use another IRQ such as 9, separate from the IRQ used by the digital audio portion of the card.

Checking for conflicts using Win95/98/2000 Device Manager

Win95/98/2000 are PnP aware operating systems. This means that they can not only determine (and display for you) what IRQs, I/O addresses, and DMA channels are in use by all of your PnP cards, Win95/98/2000 can also let you change what IRQs, I/O addresses, and DMA channels are used by each PnP card. You may be able to easily resolve any conflicts just by using the Win95/98/2000 utility that controls IRQ, I/O Address, and DMA channel assignments. That utility is the **Device Manager** page of Control Panel's **System** utility.

Here's what you do to get a listing of what IRQ's are being used for what on your system. Go into Control Panel, run the System utility, and turn to the Device Manager page. Click on the Print button. Ask for a "System Summary". You'll get a listing (actually a .prn file that you must print out) that has a section which on my system looks like this:

```
******************** IRQ SUMMARY ********************

IRQ Usage Summary:
  00 - System timer
  01 - Standard 101/102-Key or Microsoft Natural Keyboard
  02 - Programmable interrupt controller
  03 - Sportster 28800 Internal
  04 - Communications Port (COM1)
  05 - Printer Port (LPT1)
  06 - Standard Floppy Disk Controller
  08 - System CMOS/real time clock
  11 - Adaptec AHA-2940U/AHA-2940UW PCI SCSI Controller
  13 - Numeric data processor
  15 - Diamond Stealth 64 Series (Diamond GT)
```

This shows me what devices are using which IRQs. For example, IRQ 11 is being used by my Adaptec SCSI card. My Diamond video card is using IRQ 15. You'll notice that IRQ 12 isn't shown in this above list. That's because no device is using that. So if I were to add a new non-PnP card to my system, I'd set it to use IRQ 12 for example. (If you have some device that is using an old Win3.1 driver rather than a newer Win95-aware driver, it may not show up in the above listing. For example, my RAP-10 Sound Card has an old Win3.1 driver. It doesn't show up in the above list even though I have set it to use IRQ 7. So although IRQ 7 is not shown as being used in the above list, it really is. That's why it's so important to have all Win95-aware drivers on your system. Otherwise, Win95 may not be able to properly setup PnP peripherals because it may not know that some device is using an IRQ. Always update all drivers and avoid using Win3.1 drivers on your Win95 system, especially if you have PnP cards. If you must use a non-PnP card with a Win3.1 driver, then at a minimum, go into your BIOS' PnP configuration screen, and set the IRQ used by that card to be reserved for a "legacy ISA device").

If you've got a conflict, you'll typically see two cards listed using the same IRQ. For example, here's a conflict between my printer and sound card:

```
****************** IRQ SUMMARY ******************

IRQ Usage Summary:
  05 - Printer Port (LPT1)
  05 - Jeff's Sound Card
```

You may also want to check the IO Port Summary to make sure that no 2 cards have address ranges that encompass the same values.

Incidentally, that small yellow exclamation mark (or red **X**) you see next to a device name in "Device Manager" indicates a conflict (usually IRQ) or that something is wrong with the device's response. (ie, For example, I use SCSI devices rather than IDE, so I disabled my IDE ports. My PnP BIOS told Win95 that I had IDE ports, but Win95 didn't get any response from them, hence that exclamation mark next to my "PCI IDE controller". In fact, it's common to see exclamation marks related to PnP ISA cards. Often the devices appear to have several "phantom" units, each one dedicated to whatever IRQs the device supports, and Win95 thinks that these are real yet gets no response from them. Hence you may see several instances of a particular PnP ISA card listed, with all but one having an exclamation mark. It's OK to leave these phantom devices, and if Win95 ever tells you it detected new hardware the next time you boot, tell Win95 to ignore each one of those phantom devices as it pops up a dialog for each, asking you to choose a driver to install. Better yet, select the Properties of each one of these Phantom devices, and remove the checkmark next to "Original Configuration". This helps free up that wasted IRQ for another PnP card which will really use it, rather than simply reserve it for no good reason).

NOTE: If you have Windows 98/2000, a new feature was added to Device Manager which pretty much displays all of the info that you saved in that .prn file produced by Device Manager's Print feature. Just double-click on the "Computer" icon in Device Manager. This will bring up a dialog box that can show lists of which devices are using which IRQ numbers, I/O addresses, and DMA channels.

Solving conflicts

The way to solve hardware conflicts depends upon whether the cards in conflict are all Plug and Play (PnP), or whether one or more of them is a non-PnP card.

If the two, conflicting cards are PnP, then you're in luck. All of the hardware conflicts can be resolved just by using Windows' Device Manager. You won't even have to open your computer case. (But you should make sure that you have the latest version drivers for those cards. Sometimes, updated drivers solve what appear to be hardware problems with PnP cards). Merely go through Device Manager's IRQ, I/O Address, and DMA channels lists to note any resource conflicts, and then use Device Manager to change the Properties of any conflicting cards and set them to use unique IRQ, I/O Address, and DMA channels. For PnP cards, you can change a device's IRQ or I/O Address Range by clicking on the device's name in the Device Manager list, clicking the Properties button, and turning to the Resources page. Uncheck the "Use automatic settings" box. Now you can click on any individual "Interrupt Request" or "Input/Output Range" setting to change its IRQ or I/O Address respectively. Click on the "Change Setting" button to pick a new value for that setting.

If one of the conflicting cards is PnP, and the other is non-PnP, then you may be able to resolve the conflict by adjusting your computer's BIOS PnP configuration. You should reserve any IRQs used by non-PnP cards -- setting each of those IRQs to be reserved by a "legacy ISA device". At worst, you'll have to open the computer case and change jumpers or DIP switches on the non-PnP card.

If both cards are non-PnP, then you'll have to open the computer case and change jumpers or DIP switches on one of the cards.

In conclusion, here is a list of things to do to solve hardware conflicts:

1. If you have non-PnP cards in your computer, first make sure that you know what IRQ, I/O Addresses, and DMA channels (if any) that these cards are using. Don't depend upon Windows to be able to tell you. If you don't know, physically inspect the card's jumpers or DIP switches. Write this information down.

2. If you have PnP cards in your computer, make sure that your motherboard's BIOS is a PnP-aware BIOS, and

make sure that it's the latest version of that BIOS. Check the web site of the motherboard manufacturer for any BIOS update for your motherboard. They may have an updated BIOS to download, plus a utility that installs it, and instructions how to install it.

3. If you have non-PnP cards in your computer, go into the BIOS setup screen. (This is usually done by pressing the *Delete* key while the computer is just starting to boot). Go to the **PnP configuration** page. Choose a manual setup (rather than an automatic one). For every IRQ that is being used by one of those non-PnP cards, set that IRQ to be reserved for a "legacy ISA device" (rather than used for "legacy ISA/PCI"). Note that your mouse may not work in selecting items in the BIOS setup, so you typically have to use the arrow keys to move between selections, and the Page UP and Page DOWN keys to change the value of a selection.

4. Use a PnP-aware operating system such as Windows 95/98/2000. Use Device Manager to check for any hardware conflicts, and if any of the conflicting cards are PnP, use Device Manager to change the resources so that conflicts are resolved.

5. Make sure that you have the latest versions of all drivers for your cards. Check the web site of each card's manufacturer for any driver update for that card. They may have an updated driver to download, plus instructions how to install it.

Driver issues

Another, even more common problem concerns software drivers. The fact of the matter is that programmers **do** write buggy software, and there's a chance that any problem which looks hardware-related may actually be due to some bug in the card's driver. Check with the card's manufacturer that you have the latest drivers. Ask if there have been any problems reported that may be applicable to your own setup.

And definitely don't rule out the possibility that you may have configured the driver's setup (ie, "Properties" in Microsoft-speak) incorrectly. If you use Windows 95/98/2000, read the FAQ "MIDI/Audio under Win95/98/2000".

This article answers questions about MIDI and audio setup under Windows 95/98/ME/2000/XP. For more general questions/answers about MIDI/audio/computer setups, and trouble-shooting, read the FAQ "MIDI connections and computer setup".

If you're having a problem with an internal IBM PC card not working, then you should first check for any hardware conflicts in your system. Read the article "Resolving hardware conflicts" for more information.

Questions in this FAQ are:

"How do I install/setup an audio/MIDI driver?"
"How do I remove an audio/MIDI driver?"
"Where is Win95's MIDI Mapper?"
"Why didn't Windows autodetect my second sound card?"
"How do I get my 16-bit sequencer program to have more stable digital audio under Win95?"
"Why can't I get sound from my MS-DOS game programs run under Win95?"
"What is that yellow exclamation mark in Device Manager?"
"Why won't my Windows 3.1 or Windows 95/98/ME driver work under Windows NT/2000/XP?"

How do I install/setup an audio/MIDI driver?

Nowadays, most driver packages ship with a program you can run which installs the driver for you. Typically, this program is named *setup.exe*. There will often be numerous other files (in the package) along with setup.exe, but these are used only by setup.exe to install driver support. (Once the driver support is installed, you can delete setup.exe and all other files in the package, or save them if you need to reinstall). If the driver package is shipped on a CDROM, the CDROM will typically have a file named *autorun.inf* on it. All autorun.inf does is automatically run setup.exe when you insert the CDROM into the drive. (This is a function of Windows' autorun capability. Unless you have this feature disabled, Windows will "autorun" any CDROM with an autorun.inf file on it).

Setup.exe will automatically detect what operating system is on your computer, and install the correct driver. (ie, Many maunfacturers will ship one CDROM containing drivers for several versions of Windows. But not every driver is compatible with all versions of Windows, so the correct one must be installed for your system).

If the driver does not ship with its own setup program, then you will have to manually install the driver yourself. (Some manufacturers farm out their driver development to programmers who don't know how to, or don't take the time to, write a self-installing driver package).

To manually install a driver, the driver should have a .INF file (ie, created by the manufacturer and shipped with the driver). The INF file (ie, the filename ends with a .INF extension) is really just a text

file that tells Windows what kind of driver is being installed (ie, digital audio, MIDI, video, printer, etc). It also contains the text that Windows uses when it displays the "name" of the driver (ie, not the driver filename, but rather, a more meaningful name which I'll call the "device name") in various displays, such as in the Control Panel's MultiMedia notebook. The INF also lists what the driver filenames are so that Windows knows what files to copy off of the install disk. You can view this INF file in a text editor (and even edit the device name). For example, the Roland RAP-10 INF indicates that the driver supports digital audio, MIDI, and LINE IN (ie, aux). Windows treats these 3 "sections" of the card as if they were 3 separate devices. The device name for each of these sections of my RAP-10 is "Roland Audio Producer". Windows ships with several audio/MIDI drivers for various cards and the INF files (and drivers) for those are included with Windows.

Note: Before installing new MIDI/Audio drivers, I recommend that you first manually remove any currently installed drivers that the new drivers are meant to replace, unless the new drivers are written to be installed over those old drivers. Usually, if you're updating new drivers over old drivers written by the same manufacturer for the same card, then there's no need to remove the old drivers first. But, if you had also changed your sound card, or gotten drivers from a different source, then I recommend removing the old drivers. For example, here's how I would go about changing to a different sound card and installing new drivers:

1. Boot up your system with its current setup (ie, with your current sound card and drivers).
2. Remove the current drivers. See "How do I remove an audio/MIDI driver?".
3. ShutDown Windows and turn off the computer.
4. Install the new sound card.
5. Boot up the system. If Windows doesn't autodetect the new card (which Windows won't if the card's drivers weren't included with Windows itself, or if the card isn't Plug-and-Play), then proceed to manually install the new drivers.

Here are the steps to install a MIDI/audio driver:

1. Start the Control Panel's **Add New Hardware** utility.
2. Click **Next** to skip the intro screen.
3. Answer **No** to Windows asking to search your hardware, and click **Next**.
4. Move down the list until you see **Sound, video, and game controllers**. Click on (ie, highlight) that item, and click *Next*. Windows will now present a dialog that lists all of the audio/MIDI drivers included with Windows. In the box on the right is listed all of the manufacturers whose drivers are included with Windows. When you select a manufacturer name, the box on the left updates to display those included drivers for that manufacturer's products.
5. If the driver that you want to install is among Window's included drivers, and you wish to use that, simply highlight the driver name, and select OK. Now skip to the last step.

   On the other hand, if you prefer to use drivers other than the included ones (ie, you have updated drivers), or the driver you need isn't included with Windows, then click the **Have Disk** button instead.
6. Type in the directory where the driver INF file can be found, and click **OK**.

7. If Windows finds the INF file, it will present a window containing the device name. Highlight that name, and click **OK**.

   Note: If the driver package contains drivers for various versions of windows, or various models of cards, then there may be numerous INF files in the package. You may have to search around, selecting various INF files, until you find one that lists the desired model.

8. Windows will then install the driver, and run whatever setup routine is contained within the driver. Typically, the setup routine will open a window and allow you to specify such things as port address, IRQ number, and DMA channels used for digital audio (if supported). Each driver has its own proprietary setup routine, so you'll have to deal with whatever prompt/window pops up at this point.

After you've installed the driver, you can now configure Windows' (and ultimately, all Windows applications') use of the driver. For example, do you want all Windows programs to send their MIDI output to this driver, and get their MIDI input from this driver? Do you want all programs to use the card's ADC (assuming it has such) to record digital audio (usually stored in a WAVE file)? Do you want all programs to use the card's DAC to play digital audio? You use the Control Panel's **MultiMedia** notebook to configure the card's use. Open the MultiMedia notebook. It has several pages that you can flip to (ie, the tabs are at the top of the notebook).

The first page is called **Audio**. This page concerns playing and recording digital audio. The page is divided in half, with the *Playback* settings on top, and the *Recording* settings below. If your card has a Digital to Analog Converter (ie, for playback) and Analog to Digital Converter (ie, for recording), then your card's device name should be listed in both the dropdown list in the Playback section of the page as well as the dropdown list in the Recording section. Select the device name so that it appears as the "Preferred Device" for both Playback and Recording. Now all Windows apps (as well as Windows itself when it plays "system sounds") will use your card for digital audio playing and recording. For cards that support various sample rates and bit resolutions, you should be able to select a "Preferred quality" for recording. You can also set default values for playback and recording volume. The "Use preferred devices only" setting is only effective if you have more than one digital audio device (and driver) installed. When selected, Windows forces all programs to use just the one "preferred device". When not selected, if some program has the preferred device in use, and another program wants to do something with digital audio, Windows will automatically have the second program use some other digital audio device in your system. Note that it's perfectly OK to have different cards chosen for playback and recording, which is one way to get around the problem of not having a full duplex sound card (ie, one that can record digital audio at the same time that it is playing back previously recorded digital audio).

The above settings don't mean that all programs <u>must</u> use the "preferred device" for digital audio, with those volume and rate/resolution settings. Rather, these are default values (for programs that don't offer ways to change such settings). On the other hand, a Windows program could query all of the digital audio devices installed on your system assuming that you had more than one installed), and let you choose to use any and all devices, maybe even simultaneously for many tracks of digital audio. The program could provide controls with which you adjust volume and rate/resolution, and maybe even additional parameters. But such a program would have to be written to use Windows' MCI API in a fancy manner. Your average CD-ROM Encyclopedia, and other titles that aren't geared

for fancy audio work, generally use the preferred device and default settings verbatim.

There is another page in the MultiMedia notebook for **MIDI**. This page concerns MIDI input and output. If your card has MIDI input and output ports, then your card's device name should appear in the list below "Single Instrument". If you want to set all 16 MIDI channels to come in and go out through that card's MIDI IN and OUT, then select "Single Instrument" (ie, make sure that the dot is highlighted), and then highlight the device name (ie, so that it also appears in the box beneath "Single Instrument").

Note: In Windows NT/2000/XP, the MIDI page is missing. Instead, only the one, preferred instrument for MIDI playback (ie, output) is listed under the "Audio" page. There is no facility for configuring any patch remapping via the MultiMedia notebook.

On the other hand, if you have more than one MIDI device installed, and you wish to divide up the 16 MIDI channels between those devices, then select "Custom configuration" instead. For example, assume that you have two cards, an MPU-401 and a Monterrey. You've installed drivers for both, and so there are 2 device names in the MIDI device list. You want to have all MIDI events on channels 1 to 8 go through the MPU-401, and all MIDI events on channels 9 to 16 go through the Monterrey. Click the **Configure** button, and a window will pop open listing all 16 MIDI channel numbers. Now follow these steps to setup MIDI channel 1.

1. Highlight number 1.
2. Click the **Change** button. Another window will pop open containing a dropdown list with all of the installed device names. (In this example, that's the MPU-401 and Monterrey device names).
3. Select the desired device name, and click *OK*. (In this example, we want the MPU-401).

You've now set MIDI channel 1 to go through the MPU-401. Repeat the steps with channels 2 to 8. Repeat the steps for channels 9 to 16, but select the Monterrey device name instead. (Note that you can select several channels simultaneously for changing, by holding the *SHIFT* key while highlighting channel numbers).

Now, by default, any Windows program that transmits a MIDI message on channel 10 will send that message to the Monterrey's MIDI OUT.

For multi-port (ie, multiple bus) MIDI interfaces, the driver for that device should make it look like each bus on that one card is itself a separate card. Therefore, you should have a unique "device name" for each port on that multi-port interface. For example, maybe port 1 will have a device name of "Brand X MIDI Port 1", whereas port 2 will have a device name of "Brand X MIDI Port 2", etc). Hopefully, the driver will also be "multi-client" so that, if desired, you can have different programs simultaneously doing MIDI data transfers to different outputs. (In that case, you'd have to setup "custom configuration" and restrict each program to using different MIDI channels than the other programs).

Note that System Exclusive, as well as System Realtime (ie, MIDI Clock/Start/Stop, MIDI Time

Code, etc) messages do not have a "MIDI channel" associated with them. I'm not sure how Windows handles such events given a custom configuration. It may simply output such events to whatever device you've selected for "Single Instrument" or may automatically duplicate all System Realtime, Common, and Exclusive messages to all MIDI devices.

Just like with the Audio settings, the above MIDI settings don't mean that all programs **must** use the "Single Instrument" for MIDI. Rather, this is the default MIDI IN and OUT (for programs that don't offer ways to change such). On the other hand, a Windows program could query all of the MIDI devices installed on your system (assuming that you had more than one installed), and let you choose to use any and all devices, maybe even simultaneously for many MIDI channels (ie, 16 per device). The program could provide controls with which you adjust which MIDI data goes to which device. But such a program would have to be written to use Windows' MCI API in a fancy manner. In fact, CakeWalk Pro does exactly that, which is how it supports MIDI interfaces with multiple MIDI busses. Your average CD-ROM Encyclopedia, and other titles that aren't geared for fancy MIDI work, generally use either the "Single Instrument" (ie, one MIDI card) or for "Custom configuration", will automatically have the MIDI data routed to various cards as per your channel/device assignments.

How do I remove an audio/MIDI driver?

Never, never, never manually delete driver files by dragging icons to the Recycle Bin nor typing "del" from a command prompt. That's not the way that you uninstall a driver. There's a difference between uninstalling a driver and deleting files on your HD. If you do the latter, then you'll likely still leave behind references (to the now-missing driver files) in Windows' registry and INI files. This can cause all sorts of error messages to pop up. And if you later install other drivers without removing those references, you could end up with a completely confused set of error messages and weird behavior.

The proper way to uninstall a driver is as follows:

1. Open Control Panel's "System" notebook.
2. Flip to the "Device Manager" page. There you'll see a listing of all of the hardware in your computer, separated by category.

   Note: In Windows NT/2000/XP, Device Manager is located by flipping to the "Hardware" page and then clicking the "Device Manager" button.
3. You should see a category for sound and gameport devices with a small "+" sign next to it. Click on the + sign and this should drop down a list of the installed audio cards on your system. (Alternately, some sound cards list their driver components under their own category).
4. Highlight (ie, click on) the name of whichever component you wish to remove.
5. Click the "Properties" button.
6. Click the "Remove" button.

If your audio device isn't listed on System notebook's "Device Manager" page, then you'll have to open the MultiMedia notebook and flip to the "Advanced" page. This is like the Device Manager listing, but it only shows MultiMedia components broken down into smaller catagories. You'll need to go through each sub-category (ie, "Audio Devices", "MIDI Devices and Instruments", etc) to remove each component that pertains to the device that you wish to remove.

I have an external MIDI module (ie, a Roland JV-80) that doesn't have a General MIDI patch set, nor drum note assignments per GM. Whenever I use the software mixer in my sequencer to select a patch, it always selects the wrong patch because it expects a GM patch set. In Win 3.1, I could use the MIDI Mapper to remap the GM patches and drum note assignments so that they selected the proper patches and drum sounds upon my JV-80. Where's the MIDI Mapper for Win95?

Microsoft didn't have time to finish rewriting the MIDI Mapper for Win95 before the OS's release. Subsequently, MS created the IDFedit.zip utility which you can download from my Web Site now. This utility can be used to create an Instrument Definition File (ie, IDF) in which you determine how the GM patch set and drum note assignments are remapped. Then, you install this IDF as if it were another MIDI device upon your system, and use it for MIDI IN and OUT.

For example, let's assume that the very first GM patch "Acoustic Grand Piano" is actually the second patch upon your JV-80 and you wish to remap GM patch 1 to JV-80 patch 2.

1. Run IDFedit.
2. Pull down the "Edit" menu and select "New instrument". This will create a new instrument with a patch set, drum note assignments, a "name" or "ID" (which defaults to "untitled1"), plus other settings. You should see "untitled1" appear in the list box.
3. Double-click upon "untitled1". This brings up the properties notebook with which you enter your settings.
4. First, you'll want to give this instrument a meaningful name. Flip to the "ID" page of the notebook, and type in your desired Instrument Name. For this example, we'll call it "JV-80 GM Patch Set".
5. Leave "Set as default" checked if you want Windows to apply these settings to the default MIDI IN and OUT. (You can always change this later via the MultiMedia notebook's MIDI page. In fact, all this checkmark does is automatically select this new instrument as if you had picked it out of the list below "Single Instrument" on the MIDI page).
6. Now flip to the "Patch Map" page. Here you see a listing of all 128 GM patches. Highlight the "Acoustic Grand Piano", and click the "Edit" button. A dialog appears asking you to which

patch to remap this GM Patch. Remember we said that the JV-80's Acoustic Grand Piano is #2, so type a 2 here, and press ENTER. You can then highlight other GM patches and remap them to equivalent patch numbers in your JV-80 patch set. You've now remapped GM to your JV-80. Whenever you use the "JV-80 GM Patch Set" as the output for your sequencer program, this remapping automatically occurs. (If you can't find a respective patch on your module for a given GM patch, then set the GM patch to the closest equivalent. For example, if you only have one piano sound whereas the GM set includes several piano variations, then set all of the GM piano patches to point to that one piano patch. ie, It's OK to set several GM patches to the same patch number).

7. You can flip to the "Percussion Map" page and reassign each GM drum note to a different MIDI note number in order to remap the drum note assignments.

8. Furthermore, you can flip to the "Key Map" page and remap all of the MIDI note numbers for the instrumental parts (ie, not the drum part, usually on channel 10). For example, if you add 12 to each note number in the list, all of the notes sent to the JV-80 will be transposed up an octave (except for the notes on the drum part's channel).

9. The "Channels" page allows you to select which channels are excluded from being affected by this IDF. This is useful if you want to apply different remapping (ie, different IDFs simultaneously) to different MIDI channels going through one MIDI card. For example, assume that you have a Proteus listening to MIDI channels 1 to 8, and a JV-80 listening to channels 9 to 16. These two units have non-GM patch sets, and are different from each other. They are both daisy-chained to one MIDI card in your computer. You'll want to create an IDF that remaps the Proteus patch set (and only affects channels 1 to 8), and another IDF that remaps the JV-80 patch set (and only affects channels 9 to 16). Then, you'll apply both IDFs to that one MIDI card. (In fact, you can create these two sets of IDF settings within one IDF file, by creating a "New Instrument" within IDFEdit, so that you actually have two instrument definitions with the one file. The second instrument will default to a name of "untitled2". That way you'll apply both remappings when you install just the one IDF, and this is more efficient).

    You can also choose which MIDI channel the "drum kit" is assigned to (which defaults to channel 10) in case you have a drum box set to a MIDI channel other than 10 and wish to apply the IDF drum note remapping to that MIDI channel.

10. The "Info" page simply lets you add more descriptive comments to your IDF.

11. You can click on the "OK" button when you're done with your settings. Then, pull down the "File" menu, select "Save As", and pick out a filename for your IDF.

Once you've saved your new IDF, you can "install" this new instrument.

1. Go to Control Panel and open the MultiMedia notebook.
2. Flip to the MIDI page.
3. Click on the "Add New Instrument" button. The "MIDI Instrument Installation Wizard" dialog appears.
4. Highlight the MIDI (hardware port) to which you want your IDF to be applied (ie, the device that you want to use for MIDI IN/OUT in conjunction with these IDF settings). If you only have one device, that is already highlighted.
5. Click on "Next".

6. Click on the "Browse" button and locate your new IDF. Select it. Its filename will now appear as one of the Instrument Definitions in the list.
7. Click "Next".
8. Now type in a descriptive name for this instrument. We'll just use "JV-80 GM Patch Set" again.
9. Click "Finish".

"JV-80 GM Patch Set" should now appear in the MultiMedia notebook's list of "Single Instruments". You can select this as if it were a device in your system, and when you do, Windows will apply the remapping to MIDI IN and OUT. This new "device" should also appear in CakeWalk's list of output devices.

You can add as many IDFs as you like, even applying more than one to a given MIDI card. (This may be useful if you have several non-GM multitimbral devices attached to a single MIDI interface. By using "Custom Configuration", and then selecting different IDFs for the various 16 MIDI channels, you can divide up the 16 MIDI channels between those instruments, as well as apply different remappings to each MIDI module. But if dealing with multitimbral instruments, you'll still have to set each instrument to actively ignore MIDI channels reserved for other modules).

If you wish to remove a particular IDF, you can do so. Let's remove our "JV-80 GM Patch Set" IDF.

1. Go to Control Panel and open the MultiMedia notebook.
2. Flip to the "Advanced" page (ie, the last page). This displays something that looks like Device Manager's list, except that it only contains devices pertaining to audio, MIDI, and video.
3. Click on the small + sign beside "MIDI Devices and Instruments". This will drop down a list of all of your installed MIDI devices.
4. There will another + sign next to the MIDI device which you've applied the IDF to. For example, let's assume that our IDF had been applied to the "Roland Audio Producer" device. Clicking on the + sign drops down a list of the IDFs that were applied to this device. You should see "JV-80 GM Patch Set" there.
5. Highlight this IDF.
6. Click on the "Properties" button at the bottom of the page. A new dialog should pop up containing information about this IDF.
7. Click on the "Remove" button. The IDF is no longer applied to the MIDI output, and has been removed from Win95's list of instruments. The actual IDF itself has not been erased. You can reinstall that file later if desired.

NOTE: Some sequencers support defining patch maps for each of your MIDI modules. (For more information, see [MIDI connection and computer setup](#)). This is more flexible than using an IDF because you can usually define more than 128 patches with the former (ie, the sequencer supports banks of patches whereas IDFs don't). Plus, the sequencer lets you apply any patch map to any track, regardless of MIDI channel, so it's a lot easier to setup.

Just don't use an IDF in conjunction with the built-in patch naming features of your sequencer. You don't want to remap your custom patch sets. The exception to this is if your sequencer completely

bypasses Win95's MultiMedia MIDI setup. Such a sequencer would use the "low level" MIDI API of Windows. In this case, you can still setup an IDF for the benefit of sequencers and multimedia software that uses the "high level" MCI API of Windows. But then, don't select the MIDI Mapper as the output driver for your software that has its own patch naming features.

I have both a Roland MPU-401 card and a Sound Blaster 16 card in my computer. When installing Windows, the "auto-detect" found my SB16 (and installed its drivers) but didn't find my MPU-401 card (nor install its drivers). Why did this happen, and how do I get my MPU-401 working?

Windows' "AutoDetect hardware" only does a search for one (non-PnP) sound card (since that's all that is really needed for normal use). As soon as it discovers one card, that's the end of the search for an audio card. Needless to say, the first card that Windows searches for is an SB compatible. I'm not surprised that it found your SB16 (and set that up) before finding your MPU-401.

Also, Windows is written to only check for those sound cards that have drivers that are shipped with Windows. If you have a card for which drivers were not included on your Windows CDROM (or floppies), Windows will not detect it (unless it's a PnP card. In that case, Windows will detect, but if there are no drivers, it will label it an "Unknown device" without any driver support).

In conclusion, Windows will not autodetect more than one non-PnP audio/MIDI card. For any second card, you'll have to manually install the drivers. Windows will also not autodetect any card that it doesn't know anything about. (ie, Windows will not autodetect a card for which drivers aren't included with Windows itself). You'll have to manually install the drivers for this case too. This is easy to do, and is explained in "How do I install/setup an audio/MIDI driver?".

When using my 16-bit Windows sequencer software under Windows, I get erratic digital audio playback. Sometimes, there's static or tempo glitches. Why does this happen, and how can I fix it?

Most of the problems with old (16-bit) software playing digital audio tracks (ie, recording to, or playing from, the hard drive) are due to the differences between Win3.1's and Win95's disk cache handling and virtual memory. 16-bit Windows software is written for Win3.1, and consequently may not be designed to perform well with Win95's different methods. You can minimize adverse effects Win95 may have on 16-bit software by forcing the disk cache and virtual memory (ie, swap file on

your HD) to fixed sizes.

To set Win95's disk cache to a fixed size, load the file SYSTEM.INI (found in your Windows directory) into NotePad. Find the line that reads [vcache]. Alter the MinFileCache and MaxFileCache lines below it (or add these 2 lines if they aren't already there). If you have a system with 8 MEG of RAM, set both to 1024. If you have a system with 16 MEG of RAM, set both to 2048. For example, here's what it would look like for 16 MEG:

[vcache]
MinFileCache=2048
MaxFileCache=2048

Resave the file.

This limits the RAM that Win95 uses for its disk cache. Normally, Win95 allocates all free RAM for that, which could interfere with memory allocations that your sequencer is trying to make during playback, and cause slowdowns in performance.

To set Win95's virtual memory (ie, swap file) to a fixed size, open Control Panel's System notebook and flip to the Performance page. Click on the "Virtual memory" button. Select "Let me specify my own virtual memory settings". For the Minimum setting, enter 0. For the Maximum setting, enter 20 if you have 8 MEG of RAM, or 28 if you have 16 MEG.

This prevents the swap file from growing and shrinking on your hard drive. It's also recommended that you defrag your drive afterwards. If you have more than one drive, try putting the swap file on a drive other than the one where you'll be recording digital audio.

After the above adjustments, reboot your system.

I can't get any sound from my MS-DOS game programs when I play them under Win95 (ie, in a DOS prompt window or screen). I have to reboot the computer in "MS-DOS Mode" (ie, without Win95 loading at all) in order to get any sound. Why?

The problem is that this DOS prompt window or screen is really running under Win95 (ie, you're running an MS-DOS shell that uses Win95's "Current Configuration", rather than booting into a "New Configuration" without Win95 loaded and in control), and you have incomplete Win95 drivers for your sound card.

Here's the deal. When you boot into this DOS shell, Win95 is still loaded and in control (ie, you're running a Win95 DOS VDM). All of your Win95 drivers are managing access to your hardware,

including your sound card. In MS-speak, you're using Win95's "Current Configuration". Your system isn't really rebooting into DOS so much as it's running DOS on top of Win95. (And that's why you can enter and exit from that DOS prompt, back to Win95, so quickly). Whether you select an MS-DOS Prompt from the Start Menu's Programs group, or Shutdown to MS-DOS Mode, Win95 is still loaded and in control (if you're using the "Current Configuration"). In fact, the Win95 drivers for your sound card are always in control of the card, unbeknowst to your MS-DOS program even. It's sort of like how a DOS "Terminate and Stay Resident" (TSR) utility can surreptiously wedge itself inbetween a program and some hardware.

There are 2 things that your Win95 drivers do. First, they manage interaction between Windows programs and your hardware. Secondly, they manage interaction between MS-DOS programs and your hardware... when Win95 is still in control. Your MS-DOS programs don't even know that Win95 (and its drivers) are still in control. Everytime that your MS-DOS program tries to execute an instruction that reads or writes to hardware, the Intel CPU (sneaky guy that it is) immediately "traps" that access and gives control over to Win95 and its drivers, who then manage the actual access. (In the process, Win95 also gets a chance to try to implement some crash protection, and perform multi-tasking).

Whereas your sound card's Win95 drivers may well support Windows apps access to hardware, it appears that those drivers don't properly support access by an MS-DOS program running under Win95. So, you can't run your MS-DOS stuff under Win95 (unless you get new Win95 drivers that support MS-DOS programs). What you're going to have to do is completely flush Win95 and its drivers out of memory, and reload MS-DOS so that it really is in control. That's exactly what happens when you boot into "MS-DOS Mode" without loading Win95. You can also choose to have your MS-DOS programs run under a "New Configuration" which causes Win95 to kill itself, and MS-DOS 7.0 (included with Win95) then loads.

You can setup that MS-DOS Prompt (or the Shutdown "Restart in MS-DOS Mode") to boot into a new configuration of MS-DOS only, instead of using Win95's current configuration (and leaving Win95 still in control). The only problem is that, now that your DOS programs don't have Win95's drivers managing the hardware, for example Win95's drivers for your CDROM and mouse, you're going to need to copy some DOS drivers for your mouse and CD-ROM to your HD, and create CONFIG.SYS and AUTOEXEC.BAT files that load these drivers.

Here's how to setup your computer so that it boots into MS-DOS only:

1. Make sure that you copy MS-DOS drivers to your HD for any devices that need such, like your CD-ROM and mouse.
2. Open "Start -> Programs -> Windows Explorer".
3. Go to the "Windows" directory, and find the "Exit to Dos" icon. This holds the settings used when you boot into MS-DOS mode via "Start -> Programs -> MS-DOS Prompt" or Shutdown with "Restart in MS-DOS Mode".
4. Click once on it to open its pop-up menu, and select "Properties".
5. Go to the "Program" page.
6. Click on the "Advanced" button.

7. Select "Specify a new MS-DOS configuration" (instead of "Use current MS-DOS configuration"). Now the two boxes below will let you enter text. In the top box, you need to type out a CONFIG.SYS file exactly like it would have appeared if you only had MS-DOS on your system, and you wanted to load your DOS drivers. Of course, you need to specify everything. You need to load your mouse driver for MS-DOS. You need to load your CD-ROM driver for MS-DOS (probably using MSCDEX if you've got an IDE CD-ROM). You may need to copy these MS-DOS drivers from the disks shipped with your mouse and CD-ROM, to someplace on your HD. In short, you need to recreate CONFIG.SYS like it would be for MS-DOS, and copy any MS-DOS drivers for your mouse, CDROM, and sound card to your HD. For example, here's an example of the lines that I typed into the CONFIG.SYS box. I'm loading my mouse driver MOUSE.EXE, and my CD-ROM driver d011v110.sys. I'm not using any driver for my Vibra16 sound card (but the SB applets need an environment variable set).

```
REM Stuff that you probably need
DOS=HIGH,UMB
Device=C:\WINDOWS\Himem.sys
FILES=20
REM Set this to the last drive on your system
LastDriveHigh=E
REM Here's my CD-ROM driver. Your driver may have a different name.
REM It may require you to use "Device" instead of "DeviceHigh".
REM Consult your CDROM manual.
DeviceHigh=C:\d011v110.sys /d:mscd000 /n:1
```

Now, in the lower box, you need to type the lines for an AUTOEXEC.BAT file. Here's what I use:

```
REM A place where programs can toss garbage files. It must exist
SET TMP=C:\WINDOWS\TEMP
SET TEMP=C:\WINDOWS\TEMP
REM Just for the shell's prompt
SET PROMPT=&p$g
REM Where Win95 system files reside
SET winbootdir=C:\WINDOWS
REM Where MS-DOS can find system files
SET PATH=C:\WINDOWS;C:\WINDOWS\COMMAND
REM This is for the Sound Blaster Vibra16 PnP applets
SET BLASTER=A220 I5 D1 H5 P330 T6
REM Here's my mouse driver. Yours may have a different name.
C:\MOUSE.EXE
REM My CDROM is an IDE unit. Its driver uses MSCDEX to manage
REM the driver under MS-DOS.
MSCDEX.EXE /d:mscd000 /m:18
```

8. Now select "OK" on the dialogs to close them down and you're in business.

So what's happening here? What's the difference between "Specify a new MS-DOS configuration" and "Use current MS-DOS configuration"? As I explained, with the latter, Win95 doesn't reboot the computer. Win95 is still loaded and in control, using its Win95 drivers to manage your MS-DOS programs use of hardware. But, when you use a "new MS-DOS configuration", Win95 is completely flushed out of memory, and runs those CONFIG.SYS and AUTOEXEC.BAT scripts above. As you'll notice above, I've using my old MS-DOS drivers. So this is how you can boot into an environment that is the same as MS-DOS running your old drivers or TSRs, and yet, when you go back to Win95, you're back to using your new Win95 drivers (and the old real-mode drivers or TSRs are flushed from memory again -- which is good because, you don't want that stuff loaded with Win95 if you can avoid it). This is how you can have your MS-DOS games run without the interference of incomplete Win95 drivers that don't support sound for those MS-DOS programs, and yet still use those Win95 drivers with Windows programs which are supported.

The disadvantage is that, in order to completely flush Win95 and leave MS-DOS totally in control, Win95 has to reboot the computer. So, using "Specify a new MS-DOS configuration" will take much longer to switch to than "Use current MS-DOS configuration". But, it will allow you to use your old DOS TSRs with MS-DOS software, and your new Win95 drivers under Win95 (without resorting to always loading your old drivers/TSRs under Win95 and risking any weirdness between the two simultaneously loaded "old versus new" drivers). Plus, if you have Win95 drivers that don't properly support MS-DOS programs running under Win95, then you need to boot into MS-DOS mode anyway.

Incidentally, if you prefer to have only certain programs reboot into this "MS-DOS only" mode, then instead of performing the above procedure on the "Exit to Dos" shortcut, do this on the shortcut for the particular MS-DOS program(s) you desire. (ie, Click on the program's shortcut to bring up "Properties", and take it from there).

When I opened up Control's Panel's System notebook, and flipped to the Device Manager page (ie, to see a list of the hardware in my system), I noticed a small, yellow exclamation mark next to my sound card's name. What is this?

That exclamation mark you see next to a device name in Windows' System Notebook "Device Manager" page indicates a conflict (usually IRQ) or that something is wrong with the device's response. (ie, For example, I use SCSI devices rather than IDE, so I disabled my motherboard's IDE ports. My Plug and Play BIOS told Windows that I had IDE ports, but Windows didn't get any response from them, hence that exclamation mark next to my "PCI IDE controller".

In fact, it's common to see exclamation marks related to Plug And Play ISA cards. Often the devices appear to have several "phantom" units, each one dedicated to whatever IRQs the device supports,

and Windows thinks that these are real yet gets no response from them. Hence you may see several instances of a particular Plug and Play ISA card listed, with all but one having an exclamation mark. It's OK to leave these phantom devices, and if Windows ever tells you it detected new hardware the next time you boot, tell Windows to ignore each one of those phantom devices as it pops up a dialog for each, asking you to choose a driver to install). Furthermore, for such phantom devices, you should highlight the name of each one in Device Manager, click on the Properties button, and remove the checkmark for "Original Configuration (current)". Leave only the real device (ie, the one whose IRQ and base is set to what your card really is using) enabled for "Original Configuration".

Furthermore, Device Manager can be used to identify possible hardware conflicts in your system. If you click upon the "Print" button, a dialog will pop up which allows you to print out a listing of what devices are using various IRQ, base I/O addresses, and DMA channels. (The dialog also offers an option, "print to file" to save this text listing in a file on your harddrive. The file will end in a .PRN extension. This is handy if you need to email your settings to tech support). If you look through this list and find a resource being used by more than one hardware device, that would indicate a likely hardware conflict. You should change the DIP switches or jumpers upon one of the devices (or if both devices are PnP, then try to use a software configuration utility to alter one of the device's resources. Better yet, make sure that you're using a PnP BIOS, and it's the latest version).

When I use Windows NT 4.0 (or Windows 2000 or Windows XP) Control Panel's "Add New Hardware", I can manually locate my sound card's INF file, and NT/2000/XP seems to install the card's Windows 95 driver. But my card does not appear to be operating at all under Windows NT/2000/XP. Why?

Windows NT/2000/XP uses an entirely different driver model than Windows 95, 98, ME, or Windows 3.1. Windows NT/2000/XP has a 32-bit driver model whereas Windows 3.1 and Windows 95/98/ME still have some 16-bit components in their sound card drivers. You cannot use Windows 3.1 or Windows 95/98/ME drivers under Windows NT/2000/XP. You need a Windows NT/2000/XP driver for your sound card. Although Windows NT/2000/XP recognizes the INF file for a Win3.1 or Win95/98/ME driver (since WinNT/2000/XP drivers also use the INF file for installation purposes), it may appear that WinNT/2000/XP has installed the 16-bit driver, but it can't use such a driver. Typically, you'll get some sort of error message that the driver is not readable when you reboot your system

Recently, Microsoft has created a new driver model (Windows Driver Model, or WDM) which works with both the Windows ME and 2000 and XP operating systems. So a WDM driver written for Windows ME will should work under 2000/XP (and vice versa). Of course, you would need a new driver for your sound card if you wanted one of these WDM drivers. Otherwise, Win2000/XP continues to support current Windows NT drivers (so if you can't specifically find a Windows XP

driver, then at least try to find a Windows 2000 or Windows NT driver), and Win98 continues to support Win3.1 and Win95 drivers, but neither will be able to use the others' current drivers.

Here is a FAQ for Roland audio cards. After each question, the audio cards for which the answer is valid are listed.

Note that I have written two RAP-10 utilities and one SC-7 utility for Windows 95 (located on this web site under Software Programs) which allow you to easily alter some parameters that you would otherwise have to adjust via System Exclusive messages as described in this FAQ.

The questions covered are:

Can I use software other than AudioTools?
Do I have to buy an MCB-10 for the RAP-10?
Can reverb/chorus/delay be added to digital audio?
How do I control digital audio parameters using my sequencer?
How can I adjust each individual drum's tuning, pan, etc?
How can I adjust master tuning?
How do I prevent the GM module from playing MIDI channels?
How do I enable aftertouch, and adjust what it affects?
Do I use the Windows MPU-401 device driver with my SCD-10 or SCD-15?
What are the pin connectors on the RAP-10 CDROM connector?
Why is the Volume fader greyed out on Win95's MultiMedia notebook for the RAP-10?
Is there a Windows NT or Windows 2000 driver for the RAP-10?
I tried your RAP-10 Win95 utilities, and I'm having troubles. Help?
How do I reset an SCC-1 (Sound Canvas) to GM mode?

Can I use the RAP-10 with software other than AudioTools?

Covers RAP-10.

Yes. Since the RAP-10 ships with a Windows MCI driver, you can use it with any Windows program that works with a *Windows compatible soundcard*, including CakeWalk. You can also utilize the RAP-10 driver with the Windows MIDI Mapper.

Furthermore, since the RAP-10 supports the MPU-401 UART mode in hardware, you can use it with any MS-DOS software that requires an MPU-401 (UART mode only, not Intelligent mode).

Frankly, Audio Tools is a very questionably written program. It appears to trap some access to the RAP-10 and therefore actually interferes with the operation of other MIDI software used with the RAP-10. Several people have reported that mysterious problems they've had with various MIDI software has disappeared merely by removing Audio Tools from their systems. My recommendation is that you do not install Audio Tools at all unless you absolutely need to use this software. If you're using Windows 95, you should investigate my freely available RAP-10 utilities (located on this web

site under [Software Programs](#)).

Alternately, you can set some RAP-10 parameters during bootup by creating a text file named **RAP10.INI** in your Windows directory. When Windows first boots up, the RAP-10 driver will read this text file. It contains several lines which set the Line/Mic Input, Monitor On/Off, Aux Volume, Mic/Line Volume, and WAVE playback of the RAP-10. Here are the lines of text that should appear in RAP10.INI

```
Mic/Line Gain=Mic          ;alternative = Line
Monitor=On                 ;alternative = Off
Aux Volume=127             ;range = 0 to 255
Mic/Line Volume=127        ;range = 0 to 255
Wave Volume=60             ;range = 0 to ???
```

In other words, if you change the first line to:

Mic/Line Gain=Line

then the RAP-10 will power up with LINE gain instead of MIC gain. One caveat -- in order to return to MIC gain, you'll have to edit that text file, changing it back to "Mic" and then reboot your computer. Setting Wave Volume is handy for reducing the RAP-10 volume at bootup so that Windows startup sounds don't rattle your walls.

Do I have to buy the expensive MCB-10 box to connect to the RAP-10 for MIDI IN and OUT jacks?

Covers RAP-10.

No, the RAP-10's joystick port is 100% compatible with a Sound Blaster, so any "MIDI Adapter" that works for the SB will work for the RAP-10. For example, I bought a Media Vision MIDIMATE for about $15. The MCB-10 is made out of cast iron, with a heavy duty cable and connectors, and is accordingly overkill and overpriced.

Can reverb, chorus, and delay really be added to the digital audio tracks?

Covers RAP-10, SCP-55B.

Yes. Each digital audio track has its own level for adding reverb/delay and chorus, as does each "part" in the wavetable portion of the card. Each audio track also has its own volume, tuning (in cents, as well as transposing up or down in half steps), and pan (ie, the digital audio goes out the same stereo out jack as the wavetable voices).

What kind of control do I have over digital audio playback using a sequencer program like CakeWalk? I can use MCI commands, but then I have no control over the volume, panning, and effects (as applied to digital audio). Can I control these from CakeWalk just like I do using Audio Tools sliders (ie, Roland's bundled software)?

Covers RAP-10, SCP-55B.

Yes, but instead of having convenient sliders a la Audio Tools, you'll have to manually enter System Exclusive MIDI events in your sequencer track. Nevertheless, unlike with other audio cards, you can fully control the digital audio playback with just about any sequencer program.

In fact, you don't need MCI commands for that control. Roland had the foresight to add a SYSTEM EXCLUSIVE implementation to the RAP-10 and SCP-55 that allows you to adjust each digital audio track's volume, panning, brightness, chorus level, reverb level, pitch (both in cents as well as transposing by half-steps), and sample rate (11.025, 22.050, or 44.100 KHz) BY SENDING IT SYSEX MESSAGES (which most all sequencers support). So for example, if you want to change the volume of a digital audio track (even while it's playing), you send a SYSEX message to the RAP-10. You just need to know those SYSEX messages, and you can insert them wherever you want in your CakeWalk track, and viola, there's all of the control that you wanted. (In fact, when you move the Audio Tools volume slider, all it's doing is sending those SYSEX messages to the RAP-10. There's no reason why you can't have CakeWalk or any other program that supports SYSEX do the same thing, but of course, you'll have to manually enter the desired SYSEX messages as events in a CakeWalk track. Come to think of it, you should be able to write some CAL routines that will really simplify the process, for example, asking you to type in a volume, and then the CAL routine will create the actual SYSEX event for it).

Now, Roland didn't list the digital audio track's SYSEX messages in the owners manual. No problem. I'm a registered Roland developer, and have the RAP-10 technical docs right here. So, I'm going to tell you how to do this (in excruciating detail).

I'll skip the part about how you enter a SYSTEM EXCLUSIVE event in CakeWalk. Consult your manual for that. Of course, you send the SYSEX messages to the RAP-10's MIDI (out) port, and the RAP-10 automatically recognizes and uses them. So, you may want to reserve a CakeWalk track for just these RAP-10 SYSEX events and set it to output to the RAP-10.

Each digital audio parameter (ie, volume, panning, tuning, etc) has its own "address" where you adjust the setting. For example, there's a certain place (ie, address) within the RAP-10 that you reference to adjust digital audio VOLUME. There's a different place (ie, address) that you reference to adjust the digital audio PAN. The address that you specify tells the RAP-10 which digital audio parameter you wish to adjust. You specify the address within the SYSEX message itself. You also put the new value for that parameter within the same SYSEX message.

Each digital audio track has its own addresses for volume, panning, etc. For example, the left track's volume is at a different address than the right track's volume. So, you have independent control over the two tracks' parameters. If you use the 2 tracks as 1 stereo track (ie, stereo WAVE), then there is a third set of addresses for that. (ie, Think of the RAP-10's stereo digital audio as a third, stereo track).

The format (ie, template) for a SYSEX message is (in decimal bytes):

240 65 16 86 18 [address] [value] [checksum] 247

The address, value, and checksum are the parts of it that you have to fill in yourself (depending upon which parameter you're adjusting, and what value you're setting it to).

The address for the LEFT digital audio track's VOLUME is the three bytes: 2 0 2. The value can be anything from 0 (inaudible) to 127 (loudest).

The checksum is one byte. It is calculated by adding the 3 address bytes and the value byte. If this is over 128, then subtract by 128 until you have a number less than 128. Take this number and subtract it from 128. That's the checksum.

OK, let's construct a SYSEX message to set the LEFT digital audio track's volume to 64 (ie, half volume). Take the above template, and put the "address" 2 0 2 in it, and the "value" 64 in it.

240 65 16 86 18 2 0 2 64 [checksum] 247

Now add up the address and value bytes. That's 2+0+2+64 = 68. This is less than 128. Subtract it from 128, and we have the checksum of 60. Here's the complete SYSEX message:

240 65 16 86 18 2 0 2 64 60 247

The address for the RIGHT digital audio track's VOLUME is the three bytes: 2 0 12.

OK, let's construct a SYSEX message to set the RIGHT digital audio track's volume to 127 (ie, full volume). Take the above template, and put the "address" 2 0 12 in it, and the "value" 127 in it.

240 65 16 86 18 2 0 12 127 [checksum] 247

Now add up the address and value bytes. That's 2+0+12+127 = 141. This is more than 128, so we subtract 128 until we get a value less than 128. The value we get is 13. Subtract 13 from 128, and we have the checksum of 115. Here's the complete SYSEX message:

240 65 16 86 18 2 0 12 127 115 247

Just place these SYSEX events wherever you want to change the volume for a digital audio track (ie,

before the MCI "play" event, or after it if you want to change volume during playback). You can use as many of these SYSEX events as desirable (ie, for example if you want to slowly fade the audio out, then put lots of SYSEX events in a row, with decreasing values for the volume).

If you use the digital audio tracks in stereo, then the volume is at address: 3 0 2.

Here are the other parameters for the LEFT track. The Pan is at address 2 0 8. The value can be 1 (hard left) to 127 (hard right). 64 is center position. The Reverb/Delay Level is at address 2 0 9. The value is 0 (no reverb) to 127 (full reverb). The Chorus Level is at address 2 0 10. The value is 0 to 127. The Transpose is at address 2 0 5. A value of 64 means no transpose. Higher values transpose the track UP in half steps. For example, 65 plays the digital audio track a half step up. Lower values transpose the track DOWN in half steps. 63 transposes the track down a half step. The Sample Rate is at address 2 0 4. The value is 0 for 11KHz, 1 for 22KHz, or 2 for 44KHz. (I'm not going to bother detailing the last parameter which allows you to adjust the track's tuning in cents. This requires a slightly different SYSEX message format, and I don't want to get into that unless you NEED to know).

Time for another example. Let's set the LEFT track's PAN to hard left. Take the template and plug in the address 2 0 8, and the value 1. Add up those 4 bytes. 11 is less than 128. Subtract 11 from 128 to get a checksum of 117. Here's the SYSEX message:

240 65 16 86 18 2 0 8 1 117 247

Here are the other parameters for the RIGHT track. The Pan is at address 2 0 18. The Reverb Level is at address 2 0 19. The Chorus Level is at address 2 0 20. The Transpose is at address 2 0 15. The Sample Rate is at address 2 0 14.

If you use the 2 tracks in stereo, here are the other addresses for that. The Pan is at address 3 0 8. The Reverb Level is at address 3 0 9. The Chorus Level is at address 3 0 10. The Transpose is at address 3 0 5. The Sample Rate is at address 3 0 4.

Unlike owners of other sound cards, the RAP-10 owner has all of this control over digital audio tracks using any sequencer that supports SYSTEM EXCLUSIVE events. Write yourself a few CAL routines (ie, for example, one that inserts a bunch of SYSEX events to pan a track across the stereo spectrum, etc) and you'll have an arsenal of special effects for your digital audio that don't require MCI (and don't exhibit some of the difficulties that you have in syncing MCI control with MIDI playback).

How I can tune each individual drum sound within a kit plus control the volume, panning, reverb. chorus etc? I'm not talking about the overall volume, panning, etc, of the entire kit, but setting individual volume, panning, etc for EACH DRUM SOUND.

Covers RAP-10, SCP-55B, SCC-1B, SC-7, SCD-10, SCD-15, and others.

You need to use the *Non-Registered Parameter Number* controllers. For a generic discussion of what NRPN and RPN controllers are, and how they may be used, read the article [Using Registered and Non-Registered Parameter Number controllers](#) now.

The good news is that your Roland manual lists all of the various RPN and NRPN parameter numbers that the RAP-10 responds to. It shows an example template for each parameter number where the parameter number is already broken up into its 2 pieces. Look up the section where NRPN controller is discussed to see what parameter numbers Roland has defined for what purposes. (On Roland sound cards, often NRPN numbers are used only to adjust settings for individual drums of a kit. For adjusting things such as Reverb and Chorus levels, overall Pan and Volume, etc, there are dedicated controllers for that. For adjusting things such as Voice Reserve, Roland typically uses System Exclusive, rather than Controller messages).

For example, let's discuss how to change the tuning on the Open HiHat drum sound, transposing it down a half step.

If you look under the *MIDI Implementation* section of your manual (starting on pg 19), you search for the section on NRPN controller (starts on pg 19 and goes to pg 20). At the top of pg 20, you see that there are 5 parameters that can be controlled for each drum sound (ie, key):

Pitch (in half steps)
Volume (ie, TVA level)
Pan
Reverb level
Chorus level

Now, Roland already breaks up the "high bits" and "low bits" of the NRPN parameter number for you. In fact, the "high" and "low" bits aren't really related in the way that Roland uses them (ie, the "high" and "low" bits aren't ever meant to be pieced together to create a parameter number here). Rather, Roland uses the high bits to indicate which one of the above 5 parameters you wish to adjust. The low bits indicate something else totally unrelated to that as you'll see.

To adjust Pitch, the "high bits" of the NRPN number is 18 hex (ie, 24 decimal). To adjust Volume, the "high bits" of the NRPN number would instead be 1A hex (ie, 26 decimal). For Pan, "high bits" is 1C hex (ie, 28 decimal). For Reverb Level, "high bits" is 1D hex (ie, 29 decimal). For Chorus Level, "high bits" is 1E hex (ie, 30 decimal).

For "low bits", you need to know the MIDI note number for the Drum Key that you want to adjust. For example, the Open HiHat is assigned to the A#1 key which is MIDI note number 46. (There is a table called *Drum Set Table* in your manual which lists the MIDI note numbers for all drum sounds). The desired MIDI note number is the "low bits".

So, to tell the RAP-10 that we want to adjust Pitch for a drum sound, we send the NRPN with the "high bits" of 24. (I'm assuming MIDI channel 10).

185 98 24

Then, to tell the RAP-10 that we want to adjust drum key 46 (ie, the Open HiHat), we send the NRPN with the "low bits" of 46.

185 99 46

Finally, to actually transpose the Pitch down a half step, we use the Data Entry Slider controller to set the tuning. A value of 64 is centered (ie, root pitch) tuning. So, if we want a half step down, we subtract one and send a value of 63. (Pan is calculated the same way -- 64 is centered panning, lower values move left, and higher values move right).

185 6 63

What if you want to set the Volume of the Cowbell to 80? First, to indicate that we want to adjust Volume, we send the NRPN "high bits" of 26.

185 98 26

Second, we need to know the MIDI note number for the Cowbell, which is 56. We send that as the NRPN "low bits".

185 99 56

Finally, to actually set the Volume to 80, we use the Data Entry Slider controller.

185 6 80

Is that all there is to it? Well no. When the RAP-10 first powers up, it defaults to ignoring these NRPN messages. So, you have to enable NRPN recognition by sending the RAP-10 the following SYSTEM EXCLUSIVE message (values are in hexidecimal):

F0 41 10 56 12 01 00 01 01 7D F7

This only needs to be done once when the RAP-10 powers up (or always after the RAP-10 is sent a *Turn General MIDI System On* system exclusive message).

I can't use the RAP-10 and an external module at the same time since both will sound. How can I get the RAP-10 to stop responding to MIDI channels that I want only my external

module to play?

Covers RAP-10, SCP-55B, SCC-1B, SCD-10, SCD-15, SC-7, and others.

The RAP-10's built-in sound module is multi-timbral. It has 16 "Parts" (ie, 16 distinct entities inside of it, each of which can play one instrument at a time upon any one of the 16 MIDI channels). When the RAP-10 is powered up, the 16 parts are set to the default 16 MIDI channels (ie, Part 1 is the Drum Part, and is set to play all MIDI messages on channel 10, Part 2 plays messages on channel 1, Part 3 plays channel 2, etc). You're undoubtably unaware that the 16 "Parts" of the RAP-10 can be turned off individually via SYSTEM EXCLUSIVE messages sent to the RAP-10. Alternately, a Part's MIDI channel can be changed via a SYSEX message sent to the RAP-10. For example, you can set both Part 2 and Part 3 to play MIDI messages upon channel 2. You can also disable a Part from sounding at all by setting its MIDI channel to "off". You need to send a SYSEX message to the RAP-10. If you have a sequencer program that supports SYSEX, you can manually enter a message in one of your "tracks" (and set that track to output to the RAP-10). (It's even better if you have a RAP-10 Editor program that has "controls" to change a Part's MIDI channel, and the Editor takes care of sending the appropriate SYSEX messages. Anyway, I wrote a RAP-10 Editor for OS/2, and a Win95 RAP-10 Playback Panel, as I'm a registered Roland developer).

Here's the template of the SYSEX message (with values expressed in decimal) to change a Part's MIDI channel:

240 65 16 86 18 1 p# 0 ch sum 247

You need to replace p# with the Part number whose MIDI channel you wish to change (where Part 1 is 0, Part 2 is 1,... Part 16 is 15. Note that the Drum Part is Part 1. The remaining 15 Parts, 2 to 16, are the 15 instrumental parts). You need to replace ch with the desired MIDI channel (where Channel 1 is 0, Channel 2 is 1,...Channel 16 is 15, and to turn a Part off, specify ch = 16). Finally, you need to replace sum with the calculated checksum. The easiest way to come up with the checksum for this particular message is to add p# and ch, add 1, and subtract this from 128.

For example, to set Part 1 (ie, Drums) to MIDI channel 9, replace p# with 0 and replace ch with 8. Checksum is 128 - (8+0+1) or 119.

240 65 16 86 18 1 0 0 8 119 247

Normally, you'll want the 16 Parts each set to their own channel (ie, Drum Part set to Channel 10, Part 2 set to Channel 1, Part 3 set to Channel 2, etc.) which is the RAP-10's default setup. But then you'll probably want to turn off Parts that are set to MIDI channels reserved for your external gear. Use the above SYSEX message for that. For example, let's say that you want to turn off the Part set to MIDI channel 1. By default, Part 2 is set to MIDI channel 1. Replace p# with 1 and set ch to 16 (ie, off). Checksum is 128 - (2+16+1) or 109.

240 65 16 86 18 1 1 0 16 109 247

Of course, at the end of your track, you may want to turn that Part back on (ie, set it back to channel 1).

240 65 16 86 18 1 1 0 0 127 247

The original Sound Canvas (SC-55), as well as some of the other SC models, use a slightly different System Exclusive message to change a Part. For example, in the SC-55, the SysEx message takes the form:

240 65 16 66 18 64 p#+16 2 ch sum 247

Note that 86 is replaced by 66. This indicates a different model number than the RAP-10. Also, the 1 is replaced with 64. When you specify the part number, you need to add 16 to it. For example, if you want part 1, then you specify 17.

How do I turn on Aftertouch on my card? I'm looking in the manual which says that the card HAS aftertouch, but it defaults to off.

Covers RAP-10, SCP-55B, SCC-1B, SCD-10, SCD-15, SC-7, and others.

First, let's talk about MIDI Aftertouch. There are two kinds of "aftertouch" or "pressure"; "Key Pressure", sometimes called "polyphonic aftertouch", and "Channel Pressure", sometimes called "monophonic aftertouch".

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.

What's the difference between Key Pressure (which I'll call "AfterTouch" here for brevity) 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 controller 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.

OK, back to the RAP-10. The RAP-10 only recognizes Channel Pressure, not Key Pressure. (RAP-10 Parts ignore all Key Pressure messages). All of the sounding notes of a Part are affected equally by a Channel Pressure message. (NOTE: The RAP-10 has 15 Parts plus a Rhythm Part, and these are by default assigned to the 16 available MIDI Channels. For example, Part 1 is the Drum Part, and is assigned to handle everything on MIDI channel 10 by default. Part 2 handles everything upon MIDI channel 1 by default. Part 3 handles MIDI channel 2. Etc. You CAN reassign parts to other channels. For example, you could have two RAP-10 Parts both playing everything upon channel 16. You can also turn Parts off, for example if you didn't want the RAP-10 to respond to particular MIDI channels at all).

Channel Pressure can be used to control any or even ALL of the following 4 aspects of a Part:

1. Filter Cutoff Frequency (ie, Brightness) -- As pressure increases and decreases, the tone gets brighter or duller.
2. LFO Rate (ie, Vibrato Speed) -- As pressure increases and decreases, the vibrato speeds up or slows down.
3. LFO Depth (ie, Vibrato Amount) -- As pressure increases and decreases, the vibrato effect gets more or less pronounced.
4. Volume -- As pressure increases and decreases, the volume (of all sounding notes for the Part) increases or decreases.

Each of the 15 Parts, and the Rhythm Part has its own Channel Pressure settings. For example, you can set Part 1 to get brighter as it receives increasing Pressure. You can set Part 2 to get duller as it receives increasing Pressure. You can set Part 3's Vibrato Amount to increase as it receives increasing Pressure, AND set Part 3's Volume to decrease as it receives increasing Pressure. Etc.

The RAP-10 is designed such that you can alter virtually all of its settings (including digital audio playback settings) by sending System Exclusive MIDI messages to it. If you have some RAP-10 Patch Editor software such as my RAP-10 Editor for OS/2, then the software will likely have some graphical controls that allow you to set the desired effects of Channel Pressure for each Part. The software will take care of sending the proper System Exclusive messages to the RAP-10. If don't have such a tool (and check to see if Audio Tools allows setting Channel Pressure -- it should, but I

wouldn't know since I'm a developer, got my RAP-10 sans that Windows software, and wrote my own OS/2 stuff), don't fret. As long as you have a sequencer that lets you manually enter your own System Exclusive events, you can create and enter the desired messages for your sequencer to send to the RAP-10. I'm going to tell you what you need to know to set Channel Pressure for any Part.

I'll skip the part about how you enter a SYSTEM EXCLUSIVE event in your sequencer. Consult your seq manual for that. Of course, you send the SYSEX messages to the RAP-10's MIDI (out) port, and the RAP-10 automatically recognizes and uses them. So, you may want to reserve a seq track for just these RAP-10 SYSEX events and set the track to output to the RAP-10.

Each Part's Channel Pressure settings have their own "addresses" where you adjust the settings. For example, there's a certain place (ie, address) within the RAP-10 that you reference to adjust how Channel Pressure affects Part 1's Filter Cutoff Frequency. There's a different place (ie, address) that you reference to adjust how Channel Pressure affects Part 1's LFO Rate. The address that you specify tells the RAP-10 which Channel Pressure parameter you wish to adjust for which Part. You specify the address within the SYSEX message itself. You also put the new value for that parameter within the same SYSEX message.

The format (ie, template) for a SYSEX message is (in decimal bytes):

240 65 16 86 18 [address] [value] [checksum] 247

The address, value, and checksum are the parts of it that you have to fill in yourself (depending upon which parameter you're adjusting, and what value you're setting it to).

The address to set Part 1's Filter Cutoff is the three bytes: 1 1 4. The value can be anything from 0 (increasing pressure makes the part very dull) to 127 (increasing pressure makes the part very bright). A value of 64 is the middle value, meaning that pressure does not affect Part 1's Filter Cutoff at all.

The checksum is one byte. It is calculated by adding the 3 address bytes and the value byte. If this is over 128, then subtract by 128 until you have a number less than 128. Take this number and subtract it from 128. That's the checksum.

OK, let's construct a SYSEX message to set Part 1's sound to made brighter with increasing pressure. We'll pick some number above 64, for example 90. Take the above template, and put the "address" 1 1 4 in it, and the "value" 90 in it.

240 65 16 86 18 1 1 4 90 [checksum] 247

Now add up the address and value bytes. That's 1+1+4+90 = 96. This is less than 128. Subtract it from 128, and we have the checksum of 32. Here's the complete SYSEX message:

240 65 16 86 18 1 1 4 90 32 247

Let your sequencer send the above message at the start of your song, and viola, Part 1's Filter Cutoff

is now affected by Channel Pressure.

Here's a chart showing the addresses of all Channel Pressure parameters for all 15 Parts and the Rhythm Part:

```
Part 2                          Part 3                          Part 4
   Brightness  1 1 4              Brightness  1 2 4              Brightness  1 3 4
   Volume      1 1 5              Volume      1 2 5              Volume      1 3 5
   LFO Rate    1 1 6              LFO Rate    1 2 6              LFO Rate    1 3 6
   LFO Speed   1 1 7              LFO Speed   1 2 7              LFO Speed   1 3 7

Part 5                          Part 6                          Part 7
   Brightness  1 4 4              Brightness  1 5 4              Brightness  1 6 4
   Volume      1 4 5              Volume      1 5 5              Volume      1 6 5
   LFO Rate    1 4 6              LFO Rate    1 5 6              LFO Rate    1 6 6
   LFO Speed   1 4 7              LFO Speed   1 5 7              LFO Speed   1 6 7

Part 8                          Part 9                          Part 10
   Brightness  1 7 4              Brightness  1 8 4              Brightness  1 9 4
   Volume      1 7 5              Volume      1 8 5              Volume      1 9 5
   LFO Rate    1 7 6              LFO Rate    1 8 6              LFO Rate    1 9 6
   LFO Speed   1 7 7              LFO Speed   1 8 7              LFO Speed   1 9 7

Part 11                         Part 12                         Part 13
   Brightness  1 10 4             Brightness  1 11 4             Brightness  1 12 4
   Volume      1 10 5             Volume      1 11 5             Volume      1 12 5
   LFO Rate    1 10 6             LFO Rate    1 11 6             LFO Rate    1 12 6
   LFO Speed   1 10 7             LFO Speed   1 11 7             LFO Speed   1 12 7

Part 14                         Part 15                         Part 16
   Brightness  1 13 4             Brightness  1 14 4             Brightness  1 15 4
   Volume      1 13 5             Volume      1 14 5             Volume      1 15 5
   LFO Rate    1 13 6             LFO Rate    1 14 6             LFO Rate    1 15 6
   LFO Speed   1 13 7             LFO Speed   1 14 7             LFO Speed   1 15 7

Rhythm (Part 1)
   Brightness  1 0 4
   Volume      1 0 5
   LFO Rate    1 0 6
   LFO Speed   1 0 7
```

So, for example, let's set the Drum Part's Volume to be increased with increasing Channel Pressure. Pick some value over 64, for example, 127. Now look up the address: 1 0 5. Plug these into the template, calculate the checksum, and our message is:

240 65 16 86 18 1 0 5 127 123 247

Do I use the Windows MPU-401 device driver with my SCD-10 or SCD-15?

Covers SCD-10, SCD-15, or other Roland daughterboard.

No. The Roland daughterboard plugs into a Sound Blaster compatible daughterboard connector upon some host card (ie, the card that the Roland daughterboard plugs into). The computer (and software) communicates with the Roland card through the host's connector (ie, hardware). Unlike the SCC-1 or RAP-10 (or even the SCP-55B), the Roland daughterboard is not a standalone card, and doesn't directly attach to the computer's bus. The daughterboard connects to the host, and it is the host that directly connects to the computer's bus. Therefore, you only need to install the driver for the host card. For example, if you have a Sound Blaster, you install the Creative Labs drivers included with that card. It is the responsibility of the host's driver to pass any software's MIDI data to the daughterboard. Usually, the driver allows "switches" (ie, additional parameters that you can specify on the line with the driver's DEVICE statement in your **config.sys** file) that you can use to tell the driver to recognize a daughterboard, and use it for MIDI playback (ie, instead of some cheesy FM sounds built into the host). Or, you can use Win95's MultiMedia notebook (MIDI page) to select the "MIDI output" of your host card. (ie, The daughterboard is internally attached to the MIDI OUT jack of the host. Don't select any "WaveTable Synth" choice as that isn't the daughterboard. That's some built-in sound module on your host). In other words, most hosts are pretty dumb. They can't tell when a daughterboard is plugged in, and the driver will ignore it until you tell the driver to do otherwise.

If you try to use the Windows MPU-401 driver, it will look for a standalone card (by default at base address 330 hex) that is MPU-401 compatible (in hardware). Most of these hosts are not MPU-401 compatible in hardware, even though they may even have some sort of MPU-ish (ie, a very poor, incomplete MPU-401 emulation) ports at address 330. The net result is that the MPU-401 driver will either be trying to control hardware that isn't really designed to be controlled by that driver, and/or the MPU-401 driver will be mucking around with the hardware at the same time that the host's driver is mucking around with the same hardware. You should never have two drivers controlling the same hardware simultaneously. Otherwise, you can expect weird things to happen.

Of course, if you did have a real MPU-401 compatible card, in addition to your host with a Roland daughterboard, then you'd be using 2 drivers; the Windows MPU-401 driver and the host's driver. But of course, the two cards would have to be set to use different base addresses and IRQ numbers, and you'd have to tell each driver what is the base address and IRQ of the card that it needs to control. (This is usually done when you install the Windows driver. A "setup" or "properties" screen for the driver allows you to specify base address and IRQ. Of course, you may also need to set jumpers on your card).

Buying an SCD-10 or SCD-15 does **not** buy you an MPU-401 compatible card like the SCC-1, RAP-10, MPU-401, SuperMPU, MusicQuest cards, etc. Even though the SCD-10 and SCD-15 may have the same waveforms, patch sets, effects, and even system exclusive implementations as the aforementioned MPU-401 compatibles, the daughterboards do not have the same hardware interface

to the computer's bus (nor the same audio output section -- they use the host's audio output) as the standalones. If you're using Windows software, this shouldn't matter much to you. (Well, the likely higher noise output from the host's audio output section may make the SCC-1, RAP-10, or external Roland module more attractive, but the way that the card interfaces to the computer's bus is irrelevant as long as you have good drivers for the host).

How can I adjust master tuning on my RAP-10?

Covers RAP-10, SC-7, and probably the SCP-55B, SCC-1B, SCD-10, SCD-15, and others.

The master tuning can span +/- 100 cents (ie, upto 100 cents above A440 tuning or 100 cents below A440 tuning). The parameter moves in .2 cents. In other words, each cent is divided into 5 steps -- that's how fine the resolution is on tuning.

The value range for the parameter is 24 to 2024, with 1024 representing A440 tuning. So, if you specify a value of 1025, that's 2 cents higher than A440 tuning. 1026 is .4 cents higher than A440. 1029 is one cent higher than A440. Conversely, below 1024 lowers the tuning below A440. 1023 is .2 cents below A440. 1019 is one cent below A440.

The template for the tuning system exclusive message is (in hexadecimal):

```
F0 41 10 42 12 40 00 00 [tune1 tune2 tune3 tune4 checksum] F7
```

You need to figure out what tuning value you want. Convert it to hexidecimal. (You can use the Windows calculator in scientific mode to do that). Break up the hexadecimal value into 4 nibbles (ie, take the 4 digits separately. If you only see 3 displayed on the calculator, pad it out with an extra 0 on the left). And calculate the checksum. That gives you the system exclusive message.

For example, assume that you want to raise the tuning 3 cents. Remember that every increment is .2 cents (ie, 1/5 of a cent), so to raise 3 cents, you're going to have to add 3 X 5 to 1024. The value will be 1039. Convert this to hexadecimal -- that's 040F. Break it up into 4 nibbles; that's 0, 4, 0, F. Plug these 4 values into the template above:

```
F0 41 10 42 12 40 00 00 00 04 00 0F checksum F7
```

All that's left is to calculate the checksum. For this tuning message, you do that by taking the 4 nibbles, convert them to decimal, add them together, then add 64. If this sum is over 128, then subtract by 128 until you have a number less than 128. Take this number and subtract it from 128. That's the checksum.

For our example, the 4 nibbles plus 64 is 0 + 4 + 0 + 15 + 64 = 83. This is less than 128. We subtract from 128 to get a checksum of 45 (ie, 2D in hexadecimal). So the entire message is:

```
F0 41 10 42 12 40 00 00 00 04 00 0F 2D F7
```

The audio out jack on my CDROM is a 4-pin connector, but the RAP-10 CDROM audio in jack has a 5-pin connector. Can I connect the two?

Covers RAP-10.

There have been two different standards for CDROM audio connectors, a 4 and a 5 pin connector. The 4-pin connector is now more common, but the RAP-10 used a 5-pin connector. Fortunately, the signals for the pins are identical. It's just that some of the pins on the 5-pin jack aren't used. You merely need a cable that is wired to connect the 4-pin jack to the RAP-10's 5-pin jack. Here's the pin signals on the RAP-10 connector:

Pin #1 - Not used
Pin #2 - Left
Pin #3 - Ground
Pin #4 - Right
Pin #5 - Not used

Why is the Volume fader greyed out on Win95's MultiMedia notebook for the RAP-10?

Covers RAP-10.

Unfortunately, the RAP-10 Windows driver does not support the Windows Mixer API. This means that there is no way for any program (or Win95) to set various RAP-10 settings via the Mixer API. For example, it's impossible for a program to set the RAP-10's Mic/Line input to Line or Mic level, or turn Monitor On or Off, or set Master Volume (which is why Win95 disables that volume slider), or set Aux In or Mic volume. (Fortunately, master volume can be set by sending the RAP-10 a System Exclusive message. But the other parameters cannot be set by programs. The only way to set such parameters is to use Audio Tools, my RAP-10 Win95 utilities, or settings in a RAP10.INI file).

Is there a Windows NT or Windows 2000 driver for the RAP-10?

Covers RAP-10.

No, there is no RAP-10 driver for NT nor Windows 2000 (which is really Windows NT 5.0).

Windows 3.1 drivers (which is what the RAP-10 driver is, and also explains why it isn't listed in Device Manager -- Device Manager does not list legacy drivers) and Win95 sound drivers will not work under Windows NT/2000. WinNT has an entirely different driver model. (In Win98 and

Windows 2000, MS supports a new driver model that works under both. But you'll need a new driver written to this specification. Older Win3.1 and Win95 drivers still will not work under Windows NT/2000).

The best that you can do is to use the MPU-401 driver that ships with NT/2000. This will give you support for the RAP-10's MIDI In and Out, and internal GM module. But, you'll lose support for the RAP-10's 2 digital audio tracks (ie, WAVE recording and playback).

There is an MPU-401 driver shipped with Win2000, but for some reason, MS didn't include an .INF file to install it. You can Download the MSMPU401.INF file to install this driver. Just save it to disk. Then go into Control Panel's "Add new hardware". Choose to install additional hardware beyond any PnP cards it detects. When it asks if your device is in the list of devices it knows, answer "No". Choose that you wish to select your own hardware from the list. Select the "hardware type" of "Sound, video, and game controllers". Click on "Have Disk" and browse to where you've saved this . INF file. Select that file and it will come up with an MPU-401 option. Select that and Win2000 should install its included MPU-401 driver.

I tried your RAP-10 utilities, but some of the parameters don't seem to be having any effect when I use CakeWalk. What's wrong?

Covers RAP-10.

First of all, you should remove Audio Tools. Audio Tools interferes with the operation of other programs using the RAP-10.

Because the RAP-10 driver doesn't support the Windows Mixer API, in order to get around this, I've had to bypass the driver and go right to the hardware. There are a couple caveats with this.

What happens is that the RAP-10 driver reads any RAP10.INI text file at bootup, and from that point on, whenever any program opens the RAP-10 driver, the driver always sets the RAP-10 back to whatever setting it established at bootup. That's why you keep seeing CakeWalk (ie, actually the RAP-10 driver) defeating some settings made with SetMic (ie, mostly the Line/Mic switch, Monitor On/Off, and perhaps Aux In and Mic volume). SetMic is working behind the driver's back, but as soon as CakeWalk opens the driver (ie, before it begins digital audio play or record), the driver resets these RAP-10 settings. As I said, due to the fact that the RAP-10 driver doesn't fully support the Windows API, there's no way for SetMic to cooperate with the driver and resolve this issue. Therefore, use of a RAP10.INI file may be your only solution, short of obtaining a new RAP-10 driver that supports the Windows mixer API.

The SCC-1 (Sound Canvas) has this MT-32 mode (which emulates an old Roland analog-synth computer card) that sounds cheesy, and some programs or midi files knock it

into this mode. How do I reset the SCC-1 (Sound Canvas) to General MIDI mode?

Covers SCC-1.

The SCC-1 was made prior to when a standard System Exclusive message was added to the MIDI specification to reset a module to General MIDI mode. Therefore, the SCC-1 uses a proprietary Roland Sysex message for this purpose. (Actually, the SCC-1 is reset into GS mode, which is the General MIDI patch set plus variations). This message is:

F0 41 10 42 12 40 00 7F 00 41 F7

Simply enter this message into your sequencer, and play it back into the SCC-1.

Newer Roland cards such as the RAP-10 use the standard GM Reset Sysex message of:

F0 7E **xx** 09 01 F7

where **xx** is the Device ID (or 7F if you wish all modules to respond to this message).

Recommended Reading:

Roland Audio Card Specs
MIDI/Audio cards for a PC clone computer

Roland JV series sound modules (such as the JV-90, JV-1080, etc, or series based upon it such as the XP series) often seem confusing to many people. Although Roland's user manuals are serviceable in their content (ie, they do contain most of the information that you need to know), some folks seem to find the language and presentation a bit too jargon-ish and terse. This article attempts to give you a broader understanding of the design of a typical Roland sound module, so that when you dig into the user manual for specific details, the manual will make more sense to you.

Banks/Patches

A typical Roland multitimbral module such as the JV-90 has numerous banks of patches. According to the MIDI specification, a MIDI bank can have, at most, 128 patches in it. But Roland often arranges patches into smaller "groups" so that they better map out to buttons upon the front panel. For example, Roland often puts 16 buttons on the front panel (dedicated to the selection of patches), in 2 rows of 8. The top row selects among 8 "subgroups" of patches. In each subgroup, there are 8 patches, and you select among those 8 using the bottom row. In this way, you always have access to 8 patches via one button per patch (ie, each bottom row button selects a different patch). You don't need to press 2 buttons in order to switch between patches, as long as you stick with those 8 patches. Of course, if you want to switch to a different subgroup of 8 patches, then you have to use the top row of buttons as well. Now, there are a total of 8 times 8 (ie, 8 subgroups of 8 patches each), or 64 patches that you can switch to at any given moment using Roland's grouping of 16 buttons on the front panel. So that's why Roland tends to arrange patches into groups of 64. Roland typically refers to such a "partial MIDI bank" (ie, half of a full, 128 patch MIDI bank) as a "Patch Group".

To switch the buttons to select between 64 other patches (ie, a different Patch Group), then you usually need to press one of the "Patch Group" buttons on the Roland. Roland gives various "labels" to these Patch Groups. Groups which are stored in a ROM chip are typically labeled with letters following the word "Preset", for example, the first 4 groups in ROM may be labeled as "Preset A", "Preset B", "Preset C", and "Preset D". Since these patches are in ROM (ie, READ-ONLY MEMORY), you can't store any edits you make to them (unless you store the newly edited patch into some patch location in a RAM-located "User" group, thus overwriting that user patch). Hence the designation "Preset". Typically, Roland also provides some battery-backed RAM where you can store your own, customized patches. There may be one or more Patch Groups in this RAM, and they are typically labeled with letters following the word "User" or "Internal", for example the first 4 groups in RAM may be labeled as "User A", User B", User C", and "User D". You can edit these patches and store your changes back to those patch locations in RAM.

Also, Roland tends to arrange the various additional "components" of a module into their own patch groups. For example, Roland modules sometimes have patches on "data cards" that are inserted into a slot (sort of looks like PCMCIA) in the back of the unit. The module may also have a connector whereby a "voice expansion" board (to allow more polyphony) can be internally plugged into the module, and this card contains further patches. Many modules can also use a "wave expansion" board (to allow new waveforms to be used in patches) which contains yet more patches. Roland typically puts the patches upon each of these components into its own Patch Group which you select by pressing some Patch Group button on the front panel (appropriately labeled, such as W-EXP for the patches upon the wave expansion board). Although the data cards often contain battery backed

memory so that you can edit and store customized patches upon them, the wave expansion board typically has only 1 group of ROM patches. As far as the VE-JV1 voice expansion board is concerned (and all subsequent comments about voice expansion concern this board), it has RAM patches (and just 1 very small group containing only 8 patches), so that you can edit those patches' settings. But it isn't battery-backed, so every time that you turn on the unit, you have to reload the desired 8 patches into the voice expansion's RAM by sending System Exclusive MIDI messages to its MIDI input. (Roland offers an easier alternative by allowing you to internally transfer 1 Performance, and its 8 patches and drum kit, from the module's main RAM or ROM to the voice expansion's RAM. It still takes a few seconds just like a SysEx dump over MIDI, but there's no external cable connections to be made).

In conclusion, Roland typically arranges its patches into groups of 64 (or less), called "Patch Groups". There are usually 16 buttons upon the front panel which allow easily switching among 1 Patch Group's 64 patches. In order to setup the buttons to switch among a different Patch Group's 64 patches, you need to press one of the Patch Group buttons.

Rhythm Sets

A Roland "Rhythm Set" is simply a "drum kit" (ie, one mapping of various drum sounds across the entire note range of the keyboard, where each key plays a different drum sound. Henceforth, I'll simply use "drum kit" in lieu of Roland's "Rhythm Set" designation). Roland typically offers several drum kits to choose from (but sometimes you can only use 1 at a time). For example, there may be an "808 Rhythm Set" which features electronic drum sounds mapped across the keyboard. There may also be an "Orchestral Rhythm Set" which features orchestral percussion such as various timpani pitches and concert cymbals mapped across the keyboard.

Just like with patches, sometimes Roland has a few "preset" drum kits in ROM. You can't edit the settings of these kits. They may be labeled "Preset A", "Preset B", etc, just like the ROM patches. There may be one or two kits in RAM, whose settings you can edit. These are typically labeled "User (or Internal) A", "User B", etc.

The voice expansion board typically is limited to playing 1 drum kit stored in its own RAM. As with the board's RAM patches, you can edit this kit's settings (but those settings need to be saved to external media in order to be permanently saved), or load another kit into the board, thus overwriting that kit's settings.

I'll discuss the editing and creation of drum kits later.

Performances

Roland often has groups of things referred to as "Performances". For example, there may be a group of Performances labeled "Preset A" in ROM. There may be another group of Performances labeled "User A" in RAM. A Performance is mostly just a "setup" of various parameters, often having to do with multi-timbral operation, such as MIDI channel assignments, but also other settings such as settings for built-in hardware effects, what MIDI messages the keyboard generates when you play it,

how the Roland Patches respond to incoming MIDI messages such as Volume, Pan, etc, controllers, etc. I'll get into more details about Performances later.

The voice expansion board typically has only 1 Performance in its own RAM, and it is limited to playing only this Performance. You can load another Performance into the board, thus overwriting that 1 Performance with new settings, but that involves sending system exclusive messages to the board's MIDI input. In fact, the voice expansion's Performance is limited to playing only those 8 Patches also loaded into its RAM, as well as the 1 drum kit loaded into its RAM. The voice expansion is essentially a "mini JV-880" in of itself, rather than an actual augmentation of the main module.

Performance Play and Patch Play modes

"Performance Play mode" is essentially the module's multi-timbral mode, whereas "Patch Play mode" is more suitable to playing only 1 patch at a time from the local keyboard, or quickly auditioning patches in a Patch Group when you're deciding which ones to select while creating a given Performance. Some Roland modules disable multi-timbral operation when in Patch Play mode, and/or automatically set the local keyboard to play the currently selected Patch regardless of whether the keyboard is set to broadcast upon the same MIDI channel as the Patch is set to receive. This can be a boon when you're auditioning Patches to be used in a Performance, as it's often a lot more hassle (especially upon older modules) to audition Patches in "Performance Play mode" than it is to merely drop back into Patch Play mode.

Waveforms/Tones/Patches/Performances

Typically, Roland modules contain numerous digital waveforms in ROM. These are the raw "sounds" that you use when creating Tones, and ultimately Patches and Performances. Some Roland modules offer a special "waveform play" mode where you can hear the raw waveforms without any kind of processing such as effects or filtering applied. (The non-percussion waveforms are typically looped automatically so that you can use them in patches that "sustain", but other than looping, no other settings are applied to the raw waveform itself). But upon other modules, you may have to create a Tone using that waveform in order to hear it. Usually, Roland lists the names of all raw waveforms in the user manual under "Waveform List". (Adding a wave expansion board will give you even more waveforms with which to create Tones, and that board will usually come with a list of its raw waveforms).

A Roland Tone is merely **one** of the raw waveforms with various settings, such as volume, tone, and pitch shaping applied to it. The Tone has settings for a Voltage-controlled Amplifier (ie, VCA), Voltage-controlled Filter (ie, VCF), Voltage-controlled low frequency oscillator (ie, VCO), and other settings that can be applied to the raw waveform to change its tone, pitch, volume, etc.

For example, a particular Tone may be using a raw waveform named "Jazz Organ" (ie, a waveform of a typical Hammond B-3 sound). You could use that Tone's VCA to cause the waveform's initial attack (ie, volume) to slowly fade in. You can then create another Tone that uses the exact same

waveform, but alter the Tone's VCA settings so that the waveform has a fast attack, and perhaps use that Tone's VCO to add some constant vibrato.

In conclusion, Tones are used to create a variety of sounds based upon the raw waveforms, using VCA, VCF, VCO and other Tone settings applied to some raw waveform. (Note: Lately, the Roland manuals have begun referring to Tones as "Waves". And the description for the Tone settings is within the chapter that describes editing a Patch since nowadays each Patch in Roland modules has its own individual 4 Tones. Contrast this with older designs where Patches had to pick 4 Tones from among a more limited number of shared Tones. In other words, nowadays a Patch contains 4 unique Tones, and their settings, embedded inside of it. For example, in the JV-90 manual, the Tone settings are described in the "Patch Edit mode" chapter, starting with the heading of "Parameters accessed by pressing [WAVE/LFO]).

A Roland Patch typically consists of upto 4 Tones set to play across the keyboard range. (A Patch isn't required to use all 4 Tones. It can use only 1, 2, or 3, if desired. If you don't want to use one of the Tones of a Patch, then switch that Tone to "Off").

Of course, the Patch has various settings that can be applied **individually** to these Tones.

For example, the Patch has a volume setting for each Tone. So, if one Tone happens to be inherently louder than another Tone, but you want them to sound equally loud within the patch, you can balance their respective volumes. (Note: In newer Roland modules where each Patch has its own unique 4 Tones, the individual Tone volumes are actually among the Tone settings). Or, if you deliberately want some Tone to be very soft (ie, maybe to "flavor" the sound without being a prominent component of the sound), you can turn down its overall volume.

So too, the Patch has a pan setting for each Tone. For example, you can have one Tone panned to the right (ie, set to sound out of only the right "Audio Out" jack) and another Tone panned to the left to create a stereo effect between the Tones. (This is particularly useful for creating spacious "stereo patches" of a single instrument. Typically what you do is create 2 Tones that use the same waveform, but apply slightly different Tone settings to them, for example, making one slightly sharp and the other slightly flat, or use the VCF cutoff to make one brighter than the other. Then you pan those 2 Tones in stereo within a given patch. See my article Arranging for MIDI sound modules for ideas on creating stereo patches. Just substitute the word "Tone" for the word "Part" when reading the various, suggested techniques).

Also, the Patch has separate settings for the note range over which each Tone plays. So, rather than stacking the Tones so that they play over the same note range (and thus building up a thicker, perhaps stereo sound), you can limit the Tones to playing in only specific note ranges. For example, you can limit one Tone to playing over only the top 3 octaves of the keyboard, and limit another Tone to playing over only the bottom 3 octaves. (ie, You can set up "split keyboard" patches, where one Tone using a given waveform plays over a certain note range, and another Tone using an entirely different waveform plays over a different note range).

You may also be able to set the velocity range over which each Tone plays, so that, for example, playing loudly sounds only 1 Tone whereas playing softly sounds a different Tone only (ie, velocity switching or cross-fading). For example, you can setup a patch to switch between playing a picked or slapped bass note depending upon how hard you press the key.

There may be other Patch settings that determine how received MIDI controllers such as Volume, Expression, Hold Pedal, etc, or MIDI messages such as Pressure (ie, aftertouch), Mod or Pitch Wheel affect the Tones. For example, you may be able to set increasing MIDI Pressure to increase the depth of the LFO applied to a Tone (ie, increase a vibrato effect).

Upon newer modules, there may also be settings for effects such as Reverb and Chorus (although, except for the Patch's level settings for Reverb and Chorus, these are typically overridden by Performance settings when in "Performance Play mode").

A Roland Performance typically consists of several Patches set to play simultaneously (ie, in multi-timbral mode), with the keyboard (or some external controller) perhaps playing one (or maybe even more) of those patches. Actually, a Performance consists of several "Internal Parts" in Roland terminology, but since you assign 1 Patch to every "Part", you can think of a Performance as consisting of several, simultaneously playing patches. For modules that have a built-in keyboard such as the XP-50, a Performance also has settings for how the keyboard generates MIDI messages (ie, how the keyboard functions as a controller).

Older Roland modules have Performances that are limited to playing only 8 (or fewer) patches simultaneously (ie, the Performance is limited to 8 "Internal Parts"), and ultimately limited to playing patches upon only 8 of the 16 available MIDI channels simultaneously. With the addition of a Voice Expansion board, that board gives 8 more Parts, and can therefore be set to handle the remaining MIDI channels for full multi-timbral support. (But be aware of the limited patch storage capabilities of some Roland voice expansion boards, mentioned earlier. I really wish Roland had designed the voice expansion board to access the main RAM and ROM patches/performances in the module, instead of being limited to accessing patches and the performance only within its own limited RAM. On the other hand, one really nice thing is that, because the board can be treated as its own module, with its own MIDI input disconnected from the main MIDI input, if you have a multi-bus MIDI interface, you can greatly reduce MIDI data clogging by running separate MIDI Outs to the **disconnected** MIDI inputs of the voice expansion board and main module).

Each Performance "Internal Part" can be assigned to a MIDI channel. Normally, you'd set each Part to a unique MIDI channel (so you have independent MIDI control of each Part).

Then you assign one Patch (from among all of the various groups of Patches) to each Part. Of course, you can always change a Part's Patch **at any given moment** by sending the module a MIDI Program Change message upon that Part's MIDI channel. So, think of this assignment as merely a "default", temporary setup which is automatically initiated whenever you select this particular Performance, but which can be dynamically changed over MIDI very quickly and easily. Further note that the various

Parts of a Performance can choose their Patches from among any group. For example, one Part can be using a Patch in "Preset A" Patch Group and another Part can be using a Patch on a group in the wave expansion board. (One exception is with the voice expansion board. As mentioned, the board can access only the 1 Performance, and its 8 Patches and drum kit, currently loaded into its RAM. And the Performances in the main module can't access the voice expansion's Patches).

Each Part has its own individual volume so that you can balance the volume of its Patch against the other Parts' Patches. It typically also has individual pan and tuning settings. (Think of these as master settings which affect any respective settings you made when editing the Patch). Note that a Part's volume and pan are dynamically changeable via MIDI controller messages on that Part's MIDI channel, just like the Patch assignment can be changed over MIDI. Again, think of these as merely "default" settings.

There may be other Part settings such as whether it responds to various MIDI messages such as Hold pedal or Volume controller. (For example, maybe you don't want external gear to be able to change that "default" volume you set for the Part. In that case, disable receipt of Volume MIDI messages).

The Performance also has reverb and chorus settings that are applicable to all Patches. (On older modules, the individual Reverb and Chorus levels for each Part are set in that Patch's settings. On newer modules, the Performance itself may have individual effects levels for each Part which override the Reverb/Chorus levels in the Patch settings). These settings are active only in Performance Play mode (except if you're using an older module whose Patch settings don't include settings for the effects, in which case the Performance settings are active in Patch Play mode too).

There are also settings for how the module responds to some keyboard controller (or in the case of modules with a built-in keyboard, how the internal sounds respond to that keyboard). Note that the JV modules that have built-in keyboards were also designed to function as MIDI controllers of other MIDI modules (ie, besides the internal sound module). For this reason, you have separate settings for what MIDI messages the keyboard generates, versus how the internal sound module responds to the keyboard. The Performance settings that apply to how the module's internal sounds (ie, its "Internal Parts") respond to the keyboard are typically referred to under the heading "INT ZONE". The Performance settings that apply to how the keyboard generates MIDI messages are typically referred to under the heading "TX ZONE". (Obviously, rackmount modules have no keyboard, and therefore would have no "TX ZONE" settings). These are entirely independent settings. In other words, you can setup the keyboard not only to control the internal sounds while also generating MIDI messages to control other modules, but also have entirely different ways of controlling both. For example, you can set the module's Pitch Wheel to affect the internal sounds, but not actually transmit Pitch Wheel messages to the MIDI Out. You can set the keyboard to play Internal Parts that are assigned to MIDI channels 1 and 2, but have the keyboard simultaneously generating MIDI messages upon channels 4, 5, and 6 instead, for example. So, you need to think of the keyboard's internal sound routing, and the MIDI output setup as two entirely different setups. (This can sometimes be confusing to folks who are accustomed to modules whose keyboards are more closely tethered to the internal sound module. While Roland's approach can be very flexible, it can also confuse a guy when he's got a situation such as the preceding MIDI channel setup. Let's say that he records a part into his sequencer. During recording, there's no problem because he's got the local keyboard playing his two Internal Parts

assigned to channels 1 and 2. But the module is actually outputting messages upon channels 4, 5, and 6. Now when he plays back that sequencer track, it's going to play whatever Internal Parts are assigned to channels 4, 5, and 6, rather than the Parts assigned to than channels 1 and 2).

Here I want to delve into a further discussion of Parts. As mentioned, a Performance has "Internal Parts". An Internal Part is simply a Part that plays some internal Patch in the sound module. An Internal Part can be played by the keyboard, or some external controller (including a sequencer). The INT ZONE settings determine how the keyboard/sequencer plays these Internal Parts (ie, the INT ZONE settings are for the 8 Internal Parts).

A Performance (upon a module with a built-in keyboard) also has "MIDI Parts". A MIDI Part does absolutely nothing except broadcast MIDI messages to MIDI Out when you play the keyboard. Typically, there are as many MIDI Parts as there are Internal Parts. For example, a JV-90 has 8 MIDI Parts. The TX ZONE settings determine how the keyboard generates MIDI messages when you play it (ie, the TX ZONE settings are for the 8 MIDI Parts). Like with Internal Parts, each MIDI Part can be assigned a MIDI channel, and then assigned to "play" over a certain note range of the keyboard. What do I mean by "play"? I mean that when you press down a key within that note range, the MIDI Part sends a Note-On message to MIDI Out upon that Part's MIDI channel. The MIDI Part itself makes no sound because it is not attached to anything inside of the module. It connects only to MIDI Out. When you move the Pitch Wheel, the MIDI Part sends Pitch Wheel messages to MIDI Out upon that Part's MIDI channel. Since the JV-90 has 8 MIDI Parts, its keyboard can simultaneously broadcast upon upto 8 different MIDI channels. (And just like with Internal Parts, if you don't want to use all 8 MIDI Parts, you turn off the unneeded ones). So, you can setup "MIDI splits", for example, setting two MIDI Parts to broadcast upon different channels and limiting one MIDI Part to "playing" over only the bottom half of the keyboard, and the other MIDI Part playing over only the top half of the keyboard. (Incidentally, most JV modules have buttons pertaining to the Parts. For example, the JV-90 has 8 buttons that are used to turn on/off Parts when in "edit mode". If you're editing INT ZONE settings, then these buttons select the 8 Internal Parts. As soon as you switch to editing TX ZONE settings, these same 8 buttons now turn on/off the 8 MIDI Parts).

Let's take another example that shows the difference between MIDI Parts and Internal Parts (ie, the difference between TX ZONE and INT ZONE settings). Let's say that you want to transpose all of the Parts of only 1 Performance to play a half step higher. Rather than using the master transpose (which would affect all Performances you select), you want to transpose only this 1 Performance. (Maybe you want to set up 12 Performances, each transposed a different amount, so that you can easily change the key of any song just by selecting the Performance with the appropriate transpose amount. This is just an idea of some things you can do with Roland's approach). OK, you go into your Performance's INT ZONE settings. For each of the 8 Parts, you set its TRANSPOSE setting up a half step. Now, you play the keyboard and you do indeed hear any Patches being played a half step up. Next, you record a sequencer track, and play it back. The patches are no longer playing a half step up, but rather, have reverted back to normal tuning. What went wrong? Well, you forgot to go into the TX ZONE settings, and set all of your MIDI Parts to transpose up a half step. (ie, The MIDI Parts have their own TRANSPOSE setting under TX ZONE). The net result is that, while the keyboard did indeed transpose its notes when playing the Internal Parts, it simultaneously generated MIDI note numbers that weren't transposed -- and your sequencer recorded the latter. When you played the track back, those non-transposed MIDI notes were sent to the Internal Parts.

In conclusion, you **must** understand the differences between the INT ZONE and TX ZONE settings in order to setup a Roland Performance properly. (I can't stress this enough as an understanding unlocks the potential of having a very versatile controller, as well as eliminating a lot of the problems you may be having with getting the sequencer playback to sound the same as what you heard when you recorded the track). Usually, you'll want to have them correspond to each other, for example, in that sequencer scenario, you'd want the TX ZONE settings to be broadcasting MIDI messages upon channels 1 and 2, to coincide with having set the INT ZONE settings such that the keyboard played the Parts which were assigned to channels 1 and 2. But sometimes, you may want the TX ZONE different than the INT ZONE. For example, I have a separate MIDI module which I use to play my left hand bass parts. I wanted it so that my JV-90 Performances played internal sounds only upon the upper half of the keyboard, and the bottom half of the keyboard played only that external module. Here's what I did:

1. In all of my Performances, I limited the Internal Parts (that the keyboard was set to play) to only the upper range of the keyboard. I did this using the KEY RANGE UPPER and LOWER settings of INT ZONE.

2. I made sure that no Internal Part was assigned to MIDI channel 2, as that channel was being reserved for my external module.

3. I set one MIDI Part to "play" over only the lower half of the keyboard (using the KEY RANGE UPPER and LOWER settings of TX ZONE), and assigned it to channel 2 (using TRANSMIT CHANNEL of TX ZONE). I likewise set other MIDI Parts (ie, as many MIDI Parts as I had Internal Parts being played by the keyboard) to play over only the upper half of the keyboard, and assigned them to the same MIDI channels as those Internal Parts which the keyboard was playing.

The net result is that, no matter what Performance I select, I always have a MIDI split happening, and the JV-90's internal sounds are limited to playing over only the top half of the keyboard. If I record the JV's MIDI Out into a sequencer, and play that track back into the JV-90 and my bass module, then it will properly playback with that same MIDI split.

There may be other settings in the TX ZONE and INT ZONE. For example, there is a velocity curve. Maybe you have some external module that has a vastly different response to note velocity than the Roland's internal sounds. (ie, Perhaps when you use the Roland keyboard to control the external module, the module plays too softly upon soft notes and/or too loudly upon loud notes). So, you can set the VELOCITY CURVE under TX ZONE for the MIDI Part which is controlling that external module. And yet, for the Internal Parts, and the other MIDI Parts which you've setup to correspond to those Internal Parts, you have separate VELOCITY CURVE settings.

There's a lot of flexibility in using a Roland keyboard as a controller. Explore it. Use it where it helps you.

Finally, we come to the issue of the "Rhythm Part". Every Performance has at least one Rhythm Part.

Sometimes, the last Internal Part is hard-wired to being the Rhythm Part (ie, the "patch" that you select for it must be one of the drum kits, or Rhythm Set as Roland calls them). On newer modules, one or more of the Internal Parts may be set to play a drum kit. Although the Rhythm Part's settings globally affect whichever drum kit is selected (much like any other Part's settings globally affect a Patch), the individual settings of a given Drum Kit are edited in "Rhythm Edit mode". (And just like with other Parts, the drum kit that the Rhythm Part uses can be changed on-the-fly with a MIDI Program Change message on the Rhythm Part's channel. So too, can the "default" volume and pan settings be altered with respective MIDI controller messages).

Creating/Editing a Drum Kit

When creating a drum kit, you need to visualize the entire keyboard range. To each key, you assign any one of the various raw waveforms. (The raw waveforms include a selection of drum sounds. It's true that you could select a pitched waveform instead, maybe a Trumpet waveform, and assign it to that one key in the drum kit. You could even assign that same waveform to several adjacent keys, and set each key to transpose the pitch a different amount in order to create a musical scale mapped over an area of the drum kit. But that's sort of an odd approach, unless maybe you want to try some microtonal setup since each individual key has its own fine tuning in most Roland modules' drum kits).

Each key has its own settings. For example, each key has its own volume so that you can balance the various drum sounds.

So too, each key has its own pan setting. For example, you can have one drum panned to the right (ie, set to sound out of the right "Audio Out" jack) and another drum panned to the left to create a stereo drum kit.

Each key typically has its own VCA, VCF, and VCO so that you can shape the drum's pitch, timbre, and volume (ie, decay).

There may be other individual (per key) settings such as setting how the drum responds to velocity, and choosing to have one key cut off the sound of another key (ie, to have a closed hi-hat cut off an open hi-hat sound).

There is usually also some global settings, such as for effects like Reverb and Chorus (although, except for the Rhythm Set's level settings for Reverb and Chorus, these are typically overridden by Performance settings when in "Performance Play mode").

Further reading

The article Understanding the MIDI Implementation of your Roland manual guides you through your manual's MIDI Implementation chapter. The RECEIVE DATA section contains useful information in

helping you to understand what MIDI messages your sequencer (or other device) can send to your Roland Sound module to control its Parts. In other words, this may help you understand what sorts of things you're likely to be able to setup in INT ZONE. The TRANSMIT DATA section helps you to deduce what kinds of MIDI messages your Roland can send to other devices (ie, when it is used as a MIDI controller). In other words, this may help you understand what sorts of things you're likely to be able to setup in TX ZONE.

Also, the article Changing patches over MIDI using Bank Select Controller has an example for a Roland JV-90. It's use of Bank Select controller is similiar to other Roland Sound Modules, and may help you in understanding how to utilize your manual's listing under the "Bank Select" sub-subsection of the RECEIVE DATA / CONTROL CHANGE section.

The Roland Audio Card FAQ has information specific to Sound Canvas and RAP-10 units. The more basic Sound Canvas models often have hardwared Tones that you can't edit, limited Patch editing, and also only one multitimbral setup with limited editing (ie, 1 Performance which likely isn't even identified as such), for example, the SC-7. So, a lot of the programmability described in this article is not applicable to those units.

Questions about Roland architecture

Questions in this FAQ are:

"How do I pick out a Performance that uses the Patches I want?"
"What's this 'control channel' nonsense?"
"Why is my sound module playing only some of the MIDI channels?"
"How do I check what Patches a given Performance has assigned to it by default?"
"How do I edit (and save those edits of) a Preset Patch/Performance in ROM?"
"What do you recommend doing after one first gets a Roland module?"

I want to use my module in Multi-timbral mode. Does that mean that I have to be in "Performance Play mode" and must find a Performance that happens to have the particular Patches that I want to use?

Yes, you should be in Performance Play mode, but I'm not sure if you necessarily need to be searching for a Performance that is already setup with the desired selection of Patches. Rather, it seems to me that your only concern is switching around the Patches assigned to the Internal Parts of **any** given Performance. You can do that by simply sending the Bank Select and Program Change messages to change a Part's Patch assignment. For example, assume that, for the currently selected Performance, you've got 16 Internal Parts, and you've set them to the 16 MIDI channels (ie, Part 1 is

set to channel 1, Part 2 is set to channel 2, etc, Part 16 is set to channel 16). Now, at the start of the song, you maybe set Part 1 to play the "Trumpet" Patch. You do this by sending a Bank Select and Program Change messages on MIDI channel 1 to select the Trumpet Patch. In the middle of the song, you wish to switch this to the Oboe Patch. You simply send the Bank Select and Program Change messages on channel 1 to select the Oboe Patch, and then Part 1 starts playing the Oboe Patch -- all without changing to another Performance.

Remember that all a Performance is, is the 16 (or however many) Internal Parts and their settings (ie, which MIDI channel each Part uses, each Part's volume, each Part's Patch assignment, each Part's MIDI response such as whether the Part responds to various controllers, what key range over which the Part plays, whether the Part is layered with other Parts or perhaps split, etc). But remember that many of these Part settings can be quickly altered on-the-fly during playback via MIDI messages. (ie, Volume controller changes the Part's volume, Pan controller changes the Part's pan position, Bank Select and Program Change alter the Patch used by the Part, etc). So for merely switching the instrumentation of the 16 Internal Parts, and altering Part settings such as volume, pan, etc, you only need send appropriate MIDI messages to the Performance's Parts (ie, on their respective MIDI channels). It may be easier to use your computer sequencer software to pick out the desired Patches (from a nicely displayed graphical listing of all of the names), and let it send the Roland module the appropriate Bank Select and Program Change messages. This is really easy if your sequencer is setup with a "Definition File" for your Roland module. Simply manually switch to a Performance that, while it may not have the 16 Patches that you want assigned by default, is otherwise setup as desired (ie, all 16 Parts are enabled, and they are set to different MIDI channels. You may even want to create such a "template Performance" for use with your sequencer).

Of course, there are times when you want to change Performances. Remember that the effects and other global settings are also settings of a Performance (ie, whatever reverb to use such as Hall or Plate or whatever, etc). Furthermore, the keyboard settings are made in the Performance (ie, which Parts the keyboard is set to play, what kind of scale tuning is employed across the keyboard, choosing which velocity curve you desire for note velocity, etc). So, you need to change the Performance when you need to change these particular settings, for example, switching from one Performance that has the effects set to use a Hall reverb, to another Performance that has the effects set to a slap echo.

Otherwise, you won't need to search for a Performance that has the desired Patch assignments, nor even create such a Performance yourself (although I think that you should create a few Performances as "templates" which you frequently use. It makes things easy if you got a Performance already setup with 16 desired patches, and their respective volumes, and with the effects setup as desired, routed to the desired outputs, etc, etc, etc. After all, there is some Performance setup that you can't do over MIDI, for example, set the MOD wheel to control filter cutoff for a particular Part. You have to edit the Performance anyway to set that up).

For the purposes of sequencing, I suggest that you setup the first 16 Performances as templates. Turn off the Control Channel (described later). In all 16 Performances, turn on all of the Internal Parts, and set each to a different MIDI channel (ie, set Part 1 to MIDI channel 1, Part 2 to MIDI channel 2, etc... and Part 16 to MIDI channel 16). In each Performance, set the keyboard to play only 1 Internal Part, but assign a different Part than the other Performances. In other words, for Performance 1, set the keyboard to play only Internal Part 1. For Performance 2, set the keyboard to play only Part 2. Etc. In

each Performance, turn on only 1 MIDI part, but assign it to a different channel than the other Performances. In other words, for Performance 1, set that MIDI Part to channel 1. For Performance 2, set that MIDI Part to channel 2. Etc. In this way, you can pick out the respective Performance based upon the MIDI channel you wish to use. For example, if you want to record a sequencer track where the MIDI data is on channel 3, then manually select Performance 3. If you subsequently want to record another sequencer track upon channel 16, then manually select Performance 16. (Of course, you'll need to put Bank Select and Program Change events, as well as Volume and Pan controllers, at the beginning of your sequencer track to make sure that each Performance you select gets automatically setup with the proper selection of Patches and mix). The net result is that you'll be able to quickly and easily reassign the MIDI channel that the keyboard generates messages upon, and still keep it playing the desired Internal Part, simply by manually selecting from among these 16 Performances.

What's this 'control channel' nonsense?

Normally, MIDI Bank Select and Program Change messages change the Patch assigned to a given Part (ie, the Part that is assigned to the same MIDI channel as the message is broadcast upon).

But what if you want to change from one Performance to another Performance using MIDI messages? Well, that's where the control channel comes in. When the control channel is not off (ie, it's set to one of the 16 MIDI channels), that's the channel which is used to control the Performance settings themselves (rather than an individual Part within the current Performance). For example, sending a Bank Select controller, followed by a Program Change, both on the control channel, causes the module to change to another Performance (perhaps one within another group of Performances).

There's one caveat to this scheme. The control channel eats up certain messages on its channel before any Part sees them. For example, assume that you assigned Part 1 in the Performance to channel 16. You have also set the control channel to 16. The net result is that you'll never be able to change that Part's Patch assignment (from its default). Why? Because the control channel grabs those Bank Select and Program Change messages on channel 16, using them to switch Performances, before the Performance's Part ever sees those messages. (Indeed, the Part never does see them). Hence, the Part never can have its Patch assignment changed over MIDI.

In other words, you can't have the same MIDI channel used to control both a Part's settings as well as control the Performance. So, if you have 16 Parts, but want to use the control channel, then 1 Part (that is set to that same MIDI channel) will always lose some MIDI control. I prefer to turn off Control Channel, and manually change Performances when needed (which isn't often since I tend to use external MIDI control to alter any given Performance's Parts as desired, rather than switching among various Performances).

By default, the control channel is often turned on, and set to the last (ie, Rhythm) Part's MIDI channel.

Why is my MIDI sound module playing only some of the MIDI channels? I played a MIDI file that has tracks on MIDI

channels 1 to 16, but I heard only some of the tracks.

First, make sure that you're in "Performance Play mode". Some modules disable multi-timbral operation in other modes.

Secondly, does your sound module support all 16 MIDI channels simultaneously in its multi-timbral mode? Some older models do not. For example, a JV-90 has only 8 "Internal Parts", which means that it can play a maximum of only 8 MIDI channels simultaneously. You get to pick out which 8 channels to play (ie, by setting the MIDI channel for each Part), but you're still limited to 8 MIDI channels at any given moment (unless you get a voice expansion board).

Third, did you check that the Performance you selected has all of its Internal Parts turned on, and assigned to different MIDI channels? It does no good to select a Performance that has all but 1 Part turned off, or has all of the enabled Parts set to the same MIDI channel, if you want to play a MIDI file with multiple tracks upon multiple channels.

Furthermore, if you've got a voice expansion board, make sure that the 8 Parts of its Performance are set to different MIDI channels than the 8 Parts in the Performance currently selected for the "main module". Remember that the voice expansion has its own 1 Performance that it always uses, and is totally separate from the main module's Performances. You don't want the voice expansion's Performance to be duplicating the same MIDI channel assignments. You want it covering the remaining MIDI channels. Also, make sure that your sequencer is actually getting data to the voice expansion. If you don't have a direct connection to the expansion's MIDI IN jack, then flip the switch to connect it to the main MIDI input.

How do I check what patches a given Performance has assigned to it by default? I could guess that a Performance with a name of "Vibes" may have a vibraphone Patch assigned to a Part that the keyboard is playing, but what about the Patch assignments for the other Parts? How am I to know what Patch assignments the other Parts have?

Well, you **don't** really know what Patch each of the 16 Parts is assigned in any given Performance offhand. But, you can press a button and get a listing of the Patch number for each Part, and then use the list in your manual to look up the corresponding name. Let me explain how I do it on my JV-90.

Say that I call up the first Performance in the User Group. It's name is "Plucky". What Patches are the JV-90's 8 Internal Parts using? Well, there are a group of buttons on the left which Roland calls the "Edit Palette" buttons. Each of these buttons has a different function depending upon whether you're in Patch or Performance mode (or you can even get into modes whereby Roland cuts off all the effects and filters and plays the raw waveforms. There are other modes you enter where you're selecting the Patches and/or Performances on Wave Expansion cards or Data cards or whatever, and

these buttons may take on different functions in those modes too). So, there's a button that, when I'm in Performance Mode, displays the Patch numbers for the 8 Parts. It says "Patch" over the button (ie, in the row that is labeled "Perform". In the row that is labeled "Patch", that same button has the label "Wave" over it, to allow me to find out what 4 Tones are being used by a Patch when I'm in Patch mode). Suddenly, my display changes and it shows me which Patch number is used by each Part. For example, maybe its says that Part 1 is using patch A71 (ie 71 in Preset A patches). I look up A71 in my Patch list in the manual, and it says "Nylon Gtr 1". So, that's what Part 1 is using for this Performance. Of course, if you alter the Patches from the factory defaults, then the manual's listing is no longer applicable. You should keep a list of what customizations you make.

There's a nice Preset Patch/Performance I like which, with minor edits, would be great. The problem is that it's a Preset in ROM, and I can't save the edits to that same Group. What do I do?

Call up the Patch/Performance you like. Get into Patch/Performance Edit mode. Make all of the changes that you want. But, before you exit from edit mode, first choose to WRITE the Patch/Performance (ie, perform a Write command) to some location within a RAM (ie, User or Internal) Patch/Performance group. Sure, you'll have to overwrite whatever Patch/Performance was previously stored in the user location you finally choose, but if it's a Patch/Performance you'll never use, you don't care.

What do you recommend doing after one first gets a Roland module?

1. Read this FAQ.

2. Read it again. Some people have poor reading skills, and they miss or deliberately skip over important points the first time. Worse, they tend to "fall asleep at the wheel" unless they're kept awake with periodic excursions into crass, obnoxiously provocative humor -- much of which is readily available at this web site. Indeed, I blame all of my many lapses into bad taste upon people with poor reading skills. If it wasn't for having to keep them awake, I could instead concentrate upon delivering an unwavering, myopic, textbook (ie, of no practical use) drone like the typical college instructor does.

3. Read the manual. (Notice a trend here? And yes, I'm actually one of those guys who, when he gets a new piece of gear, excitedly unpacks the manual first and reads it cover to cover, quite willingly, before even plugging in the gear. That's why I **can't** buy Casio gear. By the time I'm halfway through the pigeon-english in the manual, I'm already tying the noose to the rafter).

4. Dump the factory Patches/Performances. These things are just designed to hook impressionable listeners in music showrooms. They're designed to be played, and hopefully sound decent, upon crummy, mono PA systems. They're typically gawdy effects with

dreadfully slow attacks and fade-outs for the sake of the nouveau keyboardist who is buying his first keyboard and doesn't know how to play beyond a snail's pace nor use the sustain pedal. I've discovered that many of the raw waveforms aren't ever even used in the factory Patches (so you won't hear them unless you make your own Patches), and they're some nice raw material. Furthermore, there typically aren't any Patches with nice stereo imaging. That makes a world of difference in the "thickness" and "complexity" of a Patch. I seldom ever use any Roland factory Patches, and of the few I've liked, they all needed a bit of tweaking to get them sounding better.

5. Get your divorce papers ready for filing. Your wife will have left you by the time you get around to experimenting with Performance Edit mode... if she knows what's good for her.

Windows does not yet have a MIDI Manager. What this means is that it doesn't really arbitrate access to MIDI Input and Output intelligently between applications. Essentially, it leaves it up to each device driver to arbitrate access to its MIDI Input and Output (and without any standardized means for the driver to provide information about its status to the application). Needless to say, there are some caveats to this method (and general guidelines you should follow) as described below.

Single client drivers -- Device is busy errors

Many drivers are single client. Such a driver allows only one call to open its MIDI In to succeed (regardless of whether that Input is opened via midiInOpen(), or a high level MCI call). The driver also allows only one call to open its MIDI Out to succeed (regardless of whether that Output is opened via midiOutOpen(), a high level MCI call, or even midiStreamOpen()).

For example, if your application has successfully opened that driver for output (via a midiOutOpen call perhaps), then it owns that driver's MIDI Out. (ie, Your application has exclusive access to that MIDI Output). If some other application tries to subsequently open that Output (perhaps via midiOutOpen), then that application's call to midiOutOpen() will fail (and usually return an error number that is translated to a message of "Device is busy" by midiOutGetErrorText).

Only when your application finally closes that MIDI Output (via midiOutClose perhaps) will the "lock" upon that particular Output be released, and then another application can successfully open that Output.

Of course, the above example also pertains to MIDI Inputs (ie, midiInOpen, etc).

What is the result of this? Well, it means that, despite your application running under a multitasking system (ie, Win32), if your application opens a MIDI Input or Output and "hogs" it by keeping it open the entire time that your application is running, then the enduser will not be able to simultaneously run other MIDI software that wants to use that MIDI Input/Output. He'll just get "Device is busy" error messages from that other application every time he tries to do something with that application's MIDI Input/Output. You'll effectively be preventing him from multitasking another MIDI application in conjunction with that particular MIDI Input/Output. He'll have to close down your application every time he wants to use that other application with that MIDI Input/Output. And if the other application hogs that Input/Output too, then the reverse is also true. (ie, Every time he wants to run your application, he'll have to close down that other application).

This is very annoying to an enduser. It turns his multitasking operating system into something that looks like a singletasking operating system -- and one that doesn't manage resources.

So what is one way of getting around this? Well, a simple solution is not to hog the Input/Output. Don't open the MIDI Input/Output until you're actually ready to do some MIDI input/output (ie, midiOutShortMsg, midiOutLongMsg, doing some input/output via MCI, etc). Do that MIDI input/output as quickly as possible, and then immediately close that MIDI Input/Output. The net result is that you'll only be preventing multitasking for a short time while you're actually doing some input/output. If the other applications are doing the same, then everyone will be happy, as the enduser can switch between those applications running concurrently, and as long as he doesn't try to make both applications do MIDI input/output simultaneously (which can be a bad thing as you'll learn below), his applications will be sharing the MIDI Input/Output.

For example, let's say that you're writing a patch editor for a sound module. You need to send it a System Exclusive message to request it perform a data dump. Then you need to input a bunch of System Exclusive messages until you receive the entire data dump. Here's the approach you could take:

1. Give the enduser a means to start the operation of receiving a dump. For example, maybe have a button labeled "Receive Dump" upon which he clicks to begin the operation. Do **not** open MIDI Input or Output until he clicks upon that button.
2. When he clicks upon the button (to start the data dump), open MIDI Out with a call to midiOutOpen(). Also open MIDI In with a call to midiInOpen() (and setup some means of collecting the incoming bytes that Windows will pass to your application).
3. Send your "dump request" System Exclusive message to MIDI Out.
4. Collect the incoming MIDI data that Windows passes to you.
5. When you see that you have a complete dump from the sound module, call midiOutClose() and midiInClose().

The above solution is all well and good for some applications, but it's an impractical solution for some other types of applications. Let's take the example of a "MIDI Input Viewer" program. It "displays" each incoming MIDI message in realtime (ie, immediately when the message comes into the MIDI In and is passed to the application by Windows).

Obviously, that application can't know ahead of time when MIDI messages will be forthcoming and how much data will be coming in (unlike the patch editor, which knows that a dump will be following shortly after that "dump request", and knows how many bytes to expect).

A slightly different approach could be to give the enduser a button to start recording (at which time you call midiInOpen) and a button to stop recording (at which time you call midiInClose). In this way, he could stop recording before he switched to another application, and therefore still have his applications share that MIDI Input. But there is an even better solution which makes sharing the Input/Outputs transparent to the enduser, as you'll learn later.

But as (bad) luck would have it, many programmers wrote MIDI software that doesn't even allow this minimal level of sharing. They wrote software that hogs the MIDI In or Out for the entire time that the program is running. Why? Well, sometimes it could be because the software allows access to numerous MIDI Inputs and Outputs simultaneously, and it takes too much time and trouble to open/close them all before/after each operation. If the software is a MIDI sequencer, such timing delays prior to recording/playing may be crucial. Therefore, the software simply leaves all inputs/outputs constantly open so that the software will be "ready to go" whenever the user starts it recording/playing. But this is still a poor excuse for why the software doesn't close those inputs/outputs when the enduser switches to some other application. This is easy to implement under Win32, as you'll learn later. After all, if the enduser is switching to another application, he probably doesn't really care whether the first application is ready to do input/output at that moment (and if he does, there's a simple solution to handle that case too, which I'll outline later).

Multiclient drivers -- The perils of multitasking without management

Because too many programmers wrote MIDI applications that hog the inputs/outputs, some manufacturers of sound cards and MIDI interfaces started shipping multiclient drivers with their products.

A multiclient (sometimes called multi-instance) driver is one which allows more than one application to open a MIDI input or output. (ie, More than one call to midiOutOpen or midiInOpen will succeed). In essense, the driver lets two or more applications simultaneously open a MIDI input/output, unbeknowst to each other, and the driver itself tries to manage (ie, mix) their MIDI output, and also provides the applications separate copies of each incoming MIDI message.

Lots of endusers scramble for multiclient drivers so that they don't have to shut down MIDI programs in order to run other MIDI programs.

But there is one problem here. The two applications don't know that they're both outputting to the same MIDI port perhaps simultaneously. And the driver may not be intelligent enough to know when that's a good or bad thing. Let's take the example of that patch editor above.

Let's say that when the patch editor sends a data dump back to the sound module, instead of sending one large, complete system exclusive message in a single call to midiOutLongMsg(), it breaks one System Exclusive message into numerous "blocks of data bytes" sent via several calls to midiOutLongMsg(). Well, hopefully the driver is smart enough to not allow another application to do some MIDI Output inbetween the other application's midiOutLongMsg() calls until the driver detects the end of a System Exclusive message. Otherwise, the other application will screw up the patch editor's dump.

And Windows provides no standardized way for an application to tell a driver to give it exclusive access to a port, so an application is at the mercy of a multiclient driver's behavior in handling multiple output requests.

So too, if both programs are simultaneously outputting MIDI note messages, they may be turning off each other's Note On messages (with Note Off messages) or stacking Note Ons for the same note number and resulting in "stuck notes". Or they could be doing other conflicting things which the driver doesn't resolve.

And all of this multiclient stuff, particularly in regards to System Exclusive message input and output, imposes unnecessay overhead upon the driver. A single client driver can be **much** more stream-lined and efficient.

In fact, multiclient drivers decrease efficiency and yield poorer performance where a single client driver would suffice. (And those "loopback drivers" are even worse. These really play havoc with the efficiency of MIDI input/output). A MIDI Manager that arbitrated access to multiple ports, in conjunction with application support, would not only be more efficient (because all of its drivers could then be single client, instead of every driver duplicating the functionality and overhead of a poor "MIDI Manager" inside of it), it could also be more flexible (for example, allowing applications to choose when they want to share an Input/Output and when they don't -- control which they do not have over a multi-client driver).

Needless to say, a multiclient driver is no substitute for a MIDI Manager that is capable of allowing applications some control over sharing an Input/Output. Mostly, musicians use multiclient drivers for the sole purpose of avoiding having to shut down programs that hog the MIDI Input/Output. Clearly, what is needed is a better way for an application to access a MIDI Input/Output without hogging it, but also having to avoid the speed overhead of opening an Input/Output before every operation and then immediately closing it afterward (which as I pointed out above, is not suitable for some types of MIDI applications). If the enduser had software that could do that, instead of hogging an Input/Output, then he wouldn't need multiclient drivers at all. In fact, his software could share Inputs/Outputs with the multitude of

single client drivers that are out there now.

Such a method is what we'll investigate now.

A more elegant, transparent way of sharing a MIDI input/output

If your application has a window of your own creation (and virtually all non-console Windows applications do), then a simple way of sharing a MIDI port with other applications, transparently to the enduser, is to process the WM_ACTIVATEAPP message. This message tells you whether your application is becoming active (ie, the user has selected any window that you created, and thereby given your program the focus) or inactive (ie, the user has selected a window created by another application, and therefore, if your application isn't already doing something in the background, it becomes inactive). In processing this message, you should close any MIDI ports you have open if the wParam argument is 0 (ie, you're going inactive) or reopen them if wParam is 1 (ie, you're active).

In this way, your application transparently releases the MIDI devices when the user switches away from it to another program. And your application also transparently reopens the MIDI devices when the user switches back to it. Because this processing is done in the WM_ACTIVATEAPP, it happens as soon as your application becomes active. This includes when your application initially starts, and so, the WM_ACTIVATEAPP code also serves to initially open your MIDI devices. Quick and easy.

What's so great about this? Well, you **can** hog the MIDI Input/Output to your hearts content... while your program is active (and the other MIDI programs are inactive). And when your program is inactive, you don't care about hogging the MIDI Input/Output. Now, you don't have to open/close the MIDI devices before/after each operation, and yet, you'll be sharing the MIDI ports with other applications -- even doing so transparently to the enduser, in a way that works with single client drivers too. That's what is so great about it.

Below is some example code to illustrate this. It contains a minimal WinMain() function (ie, program entry point for a Windows C program) which opens a window of my own creation. (For simplicity, I use a dialog template in a resource file, and call CreateDialog, but you could use CreateWindowEx). Its message handler is mainWndProc(). That's where I open and close MIDI devices by processing the WM_ACTIVATEAPP message. mainWndProc() references two globals, MidiInHandle and MidiOutHandle, which are initially set to 0 at program startup. Whenever these two handles are 0, I know that I don't have my MIDI In and Out devices open. By checking these globals for 0, this makes the code bulletproof. I sort of made "wrapper functions" for midiOutOpen(), midiInOpen(), midiOutClose(), and midiInClose() that operate upon those global handles, just to make it easier.

```c
DWORD MidiInHandle = 0;
DWORD MidiOutHandle = 0;

/* *************************** WinMain() ***************************
 * Program Entry point
 ***************************************************************** */

int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE hinstPrev, LPSTR lpszCmdLine,
int nCmdShow)
{
    MSG    msg;
    HWND   mainWindow;

    /* Create Main window */
    if (!(mainWindow = CreateDialog(hinstExe, MAKEINTRESOURCE(IDD_MAINWINDOW), 0,
mainWndProc)))
    {
        return(-1);
    }

    /* Show the window with default size */
    ShowWindow(mainWindow, SW_SHOWDEFAULT);
    UpdateWindow(mainWindow);

    /* Get the next msg (until WM_QUIT) */
```

```
    while (GetMessage(&msg, 0, 0, 0))
    {
        /* Send msg to window procedure */
        DispatchMessage((CONST MSG *)&msg);
    }

    /* Close any Midi Input device */
    closeMidiIn();

    /* Close any Midi Output device */
    closeMidiOut();

    /* Exit */
    return(0);
}

/* ************************** mainWndProc() *****************************
 * Main Window message handler called by Windows
 ********************************************************************
*/

long APIENTRY mainWndProc(HWND hwnd, UINT uMsg, UINT wParam, LONG lParam)
{
    switch(uMsg)
    {
        /* ============= App is gaining or losing activation ============ */
        case WM_ACTIVATEAPP:
        {
            /* Is app losing activation? */
            if (!wParam)
            {
                /* Close the MIDI devices (so other apps can use them) */
                closeMidiIn();
                closeMidiOut();
            }

            /* App gaining activation so reopen
            MIDI devices if not already open */
            else
            {
                /* Open MIDI In */
                openMidiIn();

                /* Open MIDI Out */
                openMidiOut();
            }

            /* Allow windows to continue handling WM_ACTIVATEAPP */
            break;
        }
    }

    // Let windows handle it
    return(0);
}
```

```c
/* ************************** closeMidiIn() **************************
 * Close MIDI In Device if it's open.
 ****************************************************************** */

DWORD closeMidiIn(void)
{
   DWORD    err;

   /* Is the device open? */
   if ((err = (DWORD)MidiInHandle))
   {
      /* Unqueue any buffers we added. If you don't
      input System Exclusive, you won't need this */
      midiInReset(MidiInHandle);

      /* Close device */
      if (!(err = midiInClose(MidiInHandle)))
      {
         /* Clear handle so that it's safe to call closeMidiIn() anytime */
         MidiInHandle = 0;
      }
   }

   /* Return the error */
   return(err);
}


/* ************************** openMidiIn() **************************
 * Opens MIDI In Device #0. Stores handle in MidiInHandle. Starts
 * recording. (midiInputEvt is my callback to process input).
 * Returns 0 if success. Otherwise, an error number.
 * Use midiInGetErrorText to retrieve an error message.
 ****************************************************************** */

DWORD openMidiIn(void)
{
   DWORD    err;

   /* Is it not yet open? */
   if (!MidiInHandle)
   {
      /* Open MIDI Input and set Windows to call my
      midiInputEvt() callback function. You may prefer
      to have something other than CALLBACK_FUNCTION. Also,
      I open device 0. You may want to give the user a choice */
      if (!(err = midiInOpen(&MidiInHandle, 0, (DWORD)midiInputEvt, 0,
CALLBACK_FUNCTION)))
      {
         /* Start recording Midi and return if SUCCESS */
         if (!(err = midiInStart(MidiInHandle)))
         {
            return(0);
         }
      }
```

```
      /* ============= ERROR ============= */

      /* Close MIDI In and zero handle */
      closeMidiIn();

      /* Return the error */
      return(err);
   }

   return(0);
}


/* ********************** closeMidiOut() **************************
 * Close MIDI Out Device if it's open.
 ************************************************************************ */

DWORD closeMidiOut(void)
{
   DWORD   err;

   /* Is the device open? */
   if ((err = (DWORD)MidiOutHandle))
   {
      /* If you have any system exclusive buffers that
      you sent via midiOutLongMsg(), and which are still being output,
      you may need to wait for their MIDIERR_STILLPLAYING flags to be
      cleared before you close the device. Some drivers won't close with
      pending output, and will give an error. */

      /* Close device */
      if (!(err = midiOutClose(MidiOutHandle)))
      {
         /* Clear handle so that it's safe to call closeMidiOut() anytime */
         MidiOutHandle = 0;
      }
   }

   /* Return the error */
   return(err);
}

/* ********************** openMidiOut() **************************
 * Opens MIDI Out Device #0. Stores handle in MidiOutHandle.
 * Returns 0 if success. Otherwise, an error number.
 * Use midiOutGetErrorText to retrieve an error message.
 ************************************************************************ */

DWORD openMidiOut(void)
{
   DWORD   err;

   /* Is it not yet open? */
   if (!MidiOutHandle)
   {
```

```
        /* Open MIDI Output. I open device 0. You
        may want to give the user a choice */
        if (!(err = midiOutOpen(&MidiOutHandle, 0, 0, 0, CALLBACK_NULL)))
        {
            return(0);
        }

        /* ============= ERROR ============= */

        /* Return the error */
        return(err);
    }

    return(0);
}
```

Note: For the sake of simplicity, I omit code above to notify the user if an error occurred in opening a MIDI device. That's usually something that he would want to know.

Of course, you would likely add menus and controls to your window, to allow the user to perform some MIDI operations. And you would add WM_COMMAND message processing to your window procedure to do that user interaction. One caveat is that, before you make any call to do MIDI output, you should first check that MidiOutHandle is not 0. After all, it may be that some other application has not been as nice as you, and didn't relinquish its death grip on the MIDI Output you're trying to use. In that case, the MIDI Output may not reopen. (Same thing with MIDI Input, but since Windows calls **you** for MIDI input, rather than you calling a Windows function, all that happens when MIDI Input doesn't reopen is that your MIDI input routines never get called).

There is one thing that I've noticed about some drivers. They allow an application to quickly return from a call to midiOutClose() or midiInClose(), but the driver then goes on to do some "cleanup" in the background. The driver may not yet be ready to allow someone else to do a midiInOpen() or midiInClose() during that time, and may return a "Device is busy" error. The net result is that, if another application is doing the same thing as above (ie, opening and closing devices in its WM_ACTIVATEAPP processing), there may not be enough time between when that application closes the device and when you try to open it. After all, you both are processing your WM_ACTIVATEAPP messages around the same time (on each of your timeslices).

I have found that a good work around is not to do the device opens while processing the WM_ACTIVATEAPP message. Rather, have that message post another, user-defined message to your window handler (ie, a message with an ID of WM_USER or greater). And then when your window procedure processes the WM_USER message is where you do the opens. This works because, when your application is getting the focus, there are usually a slew of messages sent and processed by your window procedure, and these get done well before you get around to that WM_USER message. Typically, a driver's background tasks get higher priority, so the driver will likely get done well before you get around to reopening the device.

Another concern I have observed under Windows 2000 (but not other versions of Win32) is when you go to close down your application. (ie, Close down all windows). You should close any MIDI handles you have before you call DestroyWindow() upon your last window. Otherwise, if you wait to close those handles after the call to DestroyWindow(), then the WM_ACTIVATEAPP message may be sent to another app which will try to open those same MIDI handles. (ie, During your call to DestroyWindow() upon your last open window, Windows 2000 will fire off the WM_ACTIVATEAPP message to some other app).

Here then is a slightly modified window procedure with these fixes:

```
/* ************************** mainWndProc() ****************************
 * Main Window message handler called by Windows
 ******************************************************************
 */

LONG APIENTRY mainWndProc(HWND hwnd, UINT uMsg, UINT wParam, LONG lParam)
{
    switch(uMsg)
    {
```

```c
    /* ============= App is gaining or losing activation ============= */
    case WM_ACTIVATEAPP:
    {
        /* Is app losing activation? */
        if (!wParam)
        {
            /* Close the MIDI devices (so other apps can use them) */
            closeMidiIn();
            closeMidiOut();
        }

        /* App gaining activation */
        else
        {
            /* Post a message to myself to reopen MIDI devices
               later, if they're not already open */
            PostMessage(hwnd, WM_USER, (WPARAM)0, (LPARAM)0);
        }

        /* Allow windows to continue handling WM_ACTIVATEAPP */
        break;
    }

    /* ================= MIDI Port allocation ================== */
    /* I send a WM_USER message to myself during WM_ACTIVATEAPP
       to open the MIDI In and Out devices. */
    case WM_USER:
    {
        /* Open MIDI In */
        openMidiIn();

        /* Open MIDI Out */
        openMidiOut();

        /* Handled this message */
        return(1);
    }

    /* ============= App is about to close its last window ============= */
    case WM_CLOSE:
    {
        /* Close the MIDI devices (so other apps can use them) */
        closeMidiIn();
        closeMidiOut();

        /* NOW close your last window */
        DestroyWindow(hwnd);

        /* Handled this message */
        return(1);
    }
}

// Let windows handle it
return(0);
```

```
}
```

Ok, you say, "But what about those times when I want to continue doing something with MIDI in the background while another program gets the focus?" For example, maybe you want to continue playing a MIDI file in the background, and don't want to stop just because the user switches to a paint program.

In that case, you should give the user the option of whether he wants your program to continue doing MIDI in the background. One good approach is to have some menu item he can check or uncheck which signals whether he wants you to "relinquish" the MIDI devices when your application is deactivated. Here I have a global, NoRelinquishFlag. When the user tells me to "hold onto the device", I set this to one. If the user wants me to relinquish the MIDI devices when he moves to another app, then I clear it. (I won't bother showing you how to set and clear variables). Below, I show you the simple modification you need to make to implement intelligent MIDI sharing that should encompass your software's requirements.

```c
unsigned char NoRelinquishFlag = 0;

/* ************************* mainWndProc() *****************************
 * Main Window message handler called by Windows
 ********************************************************************
 */

LONG APIENTRY mainWndProc(HWND hwnd, UINT uMsg, UINT wParam, LONG lParam)
{
    switch(uMsg)
    {
        /* ============= App is gaining or losing activation ============= */
        case WM_ACTIVATEAPP:
        {
            /* Is app losing activation? */
            if (!wParam)
            {
                /* Does user want the feature where this app relinquishes its
                   access to MIDI In and Out when user switches to another app? */
                if (!NoRelinquishFlag)
                {
                    /* Close the MIDI devices (so other apps can use them) */
                    closeMidiIn();
                    closeMidiOut();
                }
            }

            /* App gaining activation */
            else
            {
                /* Post a message to myself to reopen MIDI devices
                   later, if they're not already open */
                PostMessage(hwnd, WM_USER, (WPARAM)0, (LPARAM)0);
            }

            /* Allow windows to continue handling WM_ACTIVATEAPP */
            break;
        }

        /* ================= MIDI Port allocation ================= */
        /* I send a WM_USER message to myself during WM_ACTIVATEAPP
           to open the MIDI In and Out devices. */
        case WM_USER:
```

```
    {
        /* Open MIDI In */
        openMidiIn();

        /* Open MIDI Out */
        openMidiOut();

        /* Handled this message */
        return(1);
    }
}

    // Let windows handle it
    return(0);
}
```

Crashes under Windows 2000/NT/XP, and lost handles

When a program crashes, and it has no exception handling of its own, the Win32 default exception handling is invoked. This default handling attempts to free up resources for the program. For example, if you have some file open, the handle to it is closed. Under Windows 2000/NT/XP, the default exception handling unfortunately does not close any MIDI handles you have open. The net result is that, after a program crashes, no program will be able to open that MIDI port until you reboot the system.

But there is a way to solve this problem. What the default exception handling does do is to unload any Dynamic Link Libraries (DLL) that you have open. The operating system calls a special function (usually named DllMain()) in the DLL, passing a value of DLL_PROCESS_DETACH. The DLL is expected to do any cleanup then. So, if you place your MIDI handle variables inside of a DLL that you write, and then put code in your DllMain() function to close these handles if they are not zero, then your program will effectively close it MIDI handles, even when it crashes under Windows 2000/NT/XP.

So, how do you get access to those MIDI handles if they're inside of a DLL? Well, if you statically link with the DLL, and you've put the names of those variables in your Module Definition (.DEF) file, then Win32 automatically resolves those links when your program is loaded. You simply access those variables just like any other global variable in your program. But an even better idea is to put those above "wrapper functions" I wrote into a DLL, export their names in your DEF file, and call them just like any other functions in your program. Then, your DllMain()'s DLL_PROCESS_DETACH can simply call closeMidiOut() and closeMidiIn() to do the needed cleanup. You can download my MIDI Skeleton C example which implements this sharing. Run multiple copies of this program with a single client driver and you'll note that they all share a MIDI port. In fact, this example shows skeleton code for writing a MIDI C application, and may be of some use as a starting point for your own projects. Included are the Project Workspace files for Visual C++ 4.0. Remember that all apps should include MMSYSTEM.H and link with WINMM.LIB (or MMSYSTEM.LIB if Win3.1). This is a ZIP archive. Use an unzip utility that supports long filenames. It also includes the source code to a DLL with the wrapper functions in it. The C app uses this DLL to deal with the situation of freeing handles when crashing under Windows 2000/NT/XP.

Conclusion

By implementing the above, you'll be able to write a program that shares the MIDI ports with other such well-behaved MIDI programs (including multiple instances of your own program). It will work with single client drivers, and it will be transparent to the enduser. It will free MIDI handles under Windows 2000/NT/XP if your program crashes.

In conclusion, if you don't care about the overhead of opening and closing a device before and after every operation, and you do quick, MIDI operations, then you can just do the technique described in my first example. You won't hog a port that way.

But if you must hog a port, then do it the intelligent, user-friendly way as shown above.

Do Windows-using musicians a favor -- If you run across a MIDI program that doesn't share the MIDI port when you go to activate other MIDI programs, send the author of that first program a copy of this web page.

Using the Low level MIDI API, you need to first call midiOutOpen() or midiInOpen() to open some MIDI device for output or input respectively.

In order to write out MIDI data to a particular device, you need to first call midiOutOpen() once, passing it the Device ID of that desired device. Then, you can subsequently call a function such as midiOutShortMsg() which (immediately) outputs MIDI data to that device.

In order to read incoming MIDI data from a particular device, you need to first call midiInOpen() once, passing it the Device ID of that desired device. Then, Windows will subsequently pass your program each incoming MIDI message from that device.

After you're done inputting or outputting to a device (and have no further use for it), you must close that device.

Think of a MIDI device like a file. You open it, you read or write to it, and then you close it.

Opening the default MIDI device for output

How does your program choose a MIDI device for output? There are several different approaches you can take, depending upon how fancy and flexible you want your program to be.

Recall that Windows maintains separate lists of the devices which are capable of inputting MIDI data, and the devices capable of outputting MIDI data. For output, there is also the MIDI Mapper. It's not really a device in the MIDI Output list. (Ie, If you enumerate the devices in that list, you won't come across the MIDI Mapper). But it is there nonetheless. So what MIDI Output is the MIDI Mapper attached to? Well, that depends upon the settings that the user has made in Control Panel's Multimedia MIDI page. This Control Panel utility lets him route default MCI MIDI Output to a single MIDI Output (ie, one of the real MIDI Outputs in Windows' list). Or, he can use the "Custom configuration" setup (not available in Windows 2000/NT/XP) to split up the 16 MIDI channels among several of the real MIDI Outputs, for example, he could set all MIDI events on channel 1 to go to the built-in wavetable module on his Creative Labs sound card, and all MIDI events on channel 2 to go to the built-in wavetable on his Turtle Beach sound card. So when using the MIDI Mapper, although your program outputs to only one "device", it actually supports having the various MIDI channels going to different devices (which the user may desire for more polyphony or because some cards are better suited for certain sounds, etc). Plus, the "Add new Instrument" feature (not available in Windows earlier than Windows 95) allows the user to apply Instrument Definition Files thus remapping your program's MIDI output even more, for example, to make non-General MIDI instruments conform to General MIDI.

When you use the MIDI Mapper for output, think of the Multimedia utility's "MIDI" page as becoming the "MIDI Setup" dialog for your own application. Whichever way the user set that page up is where your calls to midiOutShortMsg() and midiOutLongMsg() get routed. So, by opening the MIDI Mapper, you use the "default MIDI Out" setup.

The MIDI Mapper has a defined Device ID of -1, so to open MIDI Mapper for MIDI Output:

```
unsigned long result;
HMIDIOUT        outHandle;

/* Open the MIDI Mapper */
result = midiOutOpen(&outHandle, (UINT)-1, 0, 0, CALLBACK_WINDOW);
if (result)
{
    printf("There was an error opening MIDI Mapper!\r\n");
```

```
}
```

One drawback with MIDI Mapper is that it does impose an extra layer of software processing upon your MIDI output. If the user never enables the "Custom configuration", then all MIDI data ends up going to one device anyway, so you gain nothing here (and lose a little efficiency).

Opening the default MIDI device for input

OK, that works for MIDI output, but what about MIDI input? There is no MIDI Mapper for input. You have to open one of the real devices. Remember that the first device in each list has a Device ID of 0. If he has at least one device capable of MIDI input, then you at least have a device with ID #0. So, you can simply use a Device ID of 0 with midiInOpen() as so:

```
unsigned long result;
HMIDIIN        inHandle;

/* Open the MIDI In device #0. Note: myWindow is a handle to some open window */
result = midiInOpen(&inHandle, 0, (DWORD)myWindow, 0, CALLBACK_WINDOW);
if (result)
{
    printf("There was an error opening the default MIDI In device!\r\n");
}
```

Of course, if the user has no device installed capable of inputting MIDI data, the above call returns an error, so always check that return value.

Note that the device(s) opened for output via MIDI Mapper, and the device opened for input above, may or may not be components of the same card. In other words, whichever MIDI IN jack is the default MIDI Input, and whichever MIDI OUT jack is the default MIDI Output, could be on two entirely different cards. But that is irrelevant to your purposes).

The MIDI Input device with an ID of 0 is whichever MIDI device happened to first get into the list of MIDI Input devices upon bootup. The user really has no control over setting this.

The most flexible way to choose a MIDI device for input or output

The most flexible way would be to present the user with all of the names in the list of MIDI Output devices and let him choose which one he wants (or if your program supports multiple MIDI output devices, you may wish to let him pick out several names from the list, and assign each sequencer "track" to one of those Device IDs. This is how professional sequencers implement support for multiple cards/outputs).

Whereas Windows maintains separate lists of MIDI Input and Output devices, so too, Windows has separate functions for querying the devices in each list.

Windows has a function that you can call to determine how many device names are in the list of devices that support outputting or playing MIDI data. This function is called midiOutGetNumDevs(). This returns the number of devices in the list. Remember that the Device IDs start with 0 and increment. So if Windows says that there are 3 devices in the list, then you know that their Device IDs are 0, 1, and 2 respectively. You then use these Device IDs with other Windows functions. For example, there is a function you can call to get information about one of the devices in the list, for example its name, and what sort of other features it has. You pass the Device ID of the device which you want to get information about (as well as a pointer to a special structure called a MIDIOUTCAPS into which Windows puts

the info about the device), The name of the function to get information about a particular MIDI Output device is midiOutGetDevCaps().

Here then is an example of going through the list of MIDI Output devices, and printing the name of each one:

```
MIDIOUTCAPS      moc;
unsigned long    iNumDevs, i;

/* Get the number of MIDI Out devices in this computer */
iNumDevs = midiOutGetNumDevs();

/* Go through all of those devices, displaying their names */
for (i = 0; i < iNumDevs; i++)
{
    /* Get info about the next device */
    if (!midiOutGetDevCaps(i, &moc, sizeof(MIDIOUTCAPS)))
    {
        /* Display its Device ID and name */
        printf("Device ID #%u: %s\r\n", i, moc.szPname);
    }
}
```

Likewise with MIDI Input devices, Windows has a function that you can call to determine how many device names are in the list of devices that support inputting or creating MIDI data. This function is called midiInGetNumDevs(). This returns the number of devices in the list. Again, the Device IDs start with 0 and increment. There is a function you can call to get information about one of the devices in the list, for example its name, and what sort of other features it has. You pass the Device ID of the device which you want to get information about (as well as a pointer to a special structure called a MIDIINCAPS into which Windows puts the info about the device), The name of the function to get information about a particular MIDI Input device is midiInGetDevCaps().

Here then is an example of going through the list of MIDI Input devices, and printing the name of each one:

```
MIDIINCAPS       mic;
unsigned long    iNumDevs, i;

/* Get the number of MIDI In devices in this computer */
iNumDevs = midiInGetNumDevs();

/* Go through all of those devices, displaying their names */
for (i = 0; i < iNumDevs; i++)
{
    /* Get info about the next device */
    if (!midiInGetDevCaps(i, &mic, sizeof(MIDIINCAPS)))
    {
        /* Display its Device ID and name */
        printf("Device ID #%u: %s\r\n", i, mic.szPname);
    }
}
```

You can download my ListMidiDevs C example to show how to print the names of all the installed MIDI Input and Output devices, as well as other info about each device. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it. Remember that all apps should include MMSYSTEM.H and link with WINMM.LIB (or MMSYSTEM.LIB if Win3.1). This is a ZIP archive. Use an

unzip utility that supports long filenames.

Outputting MIDI data (except System Exclusive)

How does an application tell Windows to output some MIDI bytes? That depends upon whether you're outputting System Exclusive Messages, or some other kind of MIDI message. All MIDI messages, except for System Exclusive, always have 3 or less bytes. So, Windows has a function through which you can pass such a MIDI message in its entirety for output. This function is called **midiOutShortMsg()**. What you do is pack up the 3 or less bytes of that MIDI message as an unsigned long value, and pass it to midiOutShortMsg(). The bytes of this MIDI message then get output as soon as possible (ie, hopefully immediately).

With midiOutShortMsg(), you need to pack up these 3 bytes into one unsigned long which is passed as one arg. The LSB of the low word is the MIDI status (for example, 0x90 for MIDI channel 1). The MSB of the low word is the first MIDI data byte, if any. (For Note events, this would be the MIDI note number). The LSB of the high word is the second MIDI data byte, if any. (For Note events, this would be the note velocity). The MSB of the high word is not used.

Note: Always include a status byte. The device driver for the card will implement running status when it outputs the MIDI message.

Let's take an example of playing a 3 note chord -- a C chord (ie, C, E, and G notes).

Each musical pitch of a chord is expressed as a MIDI note number (middle C is note number 60, so D# above middle C is #61, etc). We'll create a MIDI message for each one of those 3 note numbers. A MIDI message takes the form of 3 bytes; the Status byte, the note number, and velocity (usually implements note volume). The Status byte for turning a note on is 0x9**X** where X is the MIDI channel number desired (0 to F for MIDI channels 1 to 16 -- we'll use a default of 0 but you may want to allow the user to change this). So for MIDI channel 1, the status is always 0x90. For velocity, we'll use a default of 0x40.

```
HMIDIOUT      handle;

/* Open default MIDI Out device */
if (!midiOutOpen(&handle, (UINT)-1, 0, 0, CALLBACK_NULL) )
{
    /* Output the C note (ie, sound the note) */
    midiOutShortMsg(handle, 0x00403C90);

    /* Output the E note */
    midiOutShortMsg(handle, 0x00404090);

    /* Output the G note */
    midiOutShortMsg(handle, 0x00404390);

    /* Here you should insert a delay so that you can hear the notes sounding */
    Sleep(1000);

    /* Now let's turn off those 3 notes */
    midiOutShortMsg(handle, 0x00003C90);
    midiOutShortMsg(handle, 0x00004090);
    midiOutShortMsg(handle, 0x00004390);
```

```
    /* Close the MIDI device */
    midiOutClose(handle);
}
```

Note: If your application is doing some sort of sequencing (ie, playback of a musical piece), you'll have to maintain a timer in order to figure out when it's time to output the next MIDI message via midiOutShortMsg(). (Note that 32-bit Windows MultiMedia Timer callbacks under Win95 may suffer severe timing fluctuations. Since Win95's multimedia system is still 16-bit, you need to put your timer callback (and any functions it calls) into a 16-bit DLL in order for it to exhibit solid performance under Win95. WinNT doesn't exhibit this aberrant behavior with 32-bit code).

Warning: There are some badly written drivers out there, especially for Windows NT/2000/XP. midiOutShortMsg() doesn't actually output the MIDI message. Instead, the driver puts the message in some internal queue for another thread (running inside of the driver) to output later, and then midiOutShortMsg() returns immediately. There's nothing wrong with that per se, but in these badly written drivers, when you call midiOutClose(), the driver discards any MIDI messages in its queue (as opposed to properly flushing them to MIDI Out). So if you call midiOutClose() too soon after calling midiOutShortMsg() (or perhaps even midiOutLongMsg()), then your MIDI output may end up being discarded before that other thread inside of the driver gets a chance to output the message. If the above example code results in stuck notes (or no sound at all) unless you put another Sleep() before the call to midiOutClose(), then you're dealing with such a badly designed driver. You could try to write some workaround that, whenever to call midiOutShort(), set a flag variable and start a one-shot timer. At the end of the timer, clear the flag. Before you ever call midiOutClose(), make sure that this flag is clear. If not, postpone the close. But a better solution is to contact the author of your card's driver, and tell him to fix his badly designed code.

You can download my Twinkle C example to show how to use midiOutShortMsg to play MIDI notes on the default MIDI Out device. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it.

Outputting System Exclusive MIDI messages

Since System Exclusive messages can be any length, Windows has a different means for outputting them. You use the function midiOutLongMsg(), passing it a buffer filled with the MIDI message. (In fact, you can use midiOutLongMsg to pass a buffer filled with several non-System Exclusive messages. This is very handy if you need to output several MIDI messages that should occur simultaneously. Using midiOutShortMsg() for each individual MIDI message as we did in the example above may cause too much of a delay inbetween each MIDI message). You actually pass a special structure called a MIDIHDR which has a field where you store the pointer to your buffer containing the MIDI data.

But there are a few caveats:

1. If running Win3.1, the data buffer (and perhaps the MIDIHDR structure, although MS examples show otherwise) must be allocated with GlobalAlloc() using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with GlobalLock. Under Win95 and WinNT, this no longer appears to be a requirement.

2. Before you pass the buffer to midiOutLongMsg(), you must first "prepare" it by calling midiOutPrepareHeader().

3. The MIDI Output device's driver determines whether the data is sent synchronously or asynchronously. So, with some devices, your app won't return from the call to midiOutLongMsg() until all of the data is output, whereas with other devices, you may return immediately and the driver will continue outputting the data in the background.

4. After you're done with the buffer, you must "unprepare" it by calling midiOutUnprepareHeader().

Here's an example of outputting a System exclusive message under Win3.1:

```
HMIDIOUT      handle;
MIDIHDR       midiHdr;
HANDLE        hBuffer;
UINT          err;
char          sysEx[] = {0xF0, 0x7F, 0x7F, 0x04, 0x01, 0x7F, 0x7F, 0xF7};

/* Open default MIDI Out device */
if (!midiOutOpen(&handle, (UINT)-1, 0, 0, CALLBACK_NULL))
{
    /* Allocate a buffer for the System Exclusive data */
    hBuffer = GlobalAlloc(GHND, sizeof(sysEx));
    if (hBuffer)
    {
        /* Lock that buffer and store pointer in MIDIHDR */
        midiHdr.lpData = (LPBYTE)GlobalLock(hBuffer);
        if (midiHdr.lpData)
        {
            /* Store its size in the MIDIHDR */
            midiHdr.dwBufferLength = sizeof(sysEx);

            /* Flags must be set to 0 */
            midiHdr.dwFlags = 0;

            /* Prepare the buffer and MIDIHDR */
            err = midiOutPrepareHeader(handle,  &midiHdr, sizeof(MIDIHDR));
            if (!err)
            {
                /* Copy the SysEx message to the buffer */
                memcpy(midiHdr.lpData, &sysEx[0], sizeof(sysEx));

                /* Output the SysEx message */
                err = midiOutLongMsg(handle, &midiHdr, sizeof(MIDIHDR));
                if (err)
                {
                    char    errMsg[120];

                    midiOutGetErrorText(err, &errMsg[0], 120);
                    printf("Error: %s\r\n", &errMsg[0]);
                }

                /* Unprepare the buffer and MIDIHDR */
                while (MIDIERR_STILLPLAYING == midiOutUnprepareHeader(handle,
&midiHdr, sizeof(MIDIHDR)))
                {
                    /* Should put a delay in here rather than a busy-wait */
                }
            }

            /* Unlock the buffer */
            GlobalUnlock(hBuffer);
        }
```

```
        /* Free the buffer */
        GlobalFree(hBuffer);
    }

    /* Close the MIDI device */
    midiOutClose(handle);
}
```

Win95 and WinNT are easier. Here's an example to output a System Exclusive message under Win95/NT:

```
HMIDIOUT        handle;
MIDIHDR         midiHdr;
UINT            err;
char               sysEx[] = {0xF0, 0x7F, 0x7F, 0x04, 0x01, 0x7F, 0x7F, 0xF7};

/* Open default MIDI Out device */
if (!midiOutOpen(&handle, (UINT)-1, 0, 0, CALLBACK_NULL))
{
    /* Store pointer in MIDIHDR */
    midiHdr.lpData = (LPBYTE)&sysEx[0];

    /* Store its size in the MIDIHDR */
    midiHdr.dwBufferLength = sizeof(sysEx);

    /* Flags must be set to 0 */
    midiHdr.dwFlags = 0;

    /* Prepare the buffer and MIDIHDR */
    err = midiOutPrepareHeader(handle,  &midiHdr, sizeof(MIDIHDR));
    if (!err)
    {
        /* Output the SysEx message */
        err = midiOutLongMsg(handle, &midiHdr, sizeof(MIDIHDR));
        if (err)
        {
            char    errMsg[120];

            midiOutGetErrorText(err, &errMsg[0], 120);
            printf("Error: %s\r\n", &errMsg[0]);
        }

        /* Unprepare the buffer and MIDIHDR */
        while (MIDIERR_STILLPLAYING == midiOutUnprepareHeader(handle, &midiHdr,
sizeof(MIDIHDR)))
        {
            /* Should put a delay in here rather than a busy-wait
*/
        }
    }

    /* Close the MIDI device */
    midiOutClose(handle);
}
```

You can download my MidiVol C example to show how to use midiOutLongMsg to output a MIDI System Exclusive message on the default MIDI Out device. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it.

Inputting MIDI data

In the examples of outputting MIDI data, you'll notice that in the call to midiOutOpen(), I specified CALLBACK_NULL. What this flag means is that (other than the return codes from functions such as midiOutShortMsg or midiOutLongMsg), I require no feedback from Windows on the progress of the output of that MIDI data. (Remember that these output functions may return to the app before the data is finished being sent, if the driver has some means of doing that output in the background). I could have asked Windows to provide me with feedback (as we'll do in our MIDI input examples), but it wasn't necessary in those output examples, and it's simpler not to do it (although you should do it if you've called midiOutLongMsg -- in order to know when it's time to call midiOutUnprepareHeader. Otherwise, you'd have to do polling on the MHDR_DONE bit of the dwFlags field of the MIDIHDR structure. This bit will be set when the MIDI driver is finished with the MIDIHDR).

With MIDI input, you must provide a means of Windows giving you some sort of feedback. Why? Because you can't keep continuously polling the MIDI In port of a MIDI Input device waiting for incoming MIDI messages. That's an outdated MS-DOS programming technique. Rather, Windows programs are supposed to relinquish control back to Windows when the program has nothing to do except wait for something to happen, (Although it's possible, albeit not good programming practice, to do polling of the MHDR_DONE bit when inputting System Exclusive messages, for input of other types of MIDI messages, Windows requires that you provide a different means for Windows to pass you MIDI data).

Instead, Windows will interact with your program whenever each MIDI message is input (ie, all of the bytes in it, for non-System Exclusive messages) or some input buffer you've supplied is filled (for System Exclusive messages). How does Windows interact with your program? You have several options as follows:

1. CALLBACK_EVENT -- You allocate some Event with CreateEvent(), and Windows uses this to signal your app. (ie, Your app can wait on that Event signal, for example with WaitForSingleObject). You pass the handle of the Event as the 3rd arg to midiInOpen().

2. CALLBACK_THREAD -- Windows causes some suspended thread within your app to run. (ie, Your app's thread can suspend itself via SuspendThread). You pass the Thread ID of the desired thread to be run as the 3rd arg to midiInOpen().

3. CALLBACK_WINDOW -- Windows sends a message to some open window in your app. The parameters for the message will contain additional information about what caused Windows to send that message. You pass the desired window's handle as the 3rd arg to midiInOpen().

4. CALLBACK_FUNCTION -- Windows directly calls some function in your app. It passes args that contain additional information about what caused Windows to call your function. You pass a pointer to the desired function as the 3rd arg to midiInOpen(). The 4th arg to midiInOpen() can be anything you desire, and this will be passed to your callback function each time that Windows calls your callback.

The latter two methods allow you to better determine what exactly caused Windows to notify you, because they supply additional information to you. In fact, for regular MIDI messages (ie, everything except System Exclusive messages -- I'll simply refer to these as "regular messages"), the latter two methods are the only methods you can use to actually get Windows to pass you the incoming MIDI data. (For System Exclusive, you could use the CALLBACK_EVENT or

CALLBACK_THREAD, if you're not really interested in being notified of errors). For this reason, I'll only detail the latter two methods.

So when does Windows notify you? Here are the times when Windows notifies you:

1. When you open a MIDI In device via midiInOpen().

2. When you close a MIDI In Device via midiInClose().

3. When Windows has finished inputting one regular MIDI message in its entirety.

4. When Windows has filled a MIDIHDR's buffer with a portion, or all, of a System Exclusive message.

5. When there has been an error inputting a regular or System Exclusive MIDI message.

6. When your handling of a particular MIDI message (that you've been passed) is so slow that the MIDI driver (and possibly the MIDI Interface) has had to throw away incoming MIDI data while it was waiting for you to finish processing a previous message.

In conclusion, you can have Windows call a function you have written, passing the MIDI data that has been input, or you can have Windows pass a message to one of your program's windows, with the MIDI data that has been input as part of that message.

To have Windows call a function you have written, when you open the device, you specify the flag CALLBACK_FUNCTION. The third arg is a pointer to your function. (The fourth arg can be any value that you want Windows to pass to your function each time that function is called).

```
result = midiInOpen(&inHandle, 0, (DWORD)myFunc, 0, CALLBACK_FUNCTION);
```

To have Windows pass a message to one of your windows, when you open the device, you specify the flag CALLBACK_WINDOW. The third arg is a handle to your window. (The fourth arg is not used).

```
result = midiInOpen(&inHandle, 0, (DWORD)myWindow, 0, CALLBACK_WINDOW);
```

One caveat with this second method is that Windows doesn't timestamp each MIDI message. So if you need timestamps, you would have to timestamp each MIDI message yourself using some software timer (ie, perhaps the one that you're using to time the output of MIDI messages, for example, a Windows MultiMedia timer implemented using functions such as timeGetTime). But such message passing is not very efficient. By the time that your window procedure finally got around to pulling that MIDI data out of a message and obtaining a time stamp for it, a long come could have transpired since it actually arrived at the computer's MIDI IN. Trying to get an accurate time stamp using this method is very difficult, especially if other things are happening in the system such as mouse and window movement. It is recommended that you use CALLBACK_WINDOW only when you don't need time stamps. Otherwise, use the CALLBACK_FUNCTION method.

If using the CALLBACK_FUNCTION method, then you need to write a function that has the following declaration (although you can name the function anything you like):

```
void CALLBACK midiCallback(HMIDIIN handle, UINT uMsg, DWORD dwInstance, DWORD dwParam1, DWORD dwParam2);
```

As mentioned, you pass a pointer to this function as the 3rd arg to midiInOpen(). The 4th arg to midiInOpen() can be

anything you desire, and this will be passed to your callback function (as the dwInstance arg) each time that Windows calls your callback. The handle arg is what was returned from midiInOpen().

The other args may be interpreted differently depending upon why Windows has called your callback. Here are those reasons:

1. You open a MIDI In Device via midiInOpen(). In this case, the uMsg arg to your callback will be MIM_OPEN.

2. You close a MIDI In Device via midiInClose(). In this case, the uMsg arg to your callback will be MIM_CLOSE.

3. One, regular (ie, everything except System Exclusive messages) MIDI message has been completely input. In this case, the uMsg arg to your callback will be MIM_DATA. The dwParam1 arg is the bytes of the MIDI Message packed into an unsigned long in the same format that is used by midiOutShort(). The dwParam2 arg is a time stamp that the device driver created when it recorded the MIDI message.

4. Windows has either completely filled a MIDIHDR's memory buffer with part of a System Exclusive message (in which case you had better continue queuing the MIDIHDR again in order to grab the remainder of the System Exclusive), or the MIDIHDR's memory buffer contains the remainder of a System Exclusive message (or the whole message if it happened to fit into the memory buffer intact). In this case, the uMsg arg to your callback will be MIM_LONGDATA. The dwParam1 arg is a pointer to the MIDIHDR whose memory buffer contains the System Exclusive data. The dwParam2 arg is a time stamp that the device driver created when it recorded the MIDI message.

5. Your callback is not processing data fast enough such that the MIDI driver (and possibly the MIDI In port itself) has had to throw away some incoming, regular MIDI messages. In this case, the uMsg arg to your callback will be MIM_MOREDATA. The dwParam1 arg is the bytes of the MIDI Message that was not handled (by an MIM_DATA call) packed into an unsigned long in the same format that is used by midiOutShort(). The dwParam2 arg is a time stamp that the device driver created when it recorded the MIDI message. In handling a series of these events, you should store the MIDI data in a global buffer, until such time as you receive another MIM_DATA (which indicates that you can now do the more time-consuming processing that you obviously were doing in handling MIM_DATA). In other words, when Windows calls your callback with MIM_MOREDATA, this is it's way of saying "You're handling your previous MIM_DATA messages too slowly. (And in fact, I may have preemptively interrupted a previous MIM_DATA handling of your callback). This is your last chance to quickly do something with this one message that I'm passing you now. Otherwise, you're so far behind in handling the MIDI input that data is about to be permanently lost".

   NOTE: Windows sends an MIM_MOREDATA event only if you specify the MIDI_IO_STATUS flag to midiInOpen().

6. An invalid, regular MIDI message was received. In this case, the uMsg arg to your callback will be MIM_ERROR. The dwParam1 arg is the bytes of the MIDI Message that was not handled (by an MIM_DATA call) packed into an unsigned long in the same format that is used by midiOutShort(). The dwParam2 arg is a time stamp that the device driver created when it recorded the MIDI message.

7. An invalid, System Exclusive message was received. In this case, the uMsg arg to your callback will be MIM_LONGERROR. The dwParam1 arg is a pointer to the MIDIHDR whose memory buffer contains the System Exclusive data. The dwParam2 arg is a time stamp that the device driver created when it recorded the MIDI message.

MIDI time stamps are defined as the time the first byte of the message was received and are specified in milliseconds. The midiInStart() function resets the time stamps for a device to 0.

You can download my [Midi In Callback](#) C example to show how to input MIDI messages. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it.

Using the Low level Digital Audio API, you need to first call waveOutOpen() or waveInOpen() to open some Digital Audio device for output (use its Digital to Analog Converter to play audio) or input (use its Analog to Digital Converter to record audio) respectively.

In order to write out Digital Audio data to a particular device's DAC, you need to first call waveOutOpen() once, passing it the Device ID of that desired device. Then, you can subsequently call a function to send blocks of Digital Audio data to that device's DAC. One of the other parameters you pass is a pointer to a WAVEFORMATEX structure. You fill in the fields of this structure (prior to calling waveOutOpen) to tell the device such things as the sample rate and bit resolution of the digital audio data you intend to play, as well as whether it is Mono (1 channel) or stereo (2 channels).

In order to read incoming Digital Audio data from a particular device's ADC, you need to first call waveInOpen() once, passing it the Device ID of that desired device. Then, Windows will subsequently pass your program blocks of incoming Digital Audio from that device's ADC. One of the other parameters you pass is a pointer to a WAVEFORMATEX structure. You fill in the fields of this structure (prior to calling waveInOpen) to tell the device such things what sample rate and bit resolution to use when recording the digital audio data, as well as whether to record in Mono (1 channel) or stereo (2 channels).

After you're done recording or playing Digital Audio on a device (and have no further use for it), you must close that device.

Think of a Digital Audio device like a file. You open it, you read or write to it, and then you close it.

Easy way to choose a Digital Audio device for input or output

How does your program choose a Digital Audio device for input or output? There are several different approaches you can take, depending upon how fancy and flexible you want your program to be.

Recall that Windows maintains separate lists of the devices which are capable of recording Digital Audio data, and the devices capable of playing Digital Audio data. Pass the value WAVE_MAPPER as the Device ID to open the "preferred" Digital Audio Input device and Digital Audio Output device respectively. So, if you simply want to open the preferred Digital Audio Output device, then use a Device ID of WAVE_MAPPER with waveOutOpen() as so:

```
unsigned long result;
HWAVEOUT      outHandle;
WAVEFORMATEX  waveFormat;

/* Initialize the WAVEFORMATEX for 16-bit, 44KHz, stereo */
waveFormat.wFormatTag = WAVE_FORMAT_PCM;
waveFormat.nChannels = 2;
waveFormat.nSamplesPerSec = 44100;
waveFormat.wBitsPerSample = 16;
waveFormat.nBlockAlign = waveFormat.nChannels * (waveFormat.wBitsPerSample/8);
waveFormat.nAvgBytesPerSec = waveFormat.nSamplesPerSec * waveFormat.nBlockAlign;
waveFormat.cbSize = 0;

/* Open the preferred Digital Audio Out device. Note: myWindow is a handle to some
open window */
result = waveOutOpen(&outHandle, WAVE_MAPPER, &waveFormat, (DWORD)myWindow, 0,
CALLBACK_WINDOW);
if (result)
{
    printf("There was an error opening the preferred Digital Audio Out device!\r\n");
}
```

Note: If the preferred device does not support your desired choice of sample rate and channels, then Windows will instead open some other device that does (assuming that there is such other device available).

Of course, if the user has no device installed capable of outputting or playing Digital Audio data, the above call returns an error, so always check that return value.

Likewise, use a Device ID of WAVE_MAPPER with waveInOpen() to open the preferred Digital Audio Input device. (Note that these two preferred devices may or may not be components of the same card. But that is irrelevant to your purposes. The only caveat is that if they are components upon the same card, the card's driver needs to be full duplex in order to simultaneously open both the Digital Audio input and output. In this way, your program can play back previously recorded waveforms while recording new waveforms. Without a full duplex driver, you have to open for recording, record a waveform, close the device using waveInClose(), and open for playback. Some sound card designs do not allow the card to simultaneously record and play digital audio, so they have only half duplex drivers).

```c
unsigned long result;
HWAVEIN       inHandle;
WAVEFORMATEX  waveFormat;

/* Initialize the WAVEFORMATEX for 16-bit, 44KHz, stereo */
waveFormat.wFormatTag = WAVE_FORMAT_PCM;
waveFormat.nChannels = 2;
waveFormat.nSamplesPerSec = 44100;
waveFormat.wBitsPerSample = 16;
waveFormat.nBlockAlign = waveFormat.nChannels * (waveFormat.wBitsPerSample/8);
waveFormat.nAvgBytesPerSec = waveFormat.nSamplesPerSec * waveFormat.nBlockAlign;
waveFormat.cbSize = 0;

/* Open the preferred Digital Audio In device */
result = waveInOpen(&inHandle, WAVE_MAPPER, &waveFormat, (DWORD)myWindow, 0,
CALLBACK_WINDOW);
if (result)
{
    printf("There was an error opening the preferred Digital Audio In device!\r\n");
}
```

So what actually is the preferred Digital Audio Output device? Well, that's whatever device that the user choose from the dropdown list of Digital Audio Output devices under "Playback" of Control Panel's *Multimedia* utility (ie, on the "Audio" page). The list on this page is Windows actually displaying all of the names that were added to its list of devices capable of outputting or playing Digital Audio data.

The preferred Digital Audio Input device is whatever device that the user chooses from the dropdown list of Digital Audio Input devices under "Recording" of Control Panel's *Multimedia* utility (ie, on the "Audio" page). The list on this page is Windows actually displaying all of the names that were added to its list of devices capable of inputting or recording Digital Audio data.

The most flexible way to choose a Digital Audio device

The most flexible way would be to present the user with all of the names in the list of Digital Audio Output devices and let him choose which ones he wants (or if your program supports multiple Digital Audio output devices, you may wish to let him pick out several names from the list, and assign each digital audio "track" to one of those Device IDs. This is how professional sequencers implement support for multiple cards/outputs, in addition to perhaps implementing virtual tracks).

Whereas Windows maintains separate lists of Digital Audio Input and Output devices, so too, Windows has separate functions for querying the devices in each list.

Windows has a function that you can call to determine how many device names are in the list of devices that support outputting or playing Digital Audio data. This function is called waveOutGetNumDevs(). This returns the number of devices

in the list. Remember that the Device IDs start with 0 and increment. So if Windows says that there are 3 devices in the list, then you know that their Device IDs are 0, 1, and 2 respectively. You then use these Device IDs with other Windows functions. For example, there is a function you can call to get information about one of the devices in the list, for example its name, and what sort of other features it has such as what sample rates it supports. You pass the Device ID of the device which you want to get information about (as well as a pointer to a special structure called a WAVEOUTCAPS into which Windows puts the info about the device), The name of the function to get information about a particular Digital Audio Output device is waveOutGetDevCaps().

Here then is an example of going through the list of Digital Audio Output devices, and printing the name of each one:

```
WAVEOUTCAPS       woc;
unsigned long     iNumDevs, i;

/* Get the number of Digital Audio Out devices in this computer */
iNumDevs = waveOutGetNumDevs();

/* Go through all of those devices, displaying their names */
for (i = 0; i < iNumDevs; i++)
{
    /* Get info about the next device */
    if (!waveOutGetDevCaps(i, &woc, sizeof(WAVEOUTCAPS)))
    {
        /* Display its Device ID and name */
        printf("Device ID #%u: %s\r\n", i, woc.szPname);
    }
}
```

Likewise with Digital Audio Input devices, Windows has a function that you can call to determine how many device names are in the list of devices that support inputting or recording Digital Audio data. This function is called waveInGetNumDevs(). This returns the number of devices in the list. Again, the Device IDs start with 0 and increment. There is a function you can call to get information about one of the devices in the list, for example its name, and what sort of other features it has such as what sample rates it supports. You pass the Device ID of the device which you want to get information about (as well as a pointer to a special structure called a WAVEINCAPS into which Windows puts the info about the device), The name of the function to get information about a particular Digital Audio Input device is waveInGetDevCaps().

Here then is an example of going through the list of Digital Audio Input devices, and printing the name of each one:

```
WAVEINCAPS        wic;
unsigned long     iNumDevs, i;

/* Get the number of Digital Audio In devices in this computer */
iNumDevs = waveInGetNumDevs();

/* Go through all of those devices, displaying their names */
for (i = 0; i < iNumDevs; i++)
{
    /* Get info about the next device */
    if (!waveInGetDevCaps(i, &wic, sizeof(WAVEINCAPS)))
    {
        /* Display its Device ID and name */
        printf("Device ID #%u: %s\r\n", i, wic.szPname);
    }
}
```

Recording Digital Audio

The device's driver manages the actual recording of data. You can start and stop this process with waveInStart() and waveInStop(). While a driver records digital audio, it stores data into a small fixed-size buffer (for example 16K). When that buffer is full, the driver "signals" your program that the buffer is full and needs to be processed by your program (for example, your program may save that 16K of data to a disk file if your program is doing hard disk recording). The driver then goes on to store another "block" (ie, 16K section) of data into a second, similiarly-sized buffer. It's assumed that your program is simultaneously processing that first buffer of data, while the driver is recording into the second buffer. It's also assumed that your program finishes processing that first buffer before the second buffer is full. When the driver fills that second buffer, it again signals your program that now the second buffer needs to be processed. While your program is processing the second buffer, the driver is storing more audio data into the now-empty, first buffer. Etc. This all happens nonstop, so the process of recording digital audio is that two (or more if desired) buffers are constantly being filled by the driver (alternating between the 2 buffers), while your program is constantly processing each buffer immediately upon being signaled that the buffer is full. So, you end up dealing with a series of "blocks of data".

In fact, your program supplies each buffer to the driver, using waveInAddBuffer() (and waveInPrepareHeader() to initialize it). You supply the first 2 buffers to the driver using waveInAddBuffer() before recording. Every time that you're signaled that a buffer is filled, you need to use waveInAddBuffer() to indicate what buffer the driver will use after it finishes filling whatever buffer it is currently filling. (For double-buffering, that will be the same buffer that you're currently processing).

You can download my WaveIn C example to show how to record a (raw) digital audio file using double-buffering. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it. Remember that all apps should include MMSYSTEM.H and link with WINMM.LIB (or MMSYSTEM.LIB if Win3.1). This is a ZIP archive. Use an unzip utility that supports long filenames.

Playing Digital Audio

Playback is also done via "blocks of data". Here, your application reads a block of data from the WAVE file on disk (for example, you may read the next 16K of the file into a 16K buffer). (You must use waveOutPrepareHeader() to initialize the buffer before reading into it). You pass this block to the driver for playback via waveOutWrite(). While the driver is playing this block, you're reading in another block of data into a second buffer. When the driver is finished playing the first block, it signals your program that it needs another block, and your driver passes that second buffer via waveOutWrite(). Your program will now read in the next block of data into the first buffer while the driver is playing the second buffer. Etc. Again, this is all non-stop until the WAVE is fully played (at which point you can call waveOutReset() to stop the driver's playback process).

So how does the driver "signal" your program? You've got a few choices. You can choose to have the driver send messages to your program's Window, for example, the MM_WOM_DONE message is sent each time the driver finishes playing a given buffer. Parameters with that message include the address of the given buffer (actually the address of the WAVEHDR structure which encompasses the buffer) and the device's handle (ie, the handle supplied to you when you opened the device). Or, you can have the driver automatically call a particular function in your program (ie, a "callback") passing such parameters. There are a couple of other choices such as having the driver use event signals or start a particular thread in your program.

You tell the driver how you want to be signaled by setting certain flags in one of the arguments to waveInOpen() or waveOutOpen().

You can download my WavePly1 C example to show how to play a WAVE file using a callback and double-buffering.

Setting volume and other parameters

There are other APIs that you'll likely want to use, such as waveOutSetVolume to set the volume for playback. Or, you may prefer to use the Mixer API (if the card's driver supports it) so that you can mute unneeded inputs/outputs, and adjust other parameters of a particular input/output line, or to determine what types of lines are available for recording (ie, analog microphone input, digital input, etc).

If you don't want the hassle of timing out the playback of MIDI data, nor loading/parsing MIDI files, you could use Windows High level MCI functions. These functions can use the MCI Sequencer Device to play an entire MIDI file on its own, in the background (ie, while your app does other things). But, you lose control over MIDI playback other than being able to stop, start, pause, rewind, and fast-forward the playback. There's Windows MCI functions for setting up, and controlling the MCI Sequencer Device, for example using mciSendCommand to send commands to the MCI Sequencer Device such that it opens a MIDI file on disk, reads it in, and plays it back.

Before proceeding, you should now read MCI Devices. This article gives necessary background information about the MCI API.

Opening the Sequencer Device

To open the Sequencer Device, you need to issue an "open" command, using either using mciSendString() or mciSendCommand() (depending upon whether you want to specify your open command by passing formatted strings, or binary values/structures, respectively), and specify that you want the Sequencer device as the type of device opened. Of course, you also need to supply the name of a MIDI file that you want the Sequencer to open and perform operations upon.

If you're using mciSendString(), you literally include "type sequencer" as part of the command string to indicate that you're opening the Sequencer device.

You can also specify an *alias*. This is just a string name that you use to identify the open device. Think of it as a string version of a device handle. After all, the string interface doesn't return binary values, so it can't return a handle. Instead it allows you to pick out a name, which you'll use with other commands you issue, in lieu of having a handle.

Here then is an example of opening the Sequencer device using the Command String interface. The MIDI file that we ask it to open is named C:\WINDOWS\SONG.MID. The alias we give this instance of the Sequencer is A_Song. (ie, Whenever we subsequently use A_Song as the device name with other commands, we'll be performing operations on the C:\WINDOWS \SONG.MID file).

```
TCHAR    buf[128];
DWORD    err;

/* Open a Sequencer device associated with the C:\WINDOWS\SONG.MID file */
if ((err = mciSendString("open C:\\WINDOWS\\SONG.MID type sequencer alias A_Song",
0, 0, 0)))
{
    /* Error */
    printf("Sequencer device did not open!\r\n");
    if (mciGetErrorString(err, &buf[0], sizeof(buf))) printf("%s\r\n", &buf[0]);
}
```

If you're using mciSendCommand(), then you need to initialize and pass a MCI_OPEN_PARMS structure. You set the lpstrDeviceType field of this structure to MCI_DEVTYPE_SEQUENCER. You also must pass mciSendCommand() the MCI_OPEN_TYPE and MCI_OPEN_TYPE_ID flags to indicate that you've set the lpstrDeviceType field to a predefined constant (MCI_DEVTYPE_SEQUENCER).

If playing back a MIDI file, then you also need to specify the name of the MIDI file to open by setting the lpstrElementName field to point to the name of the desired file. You also must pass mciSendCommand() the MCI_OPEN_ELEMENT flag to indicate that you've set the lpstrElementName field. The Sequencer device does not support recording MIDI.

The second arg will be MCI_OPEN to indicate an open command.

Here then is an example of opening the Sequencer device using the Command Message interface.

```
DWORD            err;
```

```c
MCI_OPEN_PARMS midiParams;
TCHAR          buffer[128];

/* Open a Sequencer device associated with the C:\WINDOWS\SONG.MID file */
midiParams.lpstrDeviceType = (LPCSTR)MCI_DEVTYPE_SEQUENCER;
midiParams.lpstrElementName = "C:\\WINDOWS\\SONG.MID";
if ((err = mciSendCommand(0, MCI_OPEN, MCI_WAIT|MCI_OPEN_ELEMENT|MCI_OPEN_TYPE|
MCI_OPEN_TYPE_ID, (DWORD)(LPVOID)&midiParams)))
{
    /* Error */
    printf("ERROR: Sequencer device did not open!\r\n");
    if (mciGetErrorString(err, &buffer[0], sizeof(buffer))) printf("%s\r\n", &buffer
[0]);
}
else
{
    /* The device opened successfully. midiParams.wDeviceID now contains the device
ID */
}
```

Playing the Sequencer Device

To have the Sequencer device play its open MIDI file, you issue a play command to it.

If you're using mciSendCommand(), then you need to initialize and pass a MCI_PLAY_PARMS structure. (Actually, unless you use the MCI_NOTIFY, MCI_FROM, and/or MCI_TO flags, there is no initialization required). If you want to play only part of the MIDI file, then you can set the dwFrom and/or dwTo fields to the start and end positions respectively, and pass the MCI_FROM and/or MCI_TO flags respectively. The dwFrom and dwTo fields are normally expressed in terms of PPQN clocks. But you can utilize other means of expressing this offset, for example in terms of milliseconds, by first issuing an MCI_SET command (with the MCI_SET_TIME_FORMAT flag, and your desired choice of how to express such offsets).

The second arg will be MCI_PLAY to indicate a play command.

The first arg will be the device ID that you obtained when you opened the device.

Here then is an example of playing a MIDI file on the Sequencer device using the Command Message interface.

```c
DWORD           err;
MCI_PLAY_PARMS playParams;
TCHAR           buffer[128];

/* Play a Sequencer device. Assume that midiParams.wDeviceID was set according to
our open example above */
if ((err = mciSendCommand(midiParams.wDeviceID, MCI_PLAY, MCI_WAIT, (DWORD)(LPVOID)
&playParams)))
{
    /* Error */
    printf("ERROR: Midi did not play!\r\n");
    if (mciGetErrorString(err, &buffer[0], sizeof(buffer))) printf("%s\r\n", &buffer
[0]);
}
else
{
    /* The midi song has played from beginning to end */
}
```

You can download my [Message Midi Play](#) C example to show how to play a MIDI file using the Command Message interface. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it. Remember that all apps should include MMSYSTEM.H and link with WINMM.LIB (or MMSYSTEM.LIB if Win3.1). This is a ZIP archive. Use an unzip utility that supports long filenames.

Closing the Sequencer Device

To close the Sequencer device, you issue a close command to it.

If you're using mciSendCommand(), then you need to initialize and pass a MCI_GENERIC_PARMS structure. (Note that this structure is a subset of all of the other MCI structures that you pass to mciSendCommand(). In other words, you can substitute any of the other MCI structures for this one). Unless you use the MCI_NOTIFY flag, there is actually no initialization required.

The second arg will be MCI_CLOSE to indicate a close command.

The first arg will be the device ID that you obtained when you opened the device.

Here then is an example of closing the Sequencer device using the Command Message interface.

```
/* Close the Sequencer device. Assume that midiParams.wDeviceID was set according to
our open example above */
mciSendCommand(midiParams.wDeviceID, MCI_CLOSE, MCI_WAIT, (DWORD)(LPVOID)
&midiParams);
```

If you don't want the hassle of loading/parsing WAVE files, nor handling buffers of digital audio data, then you can use the Windows High level Digital Audio API. These functions can use the MCI Wave Audio Device to play or record an entire WAVE file on its own, in the background (ie, while your app does other things). But, you lose control over Digital Audio playback/recording other than being able to stop, start, pause, rewind, and fast-forward the playback/recording. There's Windows MCI functions for setting up, and controlling the MCI Wave Audio Device, for example using mciSendCommand() to send commands to the the MCI Wave Audio Device such that it opens a WAVE file on disk, reads it in, and plays it back.

Very simple digital audio playback: PlaySound()

Windows has a function called PlaySound(). Calling this function causes Windows to play an entire, digital audio waveform that has been loaded into memory upon the default Digital Audio Output device. (The default device is whatever device the user has picked out for Audio playback upon the Audio page of Control Panel's MultiMedia utility). It's the easiest method for playing a waveform, but offers the least amount of control, for example, you can only start and stop the playback. (ie, You can't set the playback to start at a certain location in the waveform). Of course, as its name implies, it is only concerned with playing, not recording, digital audio.

PlaySound() can open some waveform file (on disk), and load the waveform data into memory prior to playback. PlaySound() completely manages opening, loading, and closing the file, Of course, one of the args you pass to PlaySound() lets it know which WAVE file to play.

Furthermore, it has the option of doing this in the background. (ie, PlaySound can return immediately while Windows goes off and does all of the work of loading and playing that waveform file in the background). Of course, one of the args you pass to PlaySound() lets it know whether you want this option (or other options to be discussed later).

But there are a few caveats to this approach

1. The waveform must be stored in WAVE File Format. This is the standard format for digital audio on an Intel based PC. PlaySound does not deal with raw waveform data (ie, stored without the appropriate WAVE File Format headers). It also cannot play AIFF or MPEG audio files.

2. The entire waveform must be able to fit into available memory. If you need to play a waveform larger than available memory (by loading it in smaller "blocks"), then you'll either have to use the other High level options (ie, the Wave Audio device) described later, or use the Low level Digital Audio API.

3. Because the entire waveform has to be completely loaded before playback even starts, with large files, this can result in a significant delay before the audio actually starts. An exception to this is using the SND_MEMORY flag described later.

4. The PlaySound function can't really be used simultaneously by multiple threads in the same process. It provides no means to arbitrate your threads' calls to PlaySound(), so unless you do that yourself, results may be unpredictable.

Playing WAVE files on disk

In order to tell PlaySound() to load and play a particular WAVE file on disk, you pass a pointer to the filename, and also set the SND_FILENAME flag. PlaySound() will completely load and play the WAVE file, and then return when that playback is finished.

Here I play the file C:\WINDOWS\CHORD.WAV and wait for it to finish:

```
if (!PlaySound("C:\\WINDOWS\\CHORD.WAV", 0, SND_FILENAME))
{
    printf("The sound didn't play!");
}
```

NOTE: It is possible to omit the SND_FILENAME flag. In this case, Windows first searches the WIN.INI file for any [Sound] key names (described later) that match the name you've passed, and then if none match, it searches for a matching filename on disk. But to avoid possible name collisions with WIN.INI [Sound] keys, I recommend using the SND_FILENAME flag.

Background playback

If you want PlaySound() to return immediately (so that your program can go on to do other things) while Windows loads/ plays the file in the background, then also specify the SND_ASYNC flag.

Here I play the file C:\WINDOWS\CHORD.WAV in the background:

```
if (!PlaySound("C:\\WINDOWS\\CHORD.WAV", 0, SND_ASYNC | SND_FILENAME))
{
    printf("The sound didn't play!");
}

/* Now I can do other things while that waveform is playing */
```

Stopping playback

Of course, Windows stops the playback automatically when it gets to end of the waveform (unless you loop it as described later).

But what if you want to stop playback prematurely? (ie, You're a premature audiojaculator). First of all, you must use the SND_ASYNC flag (ie, specify an asyncronously playing waveform). After all, if you don't, then PlaySound() doesn't return until the waveform is completely finished playing.

As noted, when you specify the SND_ASYNC flag, then the call to PlaySound() returns while Windows loads and plays the waveform in the background. Your program can then go on to do other things. One of the things that it can do is make another call to PlaySound() to play another WAVE file. So what happens to that first waveform that has been playing in the background? Windows stops playing it and starts playing the new WAVE that you've specified. In conclusion, one way to prematurely stop a waveform from playing is to simply call PlaySound() again to play another waveform. An exception to this is if your second call to PlaySound() specifies the SND_NOSTOP flag as described later.

Alternately, if you want to prematurely stop a waveform's playback without starting another waveform playing, then simply call PlaySound(), passing a 0 as the first arg. Here, I stop any currently playing waveform:

```
PlaySound(0, 0, 0);
```

NOTE: You must make this above call from the same thread that started the wave playing.

Preventing premature playback stop

So what if you don't wish to prematurely stop the playback of some asyncronously playing waveform when you make another call to PlaySound()? Simply specify the SND_NOSTOP flag. Does this mean that the two waveforms then overlap in playback? (ie, Does the second waveform start playing along with the first, thus mixing the two waveforms' playback?) You wish! Unfortunately, it just means that if some asyncronously playing waveform is still playing when you call PlaySound() with the SND_NOSTOP flag, then PlaySound() returns immediately with a 0 indicating an error. It doesn't interrupt the previous playing waveform in order to play the new waveform. (For easy, real-time mixing of digital audio, you should look at the DirectSound API). So, SND_NOSTOP simply allows you to prevent one call to PlaySound() from inadvertently cutting off the playback of a previous call to PlaySound().

Waveform looping

You can also have PlaySound() continuously loop the playback (ie, as soon as Windows gets to the end of the waveform, it immediately starts playing again from the start of the waveform). You specify the SND_LOOP flag. In this case, you must also specify the SND_ASYNC flag so that PlaySound() returns immediately. The looping then continues (in the background) until you call PlaySound() again, either to play another waveform, or to stop the currently playing waveform.

Here is an example of playing a looped waveform in the background:

```
BOOL result;

/* Tell Windows to load and continuously loop the waveform */
result = PlaySound("C:\\WINDOWS\\CHORD.WAV", 0, SND_LOOP | SND_ASYNC | SND_FILENAME);

/* Go do something else here while the waveform is looping */

/* Now finally stop the looped playback (without playing another waveform) */
if (result) PlaySound(0, 0, 0);
```

What happens in the case of an error

If the WAVE file does not exist, or doesn't fit into the available memory, PlaySound() plays the default "system sound". What is that? That is whatever WAVE file the user has set to play as the "Default Sound" via the Control Panel's *Sound* utility. (ie, The user can set various WAVE files to play for various "operations" he performs. For example, he can set a WAVE file of a door opening to play every time that he opens a Desktop folder. These are referred to as "system sounds", and are set via the Sound utility). Of course, if the user hasn't set a WAVE file for the Default Sound, then PlaySound plays nothing. It simply returns 0 indicating an error.

If you don't want PlaySound() to play the Default Sound when it can't find or load your desired WAVE file, then specify the SND_NODEFAULT flag. In this case, if PlaySound() can't find/load your desired file, it plays nothing, and simply returns a 0 to indicate an error.

WIN.INI [Sound] keys

PlaySound() can also play sounds referred to by a keyname under the [Sound] section of the WIN.INI file. To play one of these "sound events", pass a pointer to the name of the key that identifies the sound and don't specify the SND_FILENAME flag. For example, let's assume that the [Sound] section of the WIN.INI file looks like this:

[Sound]
MouseClick = C:\WINDOWS\CHORD.WAV

To play the sound associated with the "MouseClick" key (ie, the WAVE file named "C:\WINDOWS\CHORD.WAV") and to wait for the sound to complete before returning, you do:

```
PlaySound("MouseClick", 0, SND_NODEFAULT);
```

The names of some [Sound] keys that you'll find on all Win32 platforms are:

SystemAsterisk
SystemExclamation
SystemExit
SystemHand
SystemQuestion
SystemStart

There may be other [Sound] keys. In fact, your program can add more keys to WIN.INI's [Sound] section using WriteProfileString(). For example, maybe your installer will add a key name of JoeCompanyWavePlayerStart with its value set to "C:\START.WAV". Then, when your program starts up, it can call PlaySound() passing the string

"JoeCompanyWavePlayerStart". An advantage of this is that the user can easily reassign different WAVE files to the various keys you've added, by editing the WIN.INI file. (ie, It's a text file).

But on Win32, you can add keys to the registry instead of the WIN.INI file, and group them under your own heading. Instead of having to edit the WIN.INI file, the user can use Control Panel's Sound utility to edit your keys. (ie, All of your keys will be listed under your heading, and you can give a more meaningful label to each key as well). The net result is that when they are displayed in Control Panel's Sound utility, the user sees all of your keys grouped under a heading of your choice with descriptive names. It makes it a lot easier for him to edit those WAVE associations. You can download my Playsound Registry C example to show how to add such registry keys and then play the waves associated with them. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it. Remember that all apps should include MMSYSTEM.H and link with WINMM.LIB (or MMSYSTEM.LIB if Win3.1). This is a ZIP archive. Use an unzip utility that supports long filenames.

Playing embedded WAVE resources

PlaySound() can even be used to load and play a WAVE resource embedded in your program's executable. To do so, specify the SND_RESOURCE flag. In this case, the first arg is a pointer to a string that specifies the ID of your embedded resource,

Of course, when you create your executable, you should make sure that you embed the desired WAVE file in it as a resource.

For example, assume that you have the following WAVE resource in your resource file (referencing the C:\WINDOWS \CHORD.WAV file)

```
IDR_WAVE1   WAVE    DISCARDABLE     "c:\\windows\\chord.wav"
```

And furthermore, let's say that the IDR_WAVE1 symbol is defined to be the value 129.

Here is how you play that WAVE resource:

```
PlaySound("#129", 0, SND_RESOURCE | SND_NODEFAULT);
```

Download my Playsound Resource C example to show how to play a WAVE resource. Included are the Project Workspace files for Visual C++ 4.0. This example is a plain C program that any Windows compiler should be able to compile.

If the WAVE isn't embedded in your executable, for example, it's in a resource embedded in a Dynamic Link Library, then pass a handle to that object (ie, for example, a handle to the DLL) as the second arg to PlaySound(). One good use of waveforms embedded in DLLs, is that you can have different "libraries" of WAVE files, and easily switch between them just by referencing different DLLs. For example, you can have a DLL that contains 8-bit versions of your waveforms versus a DLL that contains 16-bit versions of the same. You don't have to use different filenames for the waveforms -- just different filenames for the DLLs (ie, LoadLibrary() different DLLs depending upon which versions of waves to use).

Playing WAVEs already in memory

PlaySound() can play a WAVE file that is already loaded into memory. To do so, specify the SND_MEMORY flag. In this case, the first arg is a pointer to the memory buffer containing the waveform.

Doing this eliminates most all of the delay prior to playback that you may get using other PlaySound() methods.

One caveat is that the memory buffer you pass to PlaySound() must contain the image of a complete WAVE File Format file. It can't just be raw waveform data.

Download my Playsound Memory C example to show how to play a WAVE file in memory. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it.

If you need a more sophisticated way of playing/recording digital audio data than PlaySound() offers (but don't want to resort to the Low level Digital Audio API), then you need to utilize the MCI Wave Audio Device. This is a part of Windows that manages a lot of the playback/recording process. But because it offers you a bit more direct control over the playback/recording, needless to say, it requires a bit more programming than using the very simple approach of PlaySound().

The Wave Audio device supports playing a WAVE file, without requiring that it fit into available RAM. And for large WAVE files, it can start playback quicker than PlaySound() because the Wave Audio device loads/plays the waveform in "smaller blocks" (ie, spools the waveform data into RAM) rather than loading it entirely into RAM prior to playback.

Before proceeding, you should now read MCI Devices. This article gives necessary background information about the MCI Wave Audio Device.

Opening the Wave Audio Device

To open the Wave Audio Device, you need to issue an "open" command, using either using mciSendString() or mciSendCommand() (depending upon whether you want to specify your open command by passing formatted strings, or binary values/structures, respectively), and specify that you want the Wave Audio device as the type of device opened. Of course, you also need to supply the name of a WAVE file that you want the Wave Audio to open and perform operations upon.

If you're using mciSendString(), you literally include "type waveaudio" as part of the command string to indicate that you're opening the Wave Audio device.

You can also specify an *alias*. This is just a string name that you use to identify the open device, Think of it as a string version of a device handle. After all, the string interface doesn't return binary values, so it can't return a handle. Instead it allows you to pick out a name, which you'll use with other commands you issue, in lieu of having a handle.

Here then is an example of opening the Wave Audio device using the Command String interface. The WAVE file that we ask it to open is named C:\WINDOWS\CHORD.WAV. The alias we give this instance of the Wave Device is A_Chord. (ie, Whenever we subsequently use A_Chord as the device name with other commands, we'll be performing operations on the C:\WINDOWS\CHORD.WAV file).

```
TCHAR   buf[128];
DWORD   err;

if ((err = mciSendString("open C:\\WINDOWS\\CHORD.WAV type waveaudio alias A_Chord",
0, 0, 0)))
{
    /* Error */
    printf("Wave Audio device did not open!\r\n");
    if (mciGetErrorString(err, &buf[0], sizeof(buf))) printf("%s\r\n", &buf[0]);
}
```

If you're using mciSendCommand(), then you need to initialize and pass a MCI_WAVE_OPEN_PARMS structure. You set the lpstrDeviceType field of this structure to MCI_DEVTYPE_WAVEFORM_AUDIO. You also must pass mciSendCommand() the MCI_OPEN_TYPE and MCI_OPEN_TYPE_ID flags to indicate that you've set the lpstrDeviceType field to a predefined constant (MCI_DEVTYPE_WAVEFORM_AUDIO).

If playing back a WAVE file, then you also need to specify the name of the WAVE file to open by setting the lpstrElementName field to point to the name of the desired file. You also must pass mciSendCommand() the MCI_OPEN_ELEMENT flag to indicate that you've set the lpstrElementName field. If recording a WAVE, you have the option to specify the name now, or do so later with a Save operation.

The second arg will be MCI_OPEN to indicate an open command.

Here then is an example of opening the Wave Audio device using the Command Message interface.

```
DWORD                err;
MCI_WAVE_OPEN_PARMS  waveParams;
TCHAR                buffer[128];

/* Open a Wave Audio device associated with the C:\WINDOWS\CHORD.WAV file */
waveParams.lpstrDeviceType = (LPCSTR)MCI_DEVTYPE_WAVEFORM_AUDIO;
waveParams.lpstrElementName = "C:\\WINDOWS\\CHORD.WAV";
if ((err = mciSendCommand(0, MCI_OPEN, MCI_WAIT|MCI_OPEN_ELEMENT|MCI_OPEN_TYPE|
MCI_OPEN_TYPE_ID, (DWORD)(LPVOID)&waveParams)))
{
    /* Error */
    printf("ERROR: Wave Audio device did not open!\r\n");
    if (mciGetErrorString(err, &buffer[0], sizeof(buffer))) printf("%s\r\n", &buffer
[0]);
}
else
{
    /* The device opened successfully. waveParams.wDeviceID now contains the device
ID */
}
```

Playing the Wave Audio Device

To have the Wave Audio device plays its open WAVE file, you issue a play command to it.

If you're using mciSendCommand(), then you need to initialize and pass a MCI_PLAY_PARMS structure. (Actually, unless you use the MCI_NOTIFY, MCI_FROM, and/or MCI_TO flags, there is no initialization required). If you want to play only part of the waveform, then you can set the dwFrom and/or dwTo fields to the start and end positions respectively, and pass the MCI_FROM and/or MCI_TO flags respectively. The dwFrom and dwTo fields are normally expressed in terms of sample offset (ie, a byte offset). But you can utilize other means of expressing this offset, for example in terms of milliseconds, by first issuing an MCI_SET command (with the MCI_SET_TIME_FORMAT flag, and your desired choice of how to express such offsets).

The second arg will be MCI_PLAY to indicate a play command.

The first arg will be the device ID that you obtained when you opened the device.

Here then is an example of playing a WAVE on the Wave Audio device using the Command Message interface.

```
DWORD            err;
MCI_PLAY_PARMS   playParams;
TCHAR            buffer[128];

/* Play a Wave Audio device. Assume that waveParams.wDeviceID was set according to
our open example above */
if ((err = mciSendCommand(waveParams.wDeviceID, MCI_PLAY, MCI_WAIT, (DWORD)(LPVOID)
&playParams)))
{
    /* Error */
    printf("ERROR: Wave did not play!\r\n");
    if (mciGetErrorString(err, &buffer[0], sizeof(buffer))) printf("%s\r\n", &buffer
[0]);
}
```

```
else
{
    /* The wave has played from beginning to end */
}
```

You can download my [MCI Wave Play](#) C example to show how to play a WAVE file using the Command Message and String interfaces. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it. Remember that all apps should include MMSYSTEM.H and link with WINMM.LIB (or MMSYSTEM.LIB if Win3.1). This is a ZIP archive. Use an unzip utility that supports long filenames.

Closing the Wave Audio Device

To close the Wave Audio device, you issue a close command to it.

If you're using mciSendCommand(), then you need to initialize and pass a MCI_GENERIC_PARMS structure. (Note that this structure is a subset of all of the other MCI structures that you pass to mciSendCommand(). In other words, you can substitute any of the other MCI structures for this one). Unless you use the MCI_NOTIFY flag, there is actually no initialization required.

The second arg will be MCI_CLOSE to indicate a close command.

The first arg will be the device ID that you obtained when you opened the device.

Here then is an example of closing the Wave Audio device using the Command Message interface.

```
/* Close the Wave Audio device. Assume that waveParams.wDeviceID was set according
to our open example above */
mciSendCommand(waveParams.wDeviceID, MCI_CLOSE, MCI_WAIT, (DWORD)(LPVOID)
&waveParams);
```

Recording with the Wave Audio Device

You can download my [MCI Wave Record](#) C example to show how to record a WAVE file using the Command Message and String interfaces. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it.

To use the MIDI Stream API, you need to first call midiStreamOpen() once to open some MIDI stream device for output. (This is similiar to the Low level MIDI API's midiOutOpen). You pass the Device ID of that desired stream device. Then, you can subsequently call other Stream functions, such as midiStreamOut() to queue MIDI messages for playback upon that device, and midiStreamRestart() to start the actual playback. Windows and that device will take care of timing out each event and outputting its MIDI bytes.

After you're done outputting to a stream device (and have no further use for it), you must close that device.

Think of a stream device like a file. You open it, you write to it (ie, queue MIDI events to it which Windows "plays"), and then you close it.

Opening the default Stream for playback

This is extremely similiar to the Low level MIDI API's midiOutOpen() (except that a pointer to the device ID is passed rather than the device ID directly passed, and also, there is one extra arg that is currently set to 1). To open the default device, use a Device ID of 0 as so:

```
unsigned long result, deviceID;
HMIDISTRM     outHandle;

/* Open default MIDI Out stream device */
deviceID = 0;
result = midiStreamOpen(&outHandle, &deviceID, 1, 0, 0, CALLBACK_NULL);
if (result)
{
    printf("There was an error opening the default MIDI stream device!\r\n");
}
```

The most flexible way to choose a MIDI Stream device for input or output

Windows Stream Manager is capable of streaming MIDI data to any one of the MIDI Output Devices on a system. So, in order to get a list of MIDI Out Devices that you can open with midiStreamOpen(), you simply do things exactly as you would with the low level's midiOutOpen(). You query all of the MIDI Output devices in the system using midiOutGetDevCaps(), and remember that the first device always has an ID of 0, and subsequent devices have ascending ID numbers.

Individual Windows 95 drivers for any installed cards may be specially written to directly support the Stream API. When you query a particular MIDI Output device (using midiOutGetDevCaps), you can check the dwSupport field of the MIDIOUTCAPS structure. If the driver directly supports the Stream API, the MIDICAPS_STREAM bit of dwSupport will be set. (Download my ListMidiDevs C example to show how to query the MIDI Output devices for such support). Such a driver may offer special features such as allowing the Stream Manager to sync playback to incoming SMPTE or MIDI Time Code. Those features would be determined by what code has been added to the driver, as well as perhaps hardware support by the MIDI card itself.

One caveat is that the MIDI Mapper cannot be opened as the output for a MIDI stream. The Windows MIDI Stream Manager supports only one MIDI output at a time (unless the card's driver directly supports the Stream API and adds some sort of routing feature based upon the MIDIEVENT's dwStreamID field. Currently, the Stream Manager doesn't support this routing).

General overview of stream playback

The process of passing the MIDI events that you want Windows to time out and output is similiar to passing a buffer of data to

[midiOutLongMsg()](). In fact, you use a MIDIHDR structure, as well as calls to midiOutPrepareHeader() and midiOutUnprepareHeader(). But, you also use one or more MIDIEVENT structures, and the actual time out and output of the MIDI messages doesn't begin until you call midiStreamRestart(). (ie, Until you actually call midiStreamRestart, Windows simply queues the MIDI messages in the order you pass them).

Here's how you pass a MIDI message to be played back:

1. Place one MIDI message into a structure called a MIDIEVENT.

2. Place a pointer to this MIDIEVENT into the lpData field of a MIDIHDR structure. Also set the MIDIHDR's dwBufferLength and dwBytesRecorded fields to the size of this MIDIEVENT structure. Set the dwFlags field to 0.

3. Pass the MIDIHDR to midiOutPrepareHeader().

4. Pass the MIDIHDR to midiStreamOut().

The MIDI message is now queued for playback. If you haven't called midiStreamRestart() yet, then Windows does not yet start timing out and outputting this event. So, you can queue several MIDI messages prior to the start of playback by repeating the above steps, using a separate MIDIEVENT and MIDIHDR for each message.

It's also possible to queue several MIDI messages using one MIDIHDR and one call to midiOutPrepareHeader() and midiStreamOut(). You do this by using an array of MIDIEVENT structures (ie, one for each of your MIDI messages). Place each MIDI message into one of the MIDIEVENT structures in that array. Then place a pointer to the entire array in the MIDIHDR's lpData field. Of course, the MIDIHDR's dwBufferLength and dwBytesRecorded fields are set to the size of the entire array of MIDIEVENT structures. It is much more efficient and memory-conserving to combine all MIDI messages that occur upon the same musical beat into one array of MIDIEVENTS, and cue them with one MIDIHDR.

Note that after Windows finishes playing all of the events queued with a particular MIDIHDR, it sets the MHDR_DONE bit of the dwFlags field. This may be very useful later when you're dealing with processing MIDIHDRs as a result of being notified by Windows that a MIDIHDR's events have finished playing. Also note that a MIDIHDR has a dwUser field that you can use for your own purposes.

Now it's certainly feasible to queue up all of the MIDI messages of a sequence prior to calling midiStreamRestart() by simply putting them into one gigantic array of MIDIEVENT structures, and passing the whole array in one call to midiStreamOut(). You can download my [Simple Stream]() C example which illustrates this approach. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it. My code takes this approach in order to present the simplest possible example of using the Stream API to play a musical sequence, and therefore give you an easy introduction to how the Stream API works.

But that approach is sort of like using the High level MIDI API. After all, by passing all of the data all at once, you then lose control over too much of the playback process. The whole point of the Stream API is to feed Windows a few events at a time so that the playback doesn't get too far ahead that you lose the sense of real-time control over individual events. If you want to be doing real-time mixing of several "tracks" of MIDI messages into a single stream, giving the user the option to mute one of the tracks at any time for example, then obviously you don't want to queue MIDI messages too far in advance.

But, if you want a smooth playback, you obviously have to queue those events before they need to be played. The best approach to take is to use a sort of double-buffering scheme. In other words, you'll always have one "block" of MIDI messages queued while the current block of MIDI messages is playing. When the current block is finished playing (and the queued block starts playing), then you'll queue another block.

You'll select a brief "time window", for example 1 quarter note. (ie, Assume our "musical beat" is a quarter note). Then, before playback is started with midiStreamRestart(), you'll place all of the MIDI messages whose timing falls within the first quarter note (ie, the first beat) into an array of MIDIEVENTS (and a MIDIHDR) and queue it with midiStreamOut(). You'll also place all of the MIDI messages whose timing falls within the second quarter note (ie, second beat) into another array of MIDIEVENTS (and another MIDIHDR) and queue it with midiStreamOut(). Then, you'll call midiStreamRestart() to start

playback. As soon as the last MIDI message in the first array of MIDIEVENTS is finished playing, you'll place all of the MIDI messages whose timing falls within the third quarter note into that same array of MIDIEVENTS and queue it with midiStreamOut(). Of course, while you're doing this, the second queued array has been playing. After that array's last MIDI message is played, you'll use that array to queue another "block" of MIDI messages whose timing falls within the fourth quarter note. Etc. In this way, you always have data queued for continuous, smooth playback, but the playback is only 1 musical beat ahead of any real-time action the user instructs you to perform upon the data. So, for example, if he tells you to mute one of the tracks, maybe it won't effectively work out that way for one musical beat (some events on that track may already be queued for the next beat), but that's plenty close enough.

So what if there happens to be no MIDI events that fall within the next musical beat? What do you queue? As you'll see in the next section, you can queue a single NOP event, timed to delay for an entire beat.

The MIDIEVENT structure

Now let's take a closer look at the MIDIEVENT structure to see how various MIDI messages are stored in it. The Stream API documents it as so:

```
typedef struct {
    DWORD dwDeltaTime;
    DWORD dwStreamID;
    DWORD dwEvent;
    DWORD dwParms[];
} MIDIEVENT;
```

The dwDeltaTime field is just like the timing in MIDI File Format (except that it's an unsigned long rather than a variable length quantity). It represents the amount of time to delay before outputting this MIDI message. You can specify the time in terms of PPQN clocks, or a SMPTE time. (When using the former, you can send "Tempo Events" to the stream device to change the tempo, or call a Stream function that allows you to change the tempo on the fly. When using the latter, Windows can sync the MIDI playback to streaming video or other SMPTE cues).

Currently, the dwStreamID field isn't used and should be set to 0. (My own tests show that whatever value you stuff into this field is ignored, but unless you're using a driver that directly supports the MIDI Stream API and the driver uses this field, it is safest to set it to 0 in case some future version of Windows utilizes this field).

Although declared as an unsigned long, the 4 bytes of the dwEvent field are actually individual pieces of information. The highest byte contains some flag bits and some bits that form an "event type" value. I'll refer to this as the Event Type byte.

If this MIDIEVENT contains a normal MIDI Voice message such as a Note-On, Program Change, Aftertouch, or any of the other MIDI messages that are 3 or less bytes, then the Event Type byte is set to the value MEVT_SHORTMSG.

In this case, the remaining 3 bytes of this field are the MIDI Status byte, the first MIDI data byte (if any), and the second MIDI data byte (if any). Note that this is packed up exactly the way that a MIDI message is passed to midiOutShortMsg(). Do not use running status. The stream device will implement running status when it outputs the MIDI messages.

Here then is how you would initialize a MIDIEVENT with a Note-On MIDI message for middle C on channel 1 (with velocity of 0x40) and a delta time of 0 (gets output as soon as Windows finishes playing the previously queued event):

```
MIDIEVENT mevt;

mevt.dwDeltaTime = 0;
mevt.dwStreamID = 0;
mevt.dwEvent = ((unsigned long)MEVT_SHORTMSG<<24) | 0x00403C90;
```

Note that since MEVT_SHORTMSG is really 0x00, we can simply the above by actually removing the setting of the Event Type byte (since its already 0 in our packed MIDI message):

mevt.dwEvent = 0x00403C90;

What about that dwParms field? Well, if you look closely at the declaration, it isn't really there. There's no size for this dwParms array. Say what?!? That's right. There is no dwParms field on the above MIDIEVENT. The structure really has only 3 fields. It should have been declared as something like this (which I'll call MY_MIDI_EVT):

```
typedef struct {
    DWORD dwDeltaTime;
    DWORD dwStreamID;
    DWORD dwEvent;
} MY_MIDI_EVT;
```

So what was that dwParms field doing there? Well, here's where it gets tricky. There are other types of events that you can pass. For example, consider a System Exclusive event. It can have many more than 3 bytes. Where do you put all of those bytes? Well, **now** you use that extra dwParms array appended to the end. Its size is set to however many bytes you need to pass. In essense, you're appending the data to the end of the MIDIEVENT structure. That's right. The size of the MIDIEVENT structure you pass will vary depending upon the type of event it contains.

Let's take an example where we want to pass a System Exclusive event that consists of 7 bytes. Well, the first thing that we need to do is redeclare the MIDIEVENT structure, thereby creating a new structure. I'll call this MY_SYSEX_EVT.

```
typedef struct {
    DWORD         dwDeltaTime;
    DWORD         dwStreamID;
    DWORD         dwEvent;
    unsigned char dwParams[8];
} MY_SYSEX_EVT;
```

See? I declared an array large enough to hold my extra bytes. Why did I use a size of 8 rather than 7? Well, the Stream API requires that the size of all MIDIEVENT structures passed to it be aligned on a doubleword boundary. So, if I'm going to stuff this MIDIEVENT into a buffer containing another MIDIEVENT that comes after it, I want that second MIDIEVENT to be properly aligned. In essense, I've added a pad byte above to make the structure's size a multiple of 4, and thereby ensure that subsequent MIDIEVENT structures in the same array are properly aligned.

OK, now let's initialize it. For such a MIDIEVENT, its Event Type byte must be the value MEVT_LONGMSG. Furthermore, the remaining 3 bytes of the dwEvent field must be a 24-bit count of the number of bytes in our System Exclusive message (ie, 7 in this case).

```
MY_SYSEX_EVT  mevt;
unsigned char sysEx[] = {0xF0, 0x7F, 0x7F, 0x01, 0x02, 0xF7};

mevt.dwDeltaTime = 100; /* Just for the hell of it, delay 100 clocks before sending it */
mevt.dwStreamID = 0;
mevt.dwEvent = ((unsigned long)MEVT_LONGMSG<<24) | sizeof(sysEx);
memcpy(&mevt.dwParams[0], &sysEx[0], sizeof(sysEx));
```

What other Event Types are there (besides the "short" type for MIDI messages of 3 bytes or less, and the "long" type for System Exclusive)? The most useful is a Tempo event. For such a MIDIEVENT, its Event Type byte must be the value MEVT_TEMPO. Furthermore, the remaining 3 bytes of the dwEvent field must be a Tempo value as a 24-bit value. It is expressed in microseconds per quarter note, just like in the MIDI File Format's MetaTempo event.

Another Event Type is a comment. Like a System Exclusive MIDIEVENT, the characters that form the comment are

appended to the end of the MIDIEVENT structure itself. Its Event Type byte must be the value MEVT_COMMENT. Furthermore, the remaining 3 bytes of the dwEvent field must be a 24-bit count of the number of chars in the comment. Again, pad out the structure's size to a multiple of 4 if you're going to put it in an array with other MIDIEVENTs after it. Windows ignores comment events, so you can use them for your own purposes.

Another Event Type is a NOP (no operation). It's a "short" type of event, like Tempo and regular MIDI Voice messages, so no extra fields are appended to the MIDIEVENT structure. Its Event Type byte must be the value MEVT_NOP. Furthermore, the remaining 3 bytes of the dwEvent field can be used for any purpose you wish. Windows ignores NOP events, so you can use them for your own purposes.

The final Event Type is a version event. It has a MIDISTRMBUFFVER structure appended to the end of the MIDIEVENT structure. This extra structure just contains version information about the Stream. Its Event Type byte must be the value MEVT_VERSION. Furthermore, the remaining 3 bytes of the dwEvent field must be a 24-bit count of the size of the MIDISTRMBUFFVER structure.

OK, I'm sure that there is one nagging question in your mind. If MIDIEVENT structures can be variable size, depending upon the Event Type, then how do you declare a static array of them? Well, you can't really (unless you happen to stick to only using the "short" types). What you'll need to do is just copy them into one large char buffer, using a pointer to that buffer, and do some creative casting. For example, here I copy two MIDIEVENTs into one buffer:

```
MY_SYSEX_EVT * xevt;
MY_MIDI_EVT *  mevt;
unsigned char  sysEx[] = {0xF0, 0x7F, 0x7F, 0x01, 0x02, 0xF7};
unsigned char  buffer[20];

/* Format for a Note-On */
mevt = (MY_MIDI_EVT *)&buffer[0];
mevt->dwDeltaTime = 0;
mevt->dwStreamID = 0;
mevt->dwEvent = 0x00403C90;
mevt++;

/* Format for a System Exclusive */
xevt = (MY_SYSEX_EVT *)mevt;
xevt->dwDeltaTime = 100;
xevt->dwStreamID = 0;
xevt->dwEvent = ((unsigned long)MEVT_LONGMSG<<24) | sizeof(sysEx);
memcpy(&xevt->dwParams[0], &sysEx[0], sizeof(sysEx));
```

Of course, I should make sure that buffer is aligned upon a doubleword boundary (which you can do with your compiler's alignment directive).

But another approach to take is to simply declare your array to be an array of unsigned longs. After all, since all MIDIEVENT structures need to be padded out to a multiple of 4 bytes, then the net result is that each MIDIEVENT structure consists 3 or more unsigned longs. So, here's the above array initialized this way:

```
unsigned long  buffer[] = { 0, 0, 0x00403C90, /* The Note-On */
                            0, 0, ((unsigned long)MEVT_LONGMSG<<24) | 7, 0x017F7FF0,
0x00F70201}; /* The SysEx */
```

Note the order of the packed SysEx bytes. Each unsigned long contains the next 4 bytes, and remember that we're using little endian order on each unsigned long.

Starting playback

You start playback by calling midiStreamRestart() as so:

```
unsigned long    err;

err = midiStreamRestart(outHandle);
if (err)
{
    printf("An error starting playback!\n");
}
```

Note that when playback begins, the stream device's current time is set to 0 (if the playback hasn't been paused).

Stop the current playback

You can stop a playback in progress by calling midiStreamStop() as so:

```
unsigned long    err;

err = midiStreamStop(outHandle);
if (err)
{
    printf("An error stopping playback!\n");
}
```

This flushes all of the queued MIDIEVENTs, and you can subsequently midiOutUnprepareHeader() the MIDIHDRs. (The MHDR_DONE bit is set in the dwFlags field of all queued MIDIHDR structures).

It also turns off any notes that are still turned on. (By contrast, midiOutReset() turns off all notes regardless, and is more of a "panic" button type of response to turn off any "stuck notes").

Calling this function when the output is already stopped has no effect, and doesn't return an error.

WARNING! WARNING! WARNING! The Windows Stream Manager appears to be severely broken. Calling midiStreamStop() (or midiStreamPause) does indeed stop playback, but it appears to also close the stream handle that you pass to it. The result is that a subsequent call to another function using that handle results in the Stream Manager returning an error that the handle is no longer valid. The only options you have are:

1. Open a stream device immediately before you intend to queue and play MIDI events. Play those events. Then close the stream device.

2. Once you call midiStreamRestart(), never call midiStreamStop() or midiStreamPause(). Simply maintain your own clock variable and "play flag" variable that your callback can test to see whether it should continue queueing MIDI events for playback and increment its clock, or just do nothing. In other words, you're going to start up the stream once and leave it constantly running in the background, waiting for more MIDIEVENTs to be queued.

Pause/Resume the current playback

You can pause a playback in progress by calling midiStreamPause() as so:

```
unsigned long    err;

err = midiStreamPause(outHandle);
```

```
if (err)
{
    printf("An error pausing playback!\n");
}
```

If you wish to subsequently resume playback from the point at which it was paused, then call midiStreamRestart(). The stream device's time is not reset to 0, and playback resumes. If instead, you wish to stop the device and flush the queued MIDIEVENTs (perhaps in order to reset its time to 0 and start playback from the beginning of a sequence), then instead call midiStreamStop().

Calling this function when the output is already paused has no effect, and doesn't return an error.

Setting/Querying the Tempo and Timebase

Before playing queued MIDIEVENTs, you'll want to set the Timebase for the stream device. This is equivalent to the MIDI File Format's Division. It tells the stream device how to scale the dwDeltaTime field of each MIDIEVENT. (ie, Consider it a SMPTE time in 30 fps, or a time-stamp at 96 PPQN, or a time-stamp at 120 PPQN, etc). You use midiStreamProperty() with the MIDIPROP_SET and MIDIPROP_TIMEDIV flags as the third arg. The second arg is a pointer to a MIDIPROPTIMEDIV structure whose dwTimeDiv field is initialized to your desired Timebase. For PPQN Timebases, just specify the PPQN value, for example 96 PPQN. For SMPTE, the low two bytes are as per the MIDI File Format's Division field when SMPTE is specified.

Here I set the Timebase to 96 PPQN (which is the default if you don't specify a Timebase):

```
unsigned long   err;
MIDIPROPTIMEDIV prop;

prop.cbStruct = sizeof(MIDIPROPTIMEDIV);
prop.dwTimeDiv = 96;
err = midiStreamProperty(outHandle, (LPBYTE)&prop, MIDIPROP_SET|MIDIPROP_TIMEDIV);
if (err)
{
    printf("An error setting the timebase!\n");
}
```

Besides putting Tempo events in the stream in order to set tempo at any given time, you can also call midiStreamProperty() with the MIDIPROP_SET and MIDIPROP_TEMPO flags as the third arg to set tempo. The second arg is a pointer to a MIDIPROPTEMPO structure whose dwTempo field is initialized to your desired Tempo. Note that this is irrelevant when using a SMPTE Timebase.

Here I set the Tempo to 120 BPM (which is the default if you don't specify a Tempo):

```
unsigned long   err;
MIDIPROPTEMPO   prop;

prop.cbStruct = sizeof(MIDIPROPTEMPO);
prop.dwTempo = 0x0007A120;
err = midiStreamProperty(outHandle, (LPBYTE)&prop, MIDIPROP_SET|MIDIPROP_TEMPO);
if (err)
{
    printf("An error setting the tempo!\n");
}
```

Of course, you can query a stream device's current timebase or tempo by using the MIDIPROP_GET flag instead of MIDIPROP_SET. Windows fills in the MIDIPROPTIMEDIV or MIDIPROPTEMPO you pass.

Here I query the current Tempo:

```
unsigned long    err;
MIDIPROPTEMPO    prop;

prop.cbStruct = sizeof(MIDIPROPTEMPO);
err = midiStreamProperty(outHandle, (LPBYTE)&prop, MIDIPROP_GET|MIDIPROP_TEMPO);
if (err)
{
    printf("An error requesting the tempo!\n");
}
else
{
    printf("Tempo = %u\n", prop.dwTempo);
}
```

Notification during playback

I previously talked about how you want to queue up another array of MIDIEVENTs as soon as one array finishes playing. So how do you receive such notification from Windows? When you call midiStreamOpen(), you can tell it how you want Windows to notify you. In our above example, we specified CALLBACK_NULL (ie, we didn't want Windows to notify us). But there are other choices as follows:

1. CALLBACK_EVENT -- You allocate some Event with CreateEvent(), and Windows uses this to signal your app. (ie, Your app can wait on that Event signal, for example with WaitForSingleObject). You pass the handle of the Event as the 4th arg to midiStreamOpen().

2. CALLBACK_THREAD -- Windows causes some suspended thread within your app to run. (ie, Your app's thread can suspend itself via SuspendThread). You pass the Thread ID of the desired thread to be run as the 4th arg to midiStreamOpen().

3. CALLBACK_WINDOW -- Windows sends a message to some open window in your app. The parameters for the message will contain additional information about what caused Windows to send that message. You pass the desired window's handle as the 4th arg to midiStreamOpen().

4. CALLBACK_FUNCTION -- Windows directly calls some function in your app. It passes args that contain additional information about what caused Windows to call your function. You pass a pointer to the desired function as the 4th arg to midiStreamOpen(). The 5th arg to midiStreamOpen() can be anything you desire, and this will be passed to your callback function each time that Windows calls your callback.

The latter two methods allow you to better determine what exactly caused Windows to notify you, because they supply additional information to you.

So when does Windows notify you? Here are the times when Windows notifies you:

1. When you open a Stream device via midiStreamOpen().

2. When you close a Stream Device via midiStreamClose().

3. When midiStreamOut() encounters an event that has its MEVT_F_CALLBACK flag set.

4. When midiStreamOut() finishes playing a MIDIHDR's block of data.

Here's an example of setting the MEVT_F_CALLBACK flag of an event.

```
xevt->dwEvent = ((unsigned long)MEVT_LONGMSG<<24) | sizeof(sysEx) | MEVT_F_CALLBACK;
```

Now, after Windows plays that above event, it notifies you. For example, if you've chosen CALLBACK_FUNCTION method, Windows calls your callback function.

Windows also will notify you when the last event in a MIDIHDR's block of events is played. This is the point at which you will queue the next array of MIDIEVENTs using that same array (and its associated MIDIHDR).

In fact, you could strategically place NOP events with their dwDeltaTime set so that they just happen to fall upon each musical downbeat, and set the MEVT_F_CALLBACK flag of each of those event's dwEvent fields. Then, everytime that your callback is called, you can update a graphical display counting off the beat. (My Stream Callback C example shows that using CALLBACK_FUNCTION method).

Here's an example of using the CALLBACK_EVENT method to play an array of MIDIEVENTs. This is a complete example that shows you all the bare minimum details you need to know to play a stream of MIDIEVENTs.

```
/* The array of MIDIEVENTs to be output. We only have 2 */
unsigned long myNotes[] = {0, 0, 0x007F3C90, /* A note-on */
192, 0, 0x00003C90}; /* A note-off. It's the last event in the array */


HANDLE          event;
HMIDISTRM       outHandle;
MIDIHDR         midiHdr;
MIDIPROPTIMEDIV prop;
unsigned long   err;

/* Allocate an Event signal */
if ((event = CreateEvent(0, FALSE, FALSE, 0)))
{
    /* Open default MIDI Out stream device. Tell it to notify via CALLBACK_EVENT and
use my created Event */
    err = 0;
    if (!(err = midiStreamOpen(&outHandle, &err, 1, (DWORD)event, 0,
CALLBACK_EVENT)))
    {
        /* Windows signals me once when the driver is opened. Clear that now */
        ResetEvent(event);

        /* Set the timebase. Here I use 96 PPQN */
        prop.cbStruct = sizeof(MIDIPROPTIMEDIV);
        prop.dwTimeDiv = 96;
        midiStreamProperty(outHandle, (LPBYTE)&prop, MIDIPROP_SET|MIDIPROP_TIMEDIV);

        /* If you wanted something other than 120 BPM, here you should also set the
tempo */

        /* Store pointer to our stream (ie, array) of messages in MIDIHDR */
        midiHdr.lpData = (LPBYTE)&myNotes[0];
```

```c
        /* Store its size in the MIDIHDR */
        midiHdr.dwBufferLength = midiHdr.dwBytesRecorded = sizeof(myNotes);

        /* Flags must be set to 0 */
        midiHdr.dwFlags = 0;

        /* Prepare the buffer and MIDIHDR */
        err = midiOutPrepareHeader(outHandle,  &midiHdr, sizeof(MIDIHDR));
        if (!err)
        {
            /* Queue the Stream of messages. Output doesn't actually start
               until we later call midiStreamRestart().
             */
            err = midiStreamOut(outHandle, &midiHdr, sizeof(MIDIHDR));
            if (!err)
            {
                /* Start outputting the Stream of messages. This will return immediately
                    as the stream device will time out and output the messages on its own in
                    the background.
                 */
                err = midiStreamRestart(outHandle);
                if (!err)

                    /* Wait for playback to stop. Windows signals me using that Event I created */
                    WaitForSingleObject(event, INFINITE);
            }

            /* Unprepare the buffer and MIDIHDR */
            midiOutUnprepareHeader(outHandle, &midiHdr, sizeof(MIDIHDR));
        }

        /* Close the MIDI Stream */
        midiStreamClose(outHandle);
    }

    /* Free the Event */
    CloseHandle(event);
}
```

If using the CALLBACK_FUNCTION method, then you need to write a function that has the following declaration (although you can name the function anything you like):

```c
void CALLBACK midiCallback(HMIDIOUT handle, UINT uMsg, DWORD dwInstance, DWORD dwParam1, DWORD dwParam2);
```

As mentioned, you pass a pointer to this function as the 4th arg to midiStreamOpen(). The 5th arg to midiStreamOpen() can be anything you desire, and this will be passed to your callback function each time that Windows calls your callback. Windows calls your function whenever 1 of 4 possible things happen:

1. When you open a Stream device via midiStreamOpen(). In this case, the uMsg arg to your callback will be MOM_OPEN. The handle arg will be the same as what is returned to midiStreamOpen(). The dwInstance arg is whatever I passed to midiStreamOpen() as its dwInstance arg.

2. When you close a Stream Device via midiStreamClose().In this case, the uMsg arg to your callback will be MOM_CLOSE. The handle arg will be the same as what was passed to midiStreamClose(). The dwInstance arg is the 5th arg you passed to midiStreamOpen() when you initially opened this handle.

3. When midiStreamOut() encounters an event that has its MEVT_F_CALLBACK flag set. In this case, the uMsg arg to your callback will be MOM_POSITIONCB. The handle arg will be the same as what is passed to midiStreamOut(). The dwInstance arg is the 5th arg you passed to midiStreamOpen() when you initially opened this handle. The dwParam1 arg points to the MIDIHDR that you passed to midiStreamOut(). The dwOffset field of the MIDIHDR will indicate the byte offset into the buffer for the event that caused this callback to be called. See my note below.

4. When midiStreamOut() finishes playing a MIDIHDR's block of data. In this case, the uMsg arg to my callback will be MOM_DONE. The handle arg will be the same as what is passed to midiStreamOut(). The dwInstance arg is the 5th arg you passed to midiStreamOut() when you initially opened this handle. The dwParam1 arg points to the MIDIHDR that you passed to midiStreamOut().

NOTE: The dwParam2 arg is not used. This is reserved for future use.

Here's an example of using the CALLBACK_FUNCTION method to play an array of MIDIEVENTs. This is a complete example that shows you all the bare minimum details you need to know to play a stream of MIDIEVENTs.

```
/* The array of MIDIEVENTs to be output. We only have 2 */
unsigned long myNotes[] = {0, 0, 0x007F3C90, /* A note-on */
192, 0, 0x00003C90}; /* A note-off. It's the last event in the array */

HANDLE      Event;

void CALLBACK midiCallback(HMIDIOUT handle, UINT uMsg, DWORD dwInstance, DWORD
dwParam1, DWORD dwParam2)
{
    LPMIDIHDR    lpMIDIHeader;
    MIDIEVENT * lpMIDIEvent;

    /* Determine why Windows called me */
    switch (uMsg)
    {
        /* Got some event with its MEVT_F_CALLBACK flag set */
        case MOM_POSITIONCB:

            /* Assign address of MIDIHDR to a LPMIDIHDR variable. Makes it easier to
access the
                field that contains the pointer to our block of MIDI events */
            lpMIDIHeader = (LPMIDIHDR)dwParam1;

            /* Get address of the MIDI event that caused this call */
            lpMIDIEvent = (MIDIEVENT *)&(lpMIDIHeader->lpData[lpMIDIHeader-
>dwOffset]);

            /* Normally, if you had several different types of events with the
                MEVT_F_CALLBACK flag set, you'd likely now do a switch on the highest
                byte of the dwEvent field, assuming that you need to do different
                things for different types of events.
            */

            break;

        /* The last event in the MIDIHDR has played */
```

```
        case MOM_DONE:

            /* Wake up main() */
            SetEvent(Event);

            break;



        /* Process these messages if you desire */
        case MOM_OPEN:
        case MOM_CLOSE:

            break;
    }
}


int main(int argc, char **argv)
{
    HMIDISTRM       outHandle;
    MIDIHDR         midiHdr;
    MIDIPROPTIMEDIV prop;
    unsigned long   err;

    /* Allocate an Event signal */
    if ((event = CreateEvent(0, FALSE, FALSE, 0)))
    {
        /* Open default MIDI Out stream device. Tell it to notify via CALLBACK_EVENT
and use my created Event */
        err = 0;
        if (!(err = midiStreamOpen(&outHandle, &err, 1, (DWORD)midiCallback, 0,
CALLBACK_FUNCTION)))
        {
            /* Set the timebase. Here I use 96 PPQN */
            prop.cbStruct = sizeof(MIDIPROPTIMEDIV);
            prop.dwTimeDiv = 96;
            midiStreamProperty(outHandle, (LPBYTE)&prop, MIDIPROP_SET|
MIDIPROP_TIMEDIV);

            /* If you wanted something other than 120 BPM, here you should also set
the tempo */

            /* Store pointer to our stream (ie, array) of messages in MIDIHDR */
            midiHdr.lpData = (LPBYTE)&myNotes[0];

            /* Store its size in the MIDIHDR */
            midiHdr.dwBufferLength = midiHdr.dwBytesRecorded = sizeof(myNotes);

            /* Flags must be set to 0 */
            midiHdr.dwFlags = 0;

            /* Prepare the buffer and MIDIHDR */
            err = midiOutPrepareHeader(outHandle,  &midiHdr, sizeof(MIDIHDR));
            if (!err)
            {
                /* Queue the Stream of messages. Output doesn't actually start
                    until we later call midiStreamRestart().
                 */
```

```
                    err = midiStreamOut(outHandle, &midiHdr, sizeof(MIDIHDR));
                    if (!err)
                    {
                        /* Start outputting the Stream of messages. This will return
immediately
                            as the stream device will time out and output the messages on
its own in
                            the background.
                         */
                        err = midiStreamRestart(outHandle);
                        if (!err)

                            /* Wait for playback to stop. Windows calls my callback, which
will set this signal */
                            WaitForSingleObject(event, INFINITE);
                    }

                    /* Unprepare the buffer and MIDIHDR */
                    midiOutUnprepareHeader(outHandle, &midiHdr, sizeof(MIDIHDR));
                }

                /* Close the MIDI Stream */
                midiStreamClose(outHandle);
            }

            /* Free the Event */
            CloseHandle(event);
        }

        return(0);
}
```

NOTE: If you happen to have two or more events that occur upon the same beat (or close enough such that Windows doesn't even have time to call your callback once before both events time out), and both their MEVT_F_CALLBACK flags are set, then Windows only calls your callback once for all of those events. The dwOffset of the MIDIHDR would reference the last such event. Therefore, don't bother setting the MEVT_F_CALLBACK flag of more than one event that occurs upon a given time.

Furthermore, while Windows calls your callback, it continues timing out the next event in the background. If it happens that the next event happens to have its MEVT_F_CALLBACK flag also set, and it times out while your callback is still processing that previous MEVT_F_CALLBACK event, then Windows will call your callback again. In other words, if you're not sure that you've placed enough time inbetween each of your MEVT_F_CALLBACK flagged events in order to do what you need to get done in your callback, then you had better make sure your callback handles reentrancy properly. (ie, Don't use global variables whose value changes, or use some sort of mechanism to arbitrate access to those variables such as a Mutex). For practical purposes, you should avoid setting the MEVT_F_CALLBACK flag of an event whose time makes it occur less than 25 milliseconds after the previous event with its MEVT_F_CALLBACK flag set. Microsoft recommends the following approach to handling a situation where MEVT_F_CALLBACK events are timing out quicker than your callback can finish its work: "The callback should use the MIDIHDR's dwOffset to decide what to do. If it is expecting this to reference a particular location (ie, event) but it instead references a location somewhere further along in the array of MIDIEVENTs, it means that your callback is not keeping up or that the MEVT_F_CALLBACK flagged events are scheduled too closely together for the capabilities of the computer. The best thing to do in this case is to just throw the callback away because there will be another one in the queue that will be delivered just as soon as processing of the current callback is completed."

Querying the current playback time

You can retrieve the current playback time by calling midiStreamPosition(). It fills in an MMTIME structure.

Before calling midiStreamPosition(), you set the wType field of the MMTIME structure to indicate the time format you desire returned. After calling midiStreamPosition(), you should check the wType field. Some stream devices may not support returning certain time formats, for example, a SMPTE format. In that case, midiStreamPosition() will set the wType field to the closest supported format, and return that time.

The allowable time formats (ie, for the wType field) are:

```
TIME_BYTES        Current byte offset from beginning of the playback.
TIME_MIDI         MIDI time (ie, MIDI Song Position Pointer).
TIME_MS           Time in milliseconds.
TIME_SAMPLES      Number of waveform-audio samples (for syncing to WAVE playback).
TIME_SMPTE        SMPTE time.
TIME_TICKS        PPQN clocks.
```

Here I query the current Time. (Note that the Time is set to 0 when playback starts, except for a paused playback that is resumed).

```c
unsigned long    err;
MMTIME           time;

/* Request millisecond time returned */
time.wType = TIME_MS;
err = midiStreamPosition(outHandle, &time, sizeof(MMTIME));
if (err)
{
    printf("An error requesting the time!\n");
}
else switch (time.wType)
{
    case TIME_BYTES:
        printf("%u bytes have been played so far.\n", time.cb);
        break;

    case TIME_MIDI:
        printf("Midi Song Position Pointer = %u\n", time.midi.songptrpos);
        break;

    case TIME_MS:
        printf("Millisecond Time = %u\n", time.ms);
        break;

    case TIME_SAMPLES:
        printf("%u digital audio samples have been played so far.\n", time.sample);
        break;

    case TIME_SMPTE:
        printf("SMPTE Time (%u fps) = %u:%u:%u:%u\n", time.smpte.fps, time.smpte.
hour, time.smpte.min, time.smpte.sec, time.smpte.frame);
        break;

    case TIME_TICKS:
        printf("PPQN Clocks = %u\n", time.ticks);
}
```

Outputting MIDI data immediately while a Stream is playing

Note that you can call midiOutShortMsg() or midiOutLongMsg() while the Stream (ie, queued MIDI events) is playing. In such a case, the MIDI data you send via these two functions will be output as soon as possible. (But don't utilize running status with midiOutShortMsg. Always specify the full MIDI message with Status).

Stream recording

Currently, the Stream API does not support recording. If you need to do MIDI recording, you'll have to use the Low level MIDI API and a MultiMedia Timer instead of the Stream API.

This information is not yet completed.

In order to understand how the Mixer API works, it's important to first understand the hardware layout of a typical audio card. It's necessary to be able to visualize an audio card as having distinct, but interconnected, components upon it.

Let's consider a typical, basic audio card. First of all, if the audio card is capable of recording digital audio, then it typically has a microphone input jack (with some sort of pre-amp), and it also has an analog-to-digital converter (ADC) to convert that analog microphone signal to a digital stream. Therefore, it has two components -- the Microphone input component, and the ADC component. The Microphone input is piped into the ADC. So, we can represent the layout with the following block diagram showing two components, with the signal flow between them (ie, the arrow)
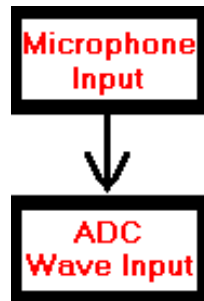
A typical audio card is also capable of playing back digital audio, so it has a DAC to convert the digital stream back to an analog signal, and also it has a speaker output jack (ie, with some sort of analog amplifier). Therefore, it has two more components -- the DAC component, and the Speaker component. The DAC output is piped to the speakers.
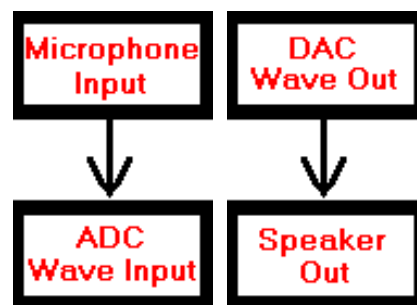
A typical audio card may have some other components. For example, it may have a built-in sound module (ie, synth) capable of playing MIDI data. The audio output of this component would typically be piped to the speaker output just like the DAC. So, our block diagram now looks like so:
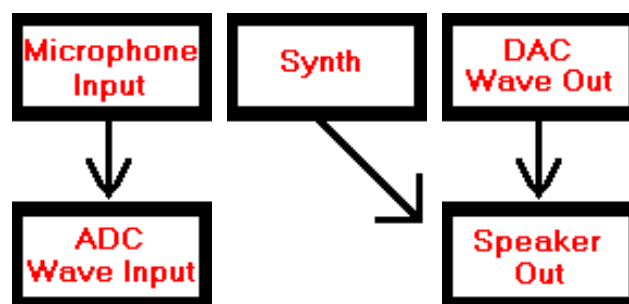
So too, a typical audio card has a connector to (internally) attach the audio out of the computer's CDROM drive (so that an audio CD played in that CDROM drive will sound through the computer's speakers). This component is also piped to the speaker output, just like the Synth and DAC. Now, our block diagram looks like this:
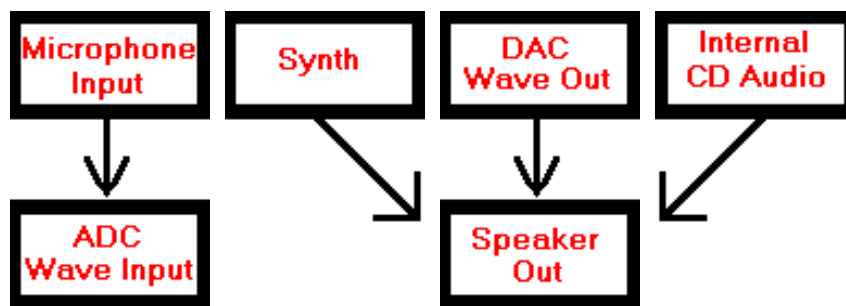
Finally, let's assume that this audio card has a Line In component so that the audio signal from an external tape deck or musical instrument or external hardware mixer can be attached to this jack and digitized. This component is piped to the ADC component just like the Microphone Input component. Here is our finished block diagram which contains 7 components (and 5 signal flows -- the arrows that interconnect them):
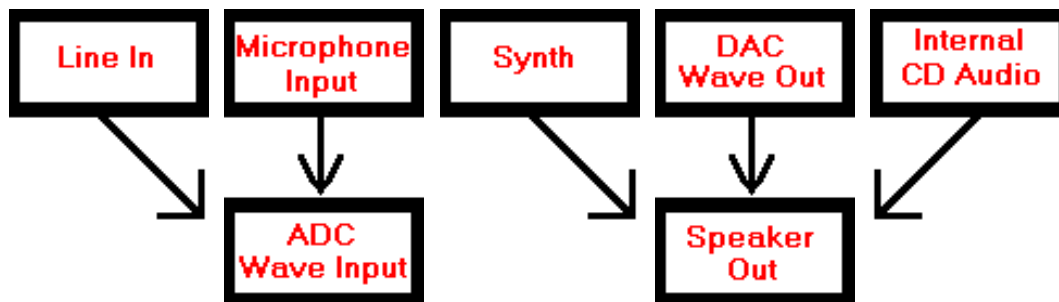
Typically, each one of these components has its own, individual parameters. For example, the Synth will usually have its own volume (gain) level. The Internal CD Audio will have its own volume level. And the DAC (ie, digital audio, or WAVE playback) will have its own volume level. In this way, if the user is playing an audio CD, playing a MIDI file, and playing back a WAVE file, simultaneously, he can balance out the volumes of all 3 components as they are being output to the speaker jack. So too, the speaker component will typically have its own volume -- a Master volume that affects the overall mix of the 3 components piped to the speaker out.

Likewise, the Line In and Microphone Input typically have separate volume levels so they can be balanced when recording simultaneously from both jacks. And the ADC may have some sort of Master Volume which affects the overall recording level of the 2 components piped to it.

A given component may have other parameters that are controllable. For example, each of the above components may have its own Mute switch so that the component's sound can be quickly turned on/off.

The Mixer device

A given audio card has one Mixer device associated with it. All of the various components on that card are controlled through that card's one Mixer device. The Windows Mixer API is used to access the card's Mixer device. The Mixer API has functions to get a listing of all the various components on a particular card, and to adjust all of their parameters. This is a new API added to Win95/98 and WinNT (4.X and above), although an add-on to Windows 3.1 makes it available to that older OS.

NOTE: A card's device driver needs extra support to work with the Mixer API. Not all Win95 and WinNT drivers have this support. Win3.1 drivers typically do not.

In any computer, there can be more than one installed audio card. You already discovered that Windows maintains lists of all of the WAVE and MIDI input/output devices in a system. Since each installed audio card has its own Mixer device too (as long as its driver supports such), Windows also maintains a list of Mixer devices installed in a system. So for example, if you have two audio cards installed in a given system, then there should be two Mixer devices installed as well (assuming that the drivers for both audio cards support the mixer API).

Just like with the WAVE and MIDI input/output devices, Windows assigns a numeric ID to each Mixer device. So, the Mixer

device with an ID of 0 is the first (default) mixer in the system. If there is a second audio card, then there should be a second Mixer device with an ID of 1.

Just like with other devices, in order to use a Mixer device, you must first open it (with mixerOpen()), and then you may call other Mixer APIs to control the card's line inputs/outputs. When finished, you must close the device (with mixerClose())

Opening a Mixer device

How does your program choose a mixer device to manipulate? There are several different approaches you can take, depending upon how fancy and flexible you want your program to be.

If you simply want to open the preferred Mixer device, then use a Device ID of 0 with mixerOpen() as so:

```
unsigned long err;
HMIXER        mixerHandle;

/* Open the mixer associated with the default
   Audio/MIDI card in the computer */
err = mixerOpen(&mixerHandle, 0, 0, 0, 0);
if (err)
{
    printf("ERROR: Can't open Mixer Device! -- %08X\n", err);
}
else
{
    /* The mixer device is now open, and its
       handle is in the variable 'mixerHandle'.
       You may now use it with other Mixer API
       functions */
}
```

Of course, if the user has no Mixer device installed, the above call returns an error, so always check that return value. (The expected error numbers from the Mixer API's are listed in MMSYSTEM.H. Unfortunately, unlike with the Wave and Midi low level API's, there is no API function to translate these error numbers into strings to present to the user).

So what actually is the preferred Mixer device? Well, that's whatever Mixer device happened to have been installed first in a system. If there is only one audio card in the system, then it's a good bet that you have the mixer device you want. But, what if you're trying to use the Wave Output on a second audio card? You definitely don't want to be using the Mixer device for the first card to control the second card's Wave Out volume. (The first card's Mixer doesn't control the second card's Wave Output).

So, how do you open the Mixer for the desired card? Fortunately, mixerOpen() allows you to pass the device ID or the open handle of some other device associated with desired card. In that case, mixerOpen() will ensure that it returns the Mixer device associated with that card's other device. So for example, here's how you would open the default WAVE OUT device (ie, the WAVE OUT on the default card), and then get the handle to that card's Mixer device:

```
unsigned long err;
HMIXER        mixerHandle;
WAVEFORMATEX  waveFormat;
HWAVEOUT      hWaveOut;

/* Open the default WAVE Out Device, specifying my callback.
   Assume that waveFormat has already been initialized as desired */
err = waveOutOpen(&hWaveOut, WAVE_MAPPER, &waveFormat, (DWORD)WaveOutProc, 0,
CALLBACK_FUNCTION);
if (err)
```

```
{
    printf("ERROR: Can't open WAVE Out Device! -- %08X\n", err);
}
else
{
    /* Open the mixer associated with the WAVE OUT device
       opened above. Note that I pass the handle obtained
       via waveOutOpen() above */
    err = mixerOpen(&mixerHandle, hWaveOut, 0, 0, MIXER_OBJECTF_HWAVEOUT);
    if (err)
    {
        printf("ERROR: Can't open Mixer Device! -- %08X\n", err);
    }
}
```

The key above is in passing not only the handle that you obtain with waveOutOpen() (or waveInOpen(), or midiOutOpen(), or midiInOpen()) but also the last parameter of mixerOpen() must be MIXER_OBJECTF_HWAVEOUT (or MIXER_OBJECTF_HWAVEIN, or MIXER_OBJECTF_HMIDIOUT, or MIXER_OBJECTF_HMIDIIN) to indicate what kind of device handle is being passed to mixerOpen()

Alternately, if you know the device ID number of the desired WAVE OUT device (but haven't yet opened its handle via waveOutOpen()), you can pass that device ID to mixerOpen() and instead specify MIXER_OBJECTF_WAVEOUT. mixerOpen() will find the Mixer that corresponds to that WAVE Out device with the specified ID.

If desired, you can then get the above Mixer's ID (number) from its handle, using mixerGetID() as so:

```
unsigned long mixerID;

err = mixerGetID(mixerHandle, &mixerID, MIXER_OBJECTF_HMIXER);
if (err)
{
    printf("ERROR: Can't get Mixer Device ID! -- %08X\n", err);
}
else
{
    printf("Mixer Device ID = %d\n", mixerID);
}
```

Listing all of the Mixer devices

If you're writing an application where you need to list all of the Mixer devices in the system, Windows has a function that you can call to determine how many Mixer devices are in the list. This function is called mixerGetNumDevs(). This returns the number of Mixer devices in the list. Remember that the Device IDs start with 0 and increment. So if Windows says that there are 3 devices in the list, then you know that their Device IDs are 0, 1, and 2 respectively. You then use these Device IDs with other Windows functions. For example, there is a function you can call to get information about one of the devices in the list, such as its name, and what sort of other features it has such as how many components it contains, and what type of component each one is. You pass the Device ID of the Mixer device which you want to get information about (as well as a pointer to a special structure called a MIXERCAPS into which Windows puts the info about the device), The name of the function to get information about a particular Mixer device is mixerGetDevCaps().

Here then is an example of going through the list of Mixer devices, and printing the name of each one:

```
MIXERCAPS      mixcaps;
unsigned long iNumDevs, i;

/* Get the number of Mixer devices in this computer */
```

```
iNumDevs = mixerGetNumDevs();

/* Go through all of those devices, displaying their IDs/names */
for (i = 0; i < iNumDevs; i++)
{
    /* Get info about the next device */
    if (!mixerGetDevCaps(i, &mixcaps, sizeof(MIXERCAPS)))
    {
        /* Display its ID number and name */
        printf("Device ID #%u: %s\r\n", i, mixcaps.szPname);
    }
}
```

About Lines and Controls

Up to this point, I have been using the term "component" to refer to a hardware section that has its own individual, adjustable parameters. The Mixer API actually concerns itself with signal flows. (ie, In our block diagram, the signal flows are the 5 arrows which connect the components). The Microsoft documention refers to each signal flow (ie, each arrow in our block diagram) as a "source line". So, a Mixer device controls "source lines" -- not components per se. It is each "source line" which has its own individual, adjustable parameters -- not a component per se. Our example audio card has 5 source lines as far as a Mixer is concerned. From here on, whenever you see the word "source line", think of the signal flow between components

I have also been using the term "parameters" to refer to settings that are adjustable on a "line", for example a volume level or a mute switch or a bass boast or a pan setting, etc. The Microsoft documentation refers to these parameters as "controls" (which can be a confusing term since a programmer typically thinks of a control as a graphical window -- part of his graphical interface. But in the case of Mixers, we mean "audio controls").

So for example, it's not really the "Microphone Input" component itself which concerns the Mixer API. Rather, it's the signal flow between the "Microphone Input" and "ADC" components. The "Microphone Input" volume control adjusts the volume of this signal flowing between the Microphone Input and ADC components.

For further illustration of the difference between components and source lines, let's add another feature to our example sound card. Sometimes, you'd like the Microphone Input to not only pipe through the ADC (so that you can record it), but also to pipe through the speakers (so that you can monitor the signal that you're actually recording, and hear what it sounds like through the speakers). So, let's adjust our block diagram to show the signal from the "Microphone Input" as going to both the ADC and the speakers.

.
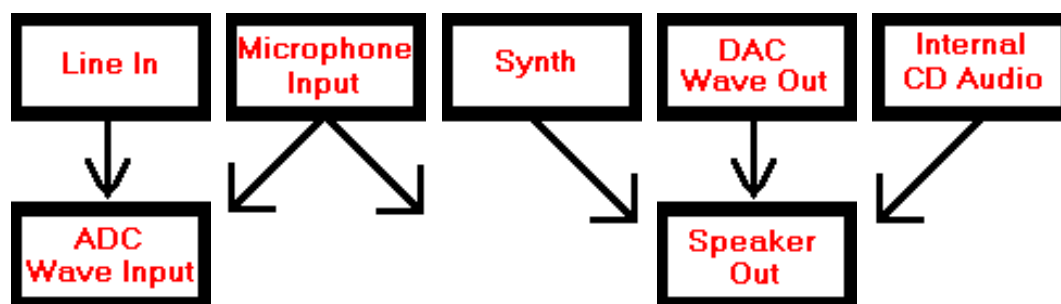
Notice that we now have 6 "source lines" (arrows). And there are two source lines from the "Microphone Input", even though it is only one component on our card. Each source line has its own settings. For example, there is a volume control for the "Microphone Input" source line going into the ADC (so that you can set the recording level for your WAVE recording software). There is another, separate volume control for the "Microphone Input" source line going to the speakers (so that you can set the monitor level of the microphone, separate from the recording level). These source lines may each have their own mute switches (so that you can, for example, have the "Microphone Input" signal going to only the ADC, but not the Speakers). They may have other controls as well, and each line's controls will be separate from the other line's controls.

There is another important concept to discuss in regard to lines. There are such things as "destination lines", and we differentiate between source lines and destination lines. What are those? Well, a destination line has other lines -- source lines -- flowing into it. In our block diagram, the "Speaker Out" is a destination line, since it has signals from the "Internal CD Audio", "Synth", "DAC Wave Out" and "Microphone Input" flowing into it. And the latter 4 are source lines. They are source lines to the "Speaker Out" destination line since their signals flow into the "Speaker Out" line. Source lines always have to flow into some destination line. (ie, A source line cannot exist without being attached to one destination line).

So, whereas source lines are the arrows in our block diagram, destination lines are actual components in our block diagram.

So too, the "ADC Wave Input" is a destination line in our example card. It has 2 source lines flowing into it -- the "Microphone Input" and "Line In". Those two lines are not source lines to the "Speaker Out" since they do not flow into it. Similiarly, the 4 source lines flowing into the "Speaker Out" are not source lines to the "ADC Wave Input" destination line, since they don't flow into the latter.

Like source lines, destination lines can have controls. And each destination line's controls are separate from the other lines' controls. For example, the "Speaker Out" may have a volume control. This would function as a master volume for all 4 source lines going to the Speaker Out destination line. (And each of those 4 source lines has its own individual volume level control).

NOTE: Although a card may have a stereo component (for example, typically the "Speaker Out" upon most cards is stereo), this is considered one line. It has 2 channels, but it nevertheless is one line on the Mixer device. Indeed, a component could have more than 2 channels, and still be considered only one line. In conclusion, lines are not the same thing as channels. In some ways, it may be helpful to think of a line like a MIDI cord. One MIDI cord connects two components, but there can be several channels going over that one cord. The same thing is true of a line.

In conclusion, our example card has 6 source lines and 2 destination lines. The 4 source lines labeled "Internal CD Audio", "Synth", "DAC Wave Out" and "Microphone Input" are attached to the "Speaker Out" destination line. The 2 source lines labeled "Microphone Input" and "Line In" are attached to the "ADC Wave Input" destination line.

Line IDs and Types

Every line in a mixer must have a unique ID number. Every line also has a type. This is just a numeric value that describes what type of line it is. These are defined in the MMSYSTEM.H include file. For example, a line may have a type of MIXERLINE_COMPONENTTYPE_SRC_SYNTHESIZER to indicate that it is a signal from a built-in sound module.

The allowable types for source lines are as so:

MIXERLINE_COMPONENTTYPE_SRC_DIGITAL	A digital source, for example, a SPDIF input jack.
MIXERLINE_COMPONENTTYPE_SRC_LINE	A line input source. Typically used for a line input jack, if there is a separate microphone input (ie, MIXERLINE_COMPONENTTYPE_SRC_MICROPHONE).
MIXERLINE_COMPONENTTYPE_SRC_MICROPHONE	Microphone input (but also used for a combination of Mic/ Line input if there isn't a separate line input source).
MIXERLINE_COMPONENTTYPE_SRC_SYNTHESIZER	Musical synth. Typically used for a card that contains a synth capable of playing MIDI. This would be the audio out of that built-in synth.
MIXERLINE_COMPONENTTYPE_SRC_COMPACTDISC	The audio feed from an internal CDROM drive (connected to the sound card).
MIXERLINE_COMPONENTTYPE_SRC_TELEPHONE	Typically used for a telephone line's incoming audio to be piped through the computer's speakers, or the telephone line in jack for a built-in modem.

MIXERLINE_COMPONENTTYPE_SRC_PCSPEAKER	Typically, to allow sound, that normally goes to the computer's built-in speaker, to instead be routed through the card's speaker output. The motherboard's system speaker connector would be internally connected to some connector on the sound card for this purpose.
MIXERLINE_COMPONENTTYPE_SRC_WAVEOUT	Wave playback (ie, this is the card's DAC).
MIXERLINE_COMPONENTTYPE_SRC_AUXILIARY	An aux jack meant to be routed to the Speaker Out, or to the ADC (for WAVE recording). Typically, this is used to connect external, analog equipment (such as tape decks, the audio outputs of musical instruments, etc) for digitalizing or playback through the sound card.
MIXERLINE_COMPONENTTYPE_SRC_ANALOG	May be used similiarly to MIXERLINE_COMPONENTTYPE_SRC_AUXILIARY (although I have seen some mixers use this like MIXERLINE_COMPONENTTYPE_SRC_PCSPEAKER). In general, this would be some analog connector on the sound card which is only accessible internally, to be used to internally connect some analog component inside of the computer case so that it plays through the speaker out.
MIXERLINE_COMPONENTTYPE_SRC_UNDEFINED	Undefined type of source. If none of the others above are applicable.

The allowable types for destination lines are as so:

MIXERLINE_COMPONENTTYPE_DST_DIGITAL	A digital destination, for example, a SPDIF output jack.
MIXERLINE_COMPONENTTYPE_DST_LINE	A line output destination. Typically used for a line output jack, if there is a separate speaker output (ie, MIXERLINE_COMPONENTTYPE_DST_SPEAKERS).
MIXERLINE_COMPONENTTYPE_DST_MONITOR	Typically a "Monitor Out" jack to be used for a speaker system separate from the main speaker out. Or, it could be some built-in monitor speaker on the sound card itself, such as a speaker for a built-in modem.
MIXERLINE_COMPONENTTYPE_DST_SPEAKERS	The audio output to a pair of speakers (ie, the "Speaker Out" jack).
MIXERLINE_COMPONENTTYPE_DST_HEADPHONES	Typically, a headphone output jack.
MIXERLINE_COMPONENTTYPE_DST_TELEPHONE	Typically used to daisy-chain a telephone to an analog modem's "telephone out" jack.
MIXERLINE_COMPONENTTYPE_DST_WAVEIN	The card's ADC (to digitize analog sources, for example, in recording WAVE files of such).
MIXERLINE_COMPONENTTYPE_DST_VOICEIN	May be some sort of hardware used for voice recognition. Typically, a microphone source line would be attached to this.
MIXERLINE_COMPONENTTYPE_DST_UNDEFINED	Undefined type of destination. If none of the others above are applicable.

Control IDs and Types

Each line can have one or more adjustable "audio controls". (But, it's possible that a line could have no controls at all). For example, the Synth line may have a volume fader and a mute switch. Each control has a type. These are defined in the MMSYSTEM.H include file. For example, the volume fader would have a type of MIXERCONTROL_CONTROLTYPE_VOLUME. The Mute switch would have a type of MIXERCONTROL_CONTROLTYPE_MUTE.

Every control also has a unique ID number. No two controls may have the same ID number, even if they belong to 2 different

lines.

The control types are divided up into classes. These classes are roughly based upon what type of value a control adjusts, and therefore what kind of graphical user interface you would normally present to the enduser to let him adjust that control's value. For example, you would normally present a graphical fader to allow the user to adjust a control of the type MIXERCONTROL_CONTROLTYPE_VOLUME. On the other hand, you'd typically use a graphical (checkmark) button to allow him to adjust a control of type MIXERCONTROL_CONTROLTYPE_MUTE (since that control has only two possible values, or states).

The allowable classes for controls are as so:

MIXERCONTROL_CT_CLASS_FADER	A control that is adjusted by a vertical fader, with a linear scale of positive values (ie, 0 is the lowest possible value). A MIXERCONTROLDETAILS_UNSIGNED structure is used to retrieve or set the control's value.
MIXERCONTROL_CT_CLASS_LIST	A control that is adjusted by a listbox containing numerous "values" to be selected. The user will single-select, or perhaps multiple-select if desired, his choice of value(s). A MIXERCONTROLDETAILS_BOOLEAN structure is used to retrieve or set the control's value. A MIXERCONTROLDETAILS_LISTTEXT structure is also used to retrieve the text description of each item of this control.
MIXERCONTROL_CT_CLASS_METER	A control that is adjusted by a graphical meter. A MIXERCONTROLDETAILS_BOOLEAN, MIXERCONTROLDETAILS_SIGNED, or MIXERCONTROLDETAILS_UNSIGNED structure is used to retrieve or set the control's value.
MIXERCONTROL_CT_CLASS_NUMBER	A control that is adjusted by numeric entry. The user enters a signed integer, unsigned integer, or integer decibel value. A MIXERCONTROLDETAILS_SIGNED or MIXERCONTROLDETAILS_UNSIGNED structure is used to retrieve or set the control's value.
MIXERCONTROL_CT_CLASS_SLIDER	A control that is adjusted by a horizontal slider with a linear scale of negative and positive values. (ie, Generally, 0 is the mid or "neutral" point). A MIXERCONTROLDETAILS_SIGNED structure is used to retrieve or set the control's value.
MIXERCONTROL_CT_CLASS_SWITCH	A control that is has only two states (ie, values), and is therefore adjusted via a button. A MIXERCONTROLDETAILS_BOOLEAN structure is used to retrieve or set the control's value.
MIXERCONTROL_CT_CLASS_TIME	A control that allows the user to enter a time value, such as Reverb Decay Time. It is a positive, integer value.
MIXERCONTROL_CT_CLASS_CUSTOM	A custom class of control. If none of the others above are applicable.

Each class has certain types associated with it. For example, the MIXERCONTROL_CT_CLASS_FADER class has the following 5 types associated with it:

MIXERCONTROL_CONTROLTYPE_VOLUME	Volume fader. The range of allowable values is 0 through 65,535.
MIXERCONTROL_CONTROLTYPE_BASS	Bass boost fader. The range of allowable values is 0 through 65,535.

MIXERCONTROL_CONTROLTYPE_TREBLE	Treble boost fader. The range of allowable values is 0 through 65,535.
MIXERCONTROL_CONTROLTYPE_EQUALIZER	A graphic EQ. The range of allowable values for each band is 0 through 65,535. A MIXERCONTROLDETAILS_LISTTEXT structure is used to retrieve the text label for each band of the EQ. Typically, this control will also have its MIXERCONTROL_CONTROLF_MULTIPLE flag set, since the EQ will likely have numerous bands.
MIXERCONTROL_CONTROLTYPE_FADER	A generic fader, to be used when none of the above are applicable. The range of allowable values is 0 through 65,535.

In fact, if you look at MMSYSTEM.H, you'll note that a control with a type of MIXERCONTROL_CONTROLTYPE_VOLUME is defined as MIXERCONTROL_CT_CLASS_FADER | MIXERCONTROL_CT_UNITS_UNSIGNED + 1. The class is actually contained in the top 4 bits of the type. So if you know a control's type, then you determine its class by masking off the top 4 bits. For example, assume that you've queried a control's type, and stored the value returned by the Mixer API in your variable named "type". Here's how you'd figure out it's class:

```c
unsigned long   type;

/* Figure out the class based upon the top 4 bits of its type */
switch (MIXERCONTROL_CT_CLASS_MASK & type)
{
    case MIXERCONTROL_CT_CLASS_FADER:
    {
        printf("It's a fader class.");
        break;
    }
    case MIXERCONTROL_CT_CLASS_LIST:
    {
        printf("It's a list class.");
        break;
    }
    case MIXERCONTROL_CT_CLASS_METER:
    {
        printf("It's a meter class.");
        break;
    }
    case MIXERCONTROL_CT_CLASS_NUMBER:
    {
        printf("It's a number class.");
        break;
    }
    case MIXERCONTROL_CT_CLASS_SLIDER:
    {
        printf("It's a slider class.");
        break;
    }
    case MIXERCONTROL_CT_CLASS_TIME:
    {
        printf("It's a time class.");
        break;
    }
    case MIXERCONTROL_CT_CLASS_CUSTOM:
    {
        printf("It's a custom class.");
```

```
        break;
    }
}
```

The MIXERCONTROL_CT_CLASS_SWITCH class has the following 7 types associated with it:

MIXERCONTROL_CONTROLTYPE_BOOLEAN	A control with a boolean value. The value is an integer that is either 0 (FALSE) or non-zero (TRUE).
MIXERCONTROL_CONTROLTYPE_BUTTON	A control whose value is 1 when the button is pressed (ie, some feature/action is enabled), or 0 if not pressed (ie, no action is taken). For example, this type of control may be used by a talkback button or pedal sustain -- the action/feature is on only while the button is pressed, and otherwise is not applicable.
MIXERCONTROL_CONTROLTYPE_LOUDNESS	A control whose value is 1 to boost bass frequencies, or 0 for normal (ie, no boost). (A MIXERCONTROL_CONTROLTYPE_BASS fader control may be used to set the actual amount of boost, in conjunction with this control turning the boost on/off)
MIXERCONTROL_CONTROLTYPE_MONO	A control whose value is 1 for mono operation (ie, all channels are summed into one), or 0 for normal (ie, stereo or multi-channel).
MIXERCONTROL_CONTROLTYPE_MUTE	A control whose value is 1 to mute some feature, or 0 for normal (ie, no mute).
MIXERCONTROL_CONTROLTYPE_ONOFF	A control whose value is 1 to enable some feature/action, or 0 to disable that feature/action. The difference between this and MIXERCONTROL_CONTROLTYPE_BUTTON is that the latter's 0 value doesn't disable the feature/action per se, but rather, simply represents a "not applicable" state for the feature/action. MIXERCONTROL_CONTROLTYPE_ONOFF would be similiar to a real on/off switch (ie, a checkmark button in Windows parlance) whereas MIXERCONTROL_CONTROLTYPE_BUTTON would be more akin to a "momentary switch" (ie, a pushbutton). The difference between MIXERCONTROL_CONTROLTYPE_ONOFF and MIXERCONTROL_CONTROLTYPE_BOOLEAN is merely the labeling/grahics that would be shown on the user interface. The former is "ON" when its value is 1. The latter is "TRUE" when its value is 1. So typically, the two buttons would be represented by different labels and/or graphics to reflect that difference in semantics.
MIXERCONTROL_CONTROLTYPE_STEREOENH	A control whose value is 1 to enable a stereo enhance feature (ie, increase stereo separation), or 0 for normal (ie, no enhance).

The MIXERCONTROL_CT_CLASS_LIST class has the following 4 types associated with it:

MIXERCONTROL_CONTROLTYPE_SINGLESELECT	Allows one selection out of a choice of many selections. For example, this can be used to select a reverb type out of choice of many types (ie, Hall, Plate, Room, etc).

MIXERCONTROL_CONTROLTYPE_MULTIPLESELECT	Like MIXERCONTROL_CONTROLTYPE_SINGLESELECT, but allows the selection of more than one item simultaneously.
MIXERCONTROL_CONTROLTYPE_MUX	Allows the selection of one audio line out of several choices of audio lines. For example, to allow a source line to be routed to one of several possible destination lines -- this control could list all of the possible choices of destination lines, and allow one to be chosen.
MIXERCONTROL_CONTROLTYPE_MIXER	Like MIXERCONTROL_CONTROLTYPE_MUX, but allows the selection of more than one audio line simultaneously. For example, this control could be for a reverb component which allows several source lines to be simultaneously routed to it, and this control determines which source lines are selected for routing to the reverb.

The MIXERCONTROL_CT_CLASS_METER class has the following 4 types associated with it:

MIXERCONTROL_CONTROLTYPE_BOOLEANMETER	A meter whose integer value is either 0 (ie, FALSE) or non-zero (TRUE). A MIXERCONTROLDETAILS_BOOLEAN struct is used to set/retrieve its value.
MIXERCONTROL_CONTROLTYPE_PEAKMETER	A control with a value that is an integer whose allowable, maximum range is -32,768 (lowest) through 32,767 (highest). In other words, its value is a SHORT. A MIXERCONTROLDETAILS_SIGNED struct is used to set/retrieve its value.
MIXERCONTROL_CONTROLTYPE_SIGNEDMETER	A control with a value that is an integer whose allowable, maximum range is -2,147,483,648 (lowest) through 2,147,483,647 (highest) inclusive. In other words, its value is a LONG. A MIXERCONTROLDETAILS_SIGNED struct is used to set/retrieve its value.
MIXERCONTROL_CONTROLTYPE_UNSIGNEDMETER	Like MIXERCONTROL_CONTROLTYPE_SIGNEDMETER, but its allowable, maximum value range is from 0 (lowest) to 4,294,967,295. In other words, its value is a ULONG. A MIXERCONTROLDETAILS_UNSIGNED struct is used to set/retrieve its value.

Some of the allowable types for the MIXERCONTROL_CT_CLASS_NUMBER class are similiar to the MIXERCONTROL_CT_CLASS_METER class types. But with the MIXERCONTROL_CT_CLASS_NUMBER class, you would typically use an Edit control for the graphical user interface to allow him to enter a value. The MIXERCONTROL_CT_CLASS_METER would typically use some sort of graphical display similiar to an audio meter. MIXERCONTROL_CT_CLASS_NUMBER class has the following 4 types associated with it:

MIXERCONTROL_CONTROLTYPE_SIGNED	A control with a value that is an integer whose allowable, maximum range is -2,147,483,648 (lowest) through 2,147,483,647 (highest) inclusive. In other words, its value is a LONG. A MIXERCONTROLDETAILS_SIGNED struct is used to set/retrieve its value.
MIXERCONTROL_CONTROLTYPE_UNSIGNED	Like MIXERCONTROL_CONTROLTYPE_SIGNEDMETER, but its allowable, maximum value range is from 0 (lowest) to 4,294,967,295. In other words, its value is a ULONG. A MIXERCONTROLDETAILS_UNSIGNED struct is used to set/retrieve its value.

MIXERCONTROL_CONTROLTYPE_PERCENT	A control whose integer value is a percent. A MIXERCONTROLDETAILS_UNSIGNED struct is used to set/retrieve its value.
MIXERCONTROL_CONTROLTYPE_DECIBELS	A control with a value that is an integer whose allowable, maximum range is -32,768 (lowest) through 32,767 (highest). In other words, its value is a SHORT. Each increment is a tenth of a decibel. A MIXERCONTROLDETAILS_SIGNED struct is used to set/retrieve its value.

The MIXERCONTROL_CT_CLASS_SLIDER class has the following 3 types associated with it:

MIXERCONTROL_CONTROLTYPE_SLIDER	A slider with a value that is an integer whose allowable, maximum range is -32,768 (lowest) through 32,767 (highest). In other words, its value is a SHORT.
MIXERCONTROL_CONTROLTYPE_PAN	A slider with a value that is an integer whose allowable, maximum range is -32,768 (far left) through 32,767 (far right). In other words, its value is a SHORT. It represents pan position in the stereo spectrum, where 0 is center position.
MIXERCONTROL_CONTROLTYPE_QSOUNDPAN	A slider with a value that is an integer whose allowable, maximum range is -15 (lowest) through 15 (highest). In other words, its value is a SHORT. It represents Qsound's expanded sound setting.

The MIXERCONTROL_CT_CLASS_TIME class has the following 2 types associated with it:

MIXERCONTROL_CONTROLTYPE_MICROTIME	A control with a value that is an integer whose allowable, maximum range is 0 (lowest) through 4,294,967,295. In other words, its value is a ULONG. Its value represents an amount of time in microseconds.
MIXERCONTROL_CONTROLTYPE_MILLITIME	A control with a value that is an integer whose allowable, maximum range is 0 (lowest) through 4,294,967,295. In other words, its value is a ULONG. Its value represents an amount of time in milliseconds.

The MIXERCONTROL_CT_CLASS_CUSTOM class is a proprietary class. A Mixer that uses such a type of control could expect only an application that is written specifically for that Mixer to understand what types of controls are in this class, and what structures to use to set/retrieve their values. (Perhaps even proprietary structures could be used)

MIXERLINE structures, and enumerating lines

One way to get information about a mixer's lines, if you don't know what particular lines it has (ie, you don't know the types of lines it has, nor know their ID numbers), is to first call mixerGetDevCaps() to fetch information about that Mixer device into a MIXERCAPS structure. Using information in this structure, you can determine how many destination lines are on the card. From there, you can enumerate (ie, fetch information about) each destination line, and the source lines to each destination line. And after you enumerate the lines, you can enumerate the controls for each line.

Let's examine such an approach and study the structures that are associated with the Mixer API.

In order to better understand the Mixer API, we should take a peek inside of the Mixer Device for our example audio card, and examine its internal structures which are used with the Mixer API. We'll assume that this Mixer Device is written in C, and therefore uses C structures.

As noted previously, the Mixer API mixerGetDevCaps() is used to fetch information about a Mixer Device. It fills in a MIXERCAPS structure. In particular, the *cDestinations* field of the MIXERCAPS tells you how many destination lines are on

the card. It doesn't tell you how many total lines (ie, destination and source lines) there are. It counts only destination lines. As you'll recall, our example audio card has 2 destination lines. We'll fill in the other fields, such as the Mixer name and product ID with arbitrary values for the purposes of this tutorial. Assume that this mixer is the first installed mixer in the system (ie, ID = 0). Here then is our Mixer Device's MIXERCAPS structure (which is defined in MMSYSTEM.H):

```c
MIXERCAPS mixercaps = {
    0,        /* manufacturer id */
    0,        /* product id */
    0x0100,   /* driver version #. Note that the high 8-bits
               are the version, and low 8-bits are revision. So,
               our driver version is 1.0 */
    "Example Sound Card", /* product name */
    0,        /* Support bits. None are currently defined */
    2,        /* # of destination lines */
};
```

Here is an example of passing a MIXERCAPS to mixerGetDevCaps() and letting Windows fill it in with the above values. (Assume that we have already opened the mixer and stored its handle into our variable "mixerHandle").

```c
MIXERCAPS       mixcaps;
MMRESULT        err;

/* Get info about the first Mixer Device */
if (!(err = mixerGetDevCaps((UINT)mixerHandle, &mixcaps, sizeof(MIXERCAPS))))
{
    /* Success. We can continue on with our tutorial below */
}
else
{
    /* An error */
    printf("Error #%d calling mixerGetDevCaps()\n", err);
}
```

Information about a line is contained in a MIXERLINE structure (as defined in MMSYSTEM.H). Let's assume that our "Speaker Out" destination line has two controls -- a volume slider (to control the overall mix to the speakers) and a mute switch (to mute the overall mix). Here's the MIXERLINE structure for our "Speaker Out" destination:

```c
MIXERLINE mixerline_SpkrOut = {
    sizeof(MIXERLINE),               /* size of MIXERLINE structure */
    0,                               /* zero based index of destination line */
    0,                               /* zero based source index (used only if
                                        this is a source line) */
    0xFFFF0000,                      /* unique ID # for this line */
    MIXERLINE_LINEF_ACTIVE,          /* state/information about line */
    0,                               /* driver specific information */
    MIXERLINE_COMPONENTTYPE_DST_SPEAKERS,  /* component type */
    2,                               /* # of channels this line supports */
    4,                               /* # of source lines connected (used
                                        only if this is a destination line) */
    2,                               /* # of controls in this line */
    "Spkr Out",                      /* Short name for this line */
    "Speaker Out",                   /* Long name for this line */
    MIXERLINE_TARGETTYPE_WAVEOUT,    /* MIXERLINE_TARGETTYPE_xxxx */
    /* The following info is just some info about the Mixer Device for this
    line, prepended to the above info. It's mostly just for reference. */
    0,        /* device ID of our Mixer */
    0,        /* manufacturer id */
```

```
    0,        /* product id */
    0x0100,   /* driver version # */
    "Example Sound Card", /* product name */
};
```

There are a few things to note here. First, notice that the *dwComponentType* for the "Speaker Out" destination line is one of the types for a destination line -- appropriately, MIXERLINE_COMPONENTTYPE_DST_SPEAKERS to indicate that it's a speaker output. I've chosen a value of 0xFFFF0000 for the *dwLineID*. The Mixer Device programmer may chose any value he wishes for this field, but no other line in this mixer may have the same value for its *dwLineID* (as you'll notice later). Also, note that since our speaker output is stereo, the *cChannels* field is 2. As you'll recall, there are 4 source lines connected to our "Speaker Out" destination line, so the *cConnections* field is 4. The name fields are nul-terminated strings. They can be anything that the Mixer Device programmer chooses, but the shorter name is meant to be used as a label for any tightly spaced graphical controls. When the MIXERLINE_LINEF_ACTIVE bit is set in the *fdwLine* field, this simply means that the line has not been disabled (such as what may happen if it were muted).

The *dwDestination* field is an index value based from 0. The first destination line in a mixer will have an index value of 0 (as in our example above). The second destination line will have an index of 1. The third destination line will have an index of 2. Etc. This is the same concept as how Windows enumerates Mixer Devices (ie, where the first installed Mixer Device has an ID of 0). The index value is not necessarily the same as a line's ID number (as you can see from the example above). What is the purpose of an index, and why do we need both an index and an ID number? As you may guess, the index number is primarily used when you need to enumerate what lines a Mixer has. Until you enumerate the lines (ie, fetch information about each line), you don't know the ID numbers of any of the lines. So, you need to use indexes with the mixer API to enumerate lines. But once you have retrieved info about a line, and therefore know its ID number, then you can alter its settings more directly. So, index numbers are primarily useful for initially enumerating lines and controls to find their ID numbers and types. And then the ID numbers are primarily useful for subsequently performing direct manipulation of the lines and controls.

Now let's take a look at the MIXERLINE structure for our "ADC Wave Input" destination line. Let's assume that it also has two controls -- a volume slider (to control the overall mix to the ADC) and a mute switch (to mute the overall mix). Here's the MIXERLINE structure for our "ADC Wave Input" destination:

```
MIXERLINE mixerline_WaveIn = {
    sizeof(MIXERLINE),                /* size of MIXERLINE structure */
    1,                                /* zero based index of destination line */
    0,                                /* zero based source index (used only if
                                         this is a source line) */
    0xFFFF0001,                       /* unique ID # for this line */
    MIXERLINE_LINEF_ACTIVE,           /* state/information about line */
    0,                                /* driver specific information */
    MIXERLINE_COMPONENTTYPE_DST_WAVEIN,  /* component type */
    2,                                /* # of channels this line supports */
    2,                                /* # of source lines connected (used
                                         only if this is a destination line) */
    2,                                /* # of controls in this line */
    "Wave In",                        /* Short name for this line */
    "Wave Input",                     /* Long name for this line */
    MIXERLINE_TARGETTYPE_WAVEIN,      /* MIXERLINE_TARGETTYPE_xxxx */
    /* The following info is just some info about the Mixer Device for this
    line, prepended to the above info. It's mostly just for reference and is only
valid if the
        Target Type field is not MIXERLINE_TARGETTYPE_UNDEFINED. */
    0, 0, 0, 0x0100, "Example Sound Card",
};
```

Notice that the *dwComponentType* for the "ADC Wave Input" destination line is MIXERLINE_COMPONENTTYPE_DST_WAVEIN to indicate that it's a wave input. Also, I've chosen a value of 0xFFFF0001 for the *dwLineID* -- a different value than the "Speaker Out" destination line. Also, note that since our sound card is capable of digitizing in stereo, the *cChannels* field is 2. As you'll recall, there are 2 source lines connected to our

"ADC Wave Input" destination line, so the *cConnections* field is 2. Finally, note that the *dwDestination* field is 1, since this is the second destination line in the Mixer.

The mixer API mixerGetLineInfo() fills in a MIXERLINE struct for a specified line. This is how you retrieve info about a line. If you don't know a line's ID number (as would be the case when you're first enumerating the lines), then you can reference it by index. You pass mixerGetLineInfo() the value MIXER_GETLINEINFOF_DESTINATION to notify it that you want to reference the line by its index value. For example, here's how you would retrieve info about the first destination line in our example Mixer. Before calling mixerGetLineInfo() and passing it a MIXERLINE to fill in, you must initialize 2 fields. The *cbStruct* field must be set to the size of the MIXERLINE struct you're passing, and the *dwDestination* field must be set to the index value of the line whose info you wish to retrieve. Remember that the first destination line has an index of 0, so to retrieve its info, we set dwDestination to 0.

```
MIXERLINE       mixerline;
MMRESULT        err;

/* Get info about the first destination line by its index */
mixerline.cbStruct = sizeof(MIXERLINE);
mixerline.dwDestination = 0;

if ((err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerline,
MIXER_GETLINEINFOF_DESTINATION)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineInfo()\n", err);
}
```

When the above call returns, mixerGetLineInfo() will have filled in our MIXERLINE struct as per the "Speaker Out" (mixerline_SpkrOut) MIXERLINE struct shown above. (After all, the "Speaker Out" is the first line in our example Mixer -- it has an index of 0).

Now, if you want to fetch info about the second destination line in the mixer, the only thing different is the index value you stuff into the *dwDestination* field, as so:

```
/* Get info about the second destination line by its index */
mixerLine.cbStruct = sizeof(MIXERLINE);
mixerLine.dwDestination = 1;

if ((err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerLine,
MIXER_GETLINEINFOF_DESTINATION)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineInfo()\n", err);
}
```

When the above call returns, mixerGetLineInfo() will have filled in our MIXERLINE struct as per the "ADC Wave Input" (mixerline_WaveIn) MIXERLINE struct shown above. (After all, the "ADC Wave Input" is the second line in our example Mixer -- it has an index of 1).

Now, you should see how you can enumerate the destination lines by their indexes. Here is an example of printing out the names of all destination lines in our Mixer:

```
MIXERCAPS       mixcaps;
MIXERLINE       mixerline;
MMRESULT        err;
unsigned long i;
```

```c
/* Get info about the first Mixer Device */
if (!(err = mixerGetDevCaps((UINT)mixerHandle, &mixcaps, sizeof(MIXERCAPS))))
{
    /* Print out the name of each destination line */
    for (i = 0; i < mixercaps.cDestinations; i++)
    {
        mixerline.cbStruct = sizeof(MIXERLINE);
        mixerline.dwDestination = i;

        if (!(err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerline,
MIXER_GETLINEINFOF_DESTINATION)))
        {
            printf("Destination #%lu = %s\n", i, mixerline.szName);
        }
    }
}
```

Now, we need to enumerate the source lines for each destination line. We also use an index value to reference each source line. The first source line, for a given destination line, has an index value of 0. The second source line, for that destination line, has an index of 1. The third source line has an index of 2. Etc. Remember that the "Speaker Out" had 4 source lines going into it: "Internal CD Audio", "Synth", "DAC Wave Out" and "Microphone Input". So their respective index values are 0, 1, 2, and 3. Let's look at the MIXERLINE structs for those 4 source lines. Assume that each one of them has 2 controls, a volume slider and a mute switch. Also, assume that each one is a stereo source.

```c
MIXERLINE mixerline_CD = {
    sizeof(MIXERLINE),
    0,                                  /* zero based index of destination line */
    0,                                  /* zero based source index */
    0x00000000,                         /* unique ID # for this line */
    MIXERLINE_LINEF_ACTIVE|MIXERLINE_LINEF_SOURCE, /* state/information about line */
    0,
    MIXERLINE_COMPONENTTYPE_SRC_COMPACTDISC,  /* component type */
    2,                                  /* # of channels this line supports */
    0,                                  /* Not applicable for source lines */
    2,                                  /* # of controls in this line */
    "CD Audio",                         /* Short name for this line */
    "Internal CD Audio",                /* Long name for this line */
    MIXERLINE_TARGETTYPE_UNDEFINED, /* MIXERLINE_TARGETTYPE_xxxx */
    0, 0, 0, 0x0100, "Example Sound Card",
};

MIXERLINE mixerline_Synth = {
    sizeof(MIXERLINE),
    0,                                  /* zero based index of destination line */
    1,                                  /* zero based source index */
    0x00000001,                         /* unique ID # for this line */
    MIXERLINE_LINEF_ACTIVE|MIXERLINE_LINEF_SOURCE,
    0,
    MIXERLINE_COMPONENTTYPE_SRC_SYNTHESIZER,  /* component type */
    2,                                  /* # of channels this line supports */
    0,                                  /* Not applicable for source lines */
    2,                                  /* # of controls in this line */
    "Synth",                            /* Short name for this line */
    "Synth",                            /* Long name for this line */
    MIXERLINE_TARGETTYPE_UNDEFINED, /* MIXERLINE_TARGETTYPE_xxxx */
    0, 0, 0, 0x0100, "Example Sound Card",
};
```

```
MIXERLINE mixerline_WaveOut = {
    sizeof(MIXERLINE),
    0,                                /* zero based index of destination line */
    2,                                /* zero based source index */
    0x00000002,                       /* unique ID # for this line */
    MIXERLINE_LINEF_ACTIVE|MIXERLINE_LINEF_SOURCE,
    0,
    MIXERLINE_COMPONENTTYPE_SRC_WAVEOUT,  /* component type */
    2,                                /* # of channels this line supports */
    0,                                /* Not applicable for source lines */
    2,                                /* # of controls in this line */
    "Wave Out",                       /* Short name for this line */
    "DAC Wave Out",                   /* Long name for this line */
    MIXERLINE_TARGETTYPE_WAVEOUT,     /* MIXERLINE_TARGETTYPE_xxxx */
    0, 0, 0, 0x0100, "Example Sound Card",
};

MIXERLINE mixerline_Mic = {
    sizeof(MIXERLINE),
    0,                                /* zero based index of destination line */
    3,                                /* zero based source index */
    0x00000003,                       /* unique ID # for this line */
    MIXERLINE_LINEF_ACTIVE|MIXERLINE_LINEF_SOURCE,
    0,
    MIXERLINE_COMPONENTTYPE_SRC_MICROPHONE,  /* component type */
    2,                                /* # of channels this line supports */
    0,                                /* Not applicable for source lines */
    2,                                /* # of controls in this line */
    "Mic",                            /* Short name for this line */
    "Microphone Input",               /* Long name for this line */
    MIXERLINE_TARGETTYPE_WAVEIN,      /* MIXERLINE_TARGETTYPE_xxxx */
    0, 0, 0, 0x0100, "Example Sound Card",
};
```

One thing that you'll note is, unlike with the destination lines, the MIXERLINEs for all source lines have their MIXERLINE_LINEF_SOURCE flag bit set. When this bit is set, you know that you have info on a source line. Secondly, note that the zero-based index for the destination line is 0. That's because all of the above source lines connect to the "Speaker Out" destination line, which is the first line in our mixer (and therefore has an index value of 0). Next, note that the zero-based source indexes for the 4 source lines are 0, 1, 2, and 3, respectively. Finally, note that the ID of each source line is unique -- unlike any other line, including any of the destination lines.

The mixer API mixerGetLineInfo() fills in a MIXERLINE struct for a source line too. Again, you can reference a source line by its index, but you also need to know the index of its respective destination line. You pass mixerGetLineInfo() the value MIXER_GETLINEINFOF_SOURCE to notify it that you want to reference the line by its index value. For example, here's how you would retrieve info about the first source line (of the "Speaker Out" destination line) in our example Mixer. Before calling mixerGetLineInfo() and passing it a MIXERLINE to fill in, you must initialize 3 fields. The *cbStruct* field must be set to the size of the MIXERLINE struct you're passing, the *dwSource* field must be set to the index value of the source line whose info you wish to retrieve, and the *dwDestination* field must be set to the index value of the destination line to which this source line connects.

```
MIXERLINE      mixerline;
MMRESULT       err;

/* Get info about the first source line (of the first destination line) by its index
*/
mixerline.cbStruct = sizeof(MIXERLINE);
```

```
mixerline.dwDestination = 0;
mixerline.dwSource = 0;

if ((err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerline,
MIXER_GETLINEINFOF_SOURCE)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineInfo()\n", err);
}
```

When the above call returns, mixerGetLineInfo() will have filled in our MIXERLINE struct as per the "Internal CD Audio" (mixerline_CD) MIXERLINE struct shown above. So for example, you can extract its line ID number from the MIXERLINE's dwLineID field.

Now, if you want to fetch info about the second source line of the "Speaker Out" destination line, the only thing different is the index value you stuff into the *dwSource* field, as so:

```
/* Get info about the second source line by its index */
mixerline.cbStruct = sizeof(MIXERLINE);
mixerline.dwDestination = 0;
mixerline.dwSource = 1;

if ((err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerline,
MIXER_GETLINEINFOF_SOURCE)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineInfo()\n", err);
}
```

When the above call returns, mixerGetLineInfo() will have filled in our MIXERLINE struct as per the "Synth" (mixerline_Synth) MIXERLINE struct shown above.

Now, you should see how you can enumerate the source lines (of each destination line) by their indexes. Here is an example of printing out the names of all destination lines, and their source lines, in our Mixer:

```
MIXERCAPS       mixcaps;
MIXERLINE       mixerline;
MMRESULT        err;
unsigned long i, n, numSrc;

/* Get info about the first Mixer Device */
if (!(err = mixerGetDevCaps((UINT)mixerHandle, &mixcaps, sizeof(MIXERCAPS))))
{
    /* Print out the name of each destination line */
    for (i = 0; i < mixercaps.cDestinations; i++)
    {
        mixerline.cbStruct = sizeof(MIXERLINE);
        mixerline.dwDestination = i;

        if (!(err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerline,
MIXER_GETLINEINFOF_DESTINATION)))
        {
            printf("Destination #%lu = %s\n", i, mixerline.szName);

            /* Print out the name of each source line in this destination */
            numSrc = mixerline.cConnections;
            for (n = 0; n < numSrc; n++)
```

```
        {
            mixerline.cbStruct = sizeof(MIXERLINE);
            mixerline.dwDestination = i;
            mixerline.dwSource = n;

            if (!(err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerline,
MIXER_GETLINEINFOF_SOURCE)))
            {
                printf("\tSource #%lu = %s\n", i, mixerline.szName);
            }
        }
    }
}
}
```

Getting info about a line by its ID

Once you know a line's ID number (which you can extract from the MIXERLINE's dwLineID after enumerating the line as shown above), you can later retrieve info on it by referencing its ID (instead of its index). If you're dealing with a source line, you do not need to know the index of the destination line to which the source line is connected. You merely initialize the MIXERLINE's *dwLineID* field to the ID number of the desired line, and then specify MIXER_GETLINEINFOF_LINEID when calling mixerGetLineInfo() as so:

```
/* Get info about the "Microphone Input" source line by its ID */
mixerline.cbStruct = sizeof(MIXERLINE);
mixerline.dwLineID = 0x00000003; /* The ID for "Microphone Input" */

if ((err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerline,
MIXER_GETLINEINFOF_LINEID)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineInfo()\n", err);
}
```

The above works with both destination and source lines. Once you know a line's ID, you can directly retrieve info on it without needing to know anything about indexes.

Getting info about a line by its Type

Often, you don't need to know about all of the lines in a Mixer. You may be writing a program that would deal only with a specific type of line. For example, let's say that you're writing a simple MIDI file player. Now, certain components of our example sound card are of no use to you at all. MIDI is not digital audio data, so the "DAC Wave In" component (and all source lines running into it) are of no concern to you. Likewise, the "Internal CD Audio", "DAC Wave Out" and "Microphone Input" source lines to the "Speaker Out" destination line are of no concern to you. The only component on our card which is capable of dealing with the playback of MIDI data is the "Synth" component that goes to the "Speaker Out". It is this line's controls that will affect the playback of MIDI data.

So, rather than enumerating all of the lines in the mixer until you come to the one with the MIXERLINE_COMPONENTTYPE_SRC_SYNTHESIZER type, mixerGetLineInfo() lets you directly get info about a line that matches your desired type. You merely initialize the MIXERLINE's *dwComponentType* field to the desired type of line, and then specify MIXER_GETLINEINFOF_COMPONENTTYPE when calling mixerGetLineInfo() as so:

```
/* Get info about a "Synth" type of source line */
mixerline.cbStruct = sizeof(MIXERLINE);
mixerline.dwComponentType = MIXERLINE_COMPONENTTYPE_SRC_SYNTHESIZER; /* We want a
```

```
Synth type */

if ((err = mixerGetLineInfo((HMIXEROBJ)mixerHandle, &mixerline,
MIXER_GETLINEINFOF_COMPONENTTYPE)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineInfo()\n", err);
}
```

This will fill in the MIXERLINE struct with info about the first line in the Mixer which has a type of MIXERLINE_COMPONENTTYPE_SRC_SYNTHESIZER. (If there are no such lines in the Mixer, a MIXERR_INVALLINE error is returned). Once you have that info, you can then directly manipulate its controls. This saves having to enumerate and search for a desired line when your needs are specific.

MIXERCONTROL structures, and enumerating controls

You use the mixer API mixerGetLineControls() to retrieve info about controls for a line. This API fills in a MIXERCONTROL struct with info about a control.

Let's examine the MIXERCONTROL struct. As mentioned before, our "Speaker Out" destination line has 2 controls; a volume slider, and a mute switch. Each one of these controls will have one MIXERCONTROL associated with it. Let's examine the MIXERCONTROL for each of these controls:

```
MIXERCONTROL mixerctl_Spkr_Vol = {
    sizeof(MIXERCONTROL),               /* size of a MIXERCONTROL */
    0x00000000,                         /* unique ID # for this control */
    MIXERCONTROL_CONTROLTYPE_VOLUME,    /* type of control */
    MIXERCONTROL_CONTROLF_UNIFORM,      /* flag bits */
    0,                                  /* # of items per channels (used only if the
                                        MIXERCONTROL_CONTROLF_MULTIPLE flag bit is also
set) */
    "Volume",                           /* Short name for this control */
    "Speaker Out Volume",               /* Long name for this control */
    0,                                  /* Minimum value to which this control can be
set */
    65535,                              /* Maximum value to which this control can be
set */
    0, 0, 0, 0,                         /* These fields are reserved for future use */
    31,                                 /* Step amount for the value */
    0, 0, 0, 0, 0,                      /* These fields are reserved for future use */
};

MIXERCONTROL mixerctl_Spkr_Mute = {
    sizeof(MIXERCONTROL),
    0x00000001,                         /* unique ID # for this control */
    MIXERCONTROL_CONTROLTYPE_MUTE,      /* type of control */
    MIXERCONTROL_CONTROLF_UNIFORM,
    0,
    "Mute",                             /* Short name for this control */
    "Speaker Out Mute",                 /* Long name for this control */
    0,                                  /* Minimum value to which this control can be
set */
    1,                                  /* Maximum value to which this control can be
set */
    0, 0, 0, 0,
    0,                                  /* Step amount for the value */
```

```
   0, 0, 0, 0, 0,
};
```

There are several things to note above. First, note that each control has a unique ID. These ID numbers don't have to be unique with regard to the IDs of lines. (For example, the control ID of the mixerctl_Spkr_Vol control happens to be the same as the line ID of the mixerline_CD line). But, each control must have an ID unique from any other control, including the controls of other lines. (For example, the volume slider of the "Speaker Out" line can't have the same ID number as the mute switch of the "ADC Wave In" line).

I have set the MIXERCONTROL_CONTROLF_UNIFORM flag. What this means is that, although the "Speaker Out" is stereo (ie, it has 2 channels), there is not a separate volume control for each channel. (There are not individual, left channel and right channel volume settings). There is only one volume setting for both channels, and therefore both channels will be set to the same volume always. (Later, we'll study controls that aren't uniform).

Also note that each control has an appropriate type. The volume slider has a MIXERCONTROL_CONTROLTYPE_VOLUME type, and the mute switch has a MIXERCONTROL_CONTROLTYPE_MUTE type.

The MIXERCONTROL tells you what the min and max values can be for the control. For example, the volume slider can be set to any value inbetween 0 and 65,535. 0 is the minimum setting (ie, volume is lowest), and 65,535 is the maximum setting (ie, volume is loudest). So, does that mean that the volume slider has 65,535 discrete steps? (ie, Can it be set to <u>any</u> value from 0 to 65,535, inclusive)? Not necessarily. You also have to look at the step amount field. This tells you how many valid steps the control has. In this case, we have 31 valid steps. This means that the first valid setting is 0, but the second valid setting is 65,535 - (65,535/31) and the third valid setting is 65,535 - (65,535/(31*2)), etc. In other words, we have only 31 valid settings within that 0 to 65,535 range. (NOTE: the dwMinimum/dwMaximum fields are declared in a union with the lMinimum/ lMaximum fields. You'll reference the former two when dealing with a control type with an unsigned value -- ie, a control type whose value is set with a MIXERCONTROLDETAILS_BOOLEAN or MIXERCONTROLDETAILS_UNSIGNED struct. You'll reference the latter two when dealing with a control type with a signed value -- ie, a control type whose value is set with a MIXERCONTROLDETAILS_SIGNED struct).

Enumerating controls is a bit different than enumerating lines. For one thing, you don't employ indexes with controls. Secondly, you can retrieve info about a single control only if you already know its control ID number. Otherwise, you must simultaneously retrieve info for <u>all</u> the controls for a given line.

Obviously, when you're first enumerating the controls for a line, you don't know the ID of each control. So you need to retrieve info on all of the controls with a single call to mixerGetLineControls(). In this case, you need to pass mixerGetLineControls() an array of MIXERCONTROL structs. There must be one struct for every control in the line.

For example, we know that our "Speaker Out" destination line has 2 controls; a volume slider, and a mute switch. (Remember that its MIXERLINE's cControls field is 2). So, to retrieve information about them, we must pass mixerGetLineControls() an array containing 2 MIXERCONTROL structs. We must also pass the value MIXER_GETLINECONTROLSF_ALL to indicate we want info on all the controls. We also pass a special structure called a MIXERLINECONTROLS which we must first initialize. This structure tells mixerGetLineControls() for which line we want to receive info about its controls. We also provide the pointer to our array of MIXERCONTROL structs via this additional struct. Here then is an example of retrieving info about all the controls in the "Speaker Out" line.

```
/* Let's just declare an array of 2 MIXERCONTROL structs since
   we know that the "Speaker Out" has 2 controls. For a real program,
   you typically won't know ahead of time how big an array you'll need,
   and therefore would instead allocate an appropriately sized array */
MIXERCONTROL        mixerControlArray[2];
MIXERLINECONTROLS   mixerLineControls;
MMRESULT            err;

mixerLineControls.cbStruct = sizeof(MIXERLINECONTROLS);
```

```
/* The "Speaker Out" line has a total of 2 controls. And
   that's how many we want to retrieve info for here */
mixerLineControls.cControls = 2;

/* Tell mixerGetLineControls() for which line we're retrieving info.
   We do this by putting the desired line's ID number in dwLineID.
   The "Speaker Out" line has an ID of 0xFFFF0000 */
mixerLineControls.dwLineID = 0xFFFF0000;

/* Give mixerGetLineControls() the address of our array of
   MIXERCONTROL structs big enough to hold info on all controls */
mixerLineControls.pamxctrl = &mixerControlArray[0];

/* Tell mixerGetLineControls() how big each MIXERCONTROL is. This
   saves having to initialize the cbStruct of each individual
   MIXERCONTROL in the array */
mixerLineControls.cbmxctrl = sizeof(MIXERCONTROL);

/* Retrieve info on all controls for this line simultaneously */
if ((err = mixerGetLineControls((HMIXEROBJ)mixerHandle, &mixerLineControls,
MIXER_GETLINECONTROLSF_ALL)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineControls()\n", err);
}
```

When mixerGetLineControls() returns above, our mixerControlArray[] array will have been filled in. mixerControlArray[0] will have been filled in as per the mixerctl_Spkr_Vol MIXERCONTROL struct shown above, and mixerControlArray[1] will have been filled in as per the mixerctl_Spkr_Mute MIXERCONTROL struct shown above. So, you can, for example, extract their control ID numbers from their respective MIXERCONTROL's dwControlID field.

It's possible to retrieve info on only one control, if you know its ID or Type. But it is not possible to retrieve info on more than one control, but less than the total number of controls. For example, assume that our "Speaker Out" had 5 controls (instead of only 2). You couldn't retrieve info on only the first 3, for example. You can either retrieve info on one (out of the 5) at a time, or all 5 of them at once. (ie, It's either one at a time, or all).

Getting info about a control by its ID

Once you know a control's ID number (which you can extract from its MIXERCONTROL's dwControlID field after enumerating the control as shown above), you can later retrieve info on just this one control by referencing its ID. You don't even need to know the ID of the line to which this control belongs. And you aren't forced to retrieve info for all of the other controls in this line. You merely initialize the MIXERLINECONTROLS's *dwControlID* field to the ID number of the desired control, and then specify MIXER_GETLINECONTROLSF_ONEBYID when calling mixerGetLineControls() as so:

```
/* We need only 1 MIXERCONTROL struct since
   we're fetching info for only 1 control */
MIXERCONTROL        mixerControlArray;
MIXERLINECONTROLS  mixerLineControls;
MMRESULT           err;

mixerLineControls.cbStruct = sizeof(MIXERLINECONTROLS);

/* We want to fetch info on only 1 control */
mixerLineControls.cControls = 1;

/* Tell mixerGetLineControls() for which control we're retrieving
```

```
   info. We do this by putting the desired control's ID number in
   dwControlID. The "Speaker Out" line's volume slider has an ID
   of 0x00000000 */
mixerLineControls.dwControlID = 0x00000000;

/* Give mixerGetLineControls() the address of the
   MIXERCONTROL struct to hold info */
mixerLineControls.pamxctrl = &mixerControlArray;

/* Tell mixerGetLineControls() how big the MIXERCONTROL is. This
   saves having to initialize the cbStruct of the MIXERCONTROL itself */
mixerLineControls.cbmxctrl = sizeof(MIXERCONTROL);

/* Retrieve info on only the volume slider control for the "Speaker Out" line */
if ((err = mixerGetLineControls((HMIXEROBJ)mixerHandle, &mixerLineControls,
MIXER_GETLINECONTROLSF_ONEBYID)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineControls()\n", err);
}
```

Getting info about a control by its Type

Often, you don't need to know about all of the controls in a line. You may be writing a program that would deal only with a specific type of control. For example, let's say that you're writing a simple MIDI file player and the only control you want to present to the enduser is the volume slider of the "Synth". You saw earlier how you could search for that MIDI playback component by Type and fill in a MIXERLINE struct with info about it, such as its line ID. You can then use that line ID to search for a particular type of control within that line, For example, you can search for a control with a type of MIXERCONTROL_CONTROLTYPE_VOLUME.

So, rather than enumerating all of the controls in the line until you come to the one with the MIXERCONTROL_CONTROLTYPE_VOLUME type, mixerGetLineControls() lets you directly get info about a line that matches your desired type. You merely initialize the MIXERLINECONTROLS's *dwControlType* field to the desired type of control, and then specify MIXER_GETLINECONTROLSF_ONEBYTYPE when calling mixerGetLineControls() as so. (Assume that you've already fetched the line ID for the "Synth" line, and stored it in "SynthID").

```
MIXERCONTROL         mixerControlArray;
MIXERLINECONTROLS    mixerLineControls;
MMRESULT             err;

mixerLineControls.cbStruct = sizeof(MIXERLINECONTROLS);

/* Tell mixerGetLineControls() for which line we're retrieving info.
   We do this by putting the desired line's ID number in dwLineID */
mixerLineControls.dwLineID = SynthID;

/* We want to fetch info on only 1 control */
mixerLineControls.cControls = 1;

/* Tell mixerGetLineControls() for which type of control we're
   retrieving info. We do this by putting the desired control type
   in dwControlType */
mixerLineControls.dwControlType = MIXERCONTROL_CONTROLTYPE_VOLUME;

/* Give mixerGetLineControls() the address of the MIXERCONTROL
   struct to hold info */
```

```
mixerLineControls.pamxctrl = &mixerControlArray;

/* Tell mixerGetLineControls() how big the MIXERCONTROL is. This
   saves having to initialize the cbStruct of the MIXERCONTROL itself */
mixerLineControls.cbmxctrl = sizeof(MIXERCONTROL);

/* Retrieve info on only any volume slider control for this line */
if ((err = mixerGetLineControls((HMIXEROBJ)mixerHandle, &mixerLineControls,
MIXER_GETLINECONTROLSF_ONEBYTYPE)))
{
    /* An error */
    printf("Error #%d calling mixerGetLineControls()\n", err);
}
```

This will fill in the MIXERCONTROL struct with info about the first control in the line which has a type of MIXERCONTROL_CONTROLTYPE_VOLUME. (If there are no such controls in the line, a MIXERR_INVALCONTROL error is returned). Once you have that info, you can then directly manipulate that control. For example, you can extract its control ID number from the MIXERCONTROL's dwControlID field. This saves having to enumerate and search for a desired control when your needs are specific.

Retrieving and setting a control's value

Now we're getting to the whole purpose of a mixer; to retrieve a control's value (so that you can display its current setting to the enduser), and to set a control's value (so that you can allow the enduser to adjust it).

To retrieve or set a control's value, you must know its control ID. You use mixerGetControlDetails() to retrieve its current value, and mixerSetControlDetails() to set it to a specific value. These functions utilize a MIXERCONTROLDETAILS struct. You initialize certain fields to tell mixerGetControlDetails()/mixerSetControlDetails() what control whose value you're retrieving/setting, and you also supply a pointer to another structure that will contain the actual value.

For example, let's consider retrieving the current value of the "Speaker Out" line's volume slider. By now, you know how to get info on that control, for example, its ID number. In order to retrieve a control's value, we need to supply a special structure into which its value is returned. What kind of structure do we use? Well, that depends upon what type of control it is. The volume slider is a MIXERCONTROL_CONTROLTYPE_VOLUME type. If you go back to the chart about the fader class of controls, it tells you that its value is retrieved using a MIXERCONTROLDETAILS_UNSIGNED struct. This struct has only one field into which the control's value is returned; dwValue. So we supply a MIXERCONTROLDETAILS_UNSIGNED struct to mixerGetControlDetails() (via the MIXERCONTROLDETAILS struct). Here then is an example, of retrieving and printing the current value of the "Speaker Out" line's volume slider:

```
/* We need a MIXERCONTROLDETAILS_UNSIGNED struct to retrieve the
   value of a control whose type is MIXERCONTROL_CONTROLTYPE_VOLUME */
MIXERCONTROLDETAILS_UNSIGNED value;
MIXERCONTROLDETAILS         mixerControlDetails;
MMRESULT                    err;

mixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);

/* Tell mixerGetControlDetails() which control whose value we
   want to retrieve. We do this by putting the desired control's
   ID number in dwControlID. Remember that the "Speaker Out" line's
   volume slider has an ID of 0x00000000 */
mixerControlDetails.dwControlID = 0x00000000;

/* This is always 1 for a MIXERCONTROL_CONTROLF_UNIFORM control */
mixerControlDetails.cChannels = 1;
```

```c
/* This is always 0 except for a MIXERCONTROL_CONTROLF_MULTIPLE control */
mixerControlDetails.cMultipleItems = 0;

/* Give mixerGetControlDetails() the address of the
   MIXERCONTROLDETAILS_UNSIGNED struct into which to return the value */
mixerControlDetails.paDetails = &value;

/* Tell mixerGetControlDetails() how big the MIXERCONTROLDETAILS_UNSIGNED is */
mixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);

/* Retrieve the current value of the volume slider control for this line */
if ((err = mixerGetControlDetails((HMIXEROBJ)mixerHandle, &mixerControlDetails,
MIXER_GETCONTROLDETAILSF_VALUE)))
{
    /* An error */
    printf("Error #%d calling mixerGetControlDetails()\n", err);
}
else
{
    printf("It's value is %lu\n", value.dwValue);
}
```

To set a control's value, you simply fill in the special structure that contains the value, and pass it to mixerSetControlDetails(). You also specify MIXER_SETCONTROLDETAILSF_VALUE. Here is an example of setting the "Speaker Out" line's volume slider to a value of 31:

```c
MIXERCONTROLDETAILS_UNSIGNED value;
MIXERCONTROLDETAILS          mixerControlDetails;
MMRESULT                     err;

mixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);

/* Tell mixerSetControlDetails() which control whose value we
   want to set. We do this by putting the desired control's
   ID number in dwControlID. Remember that the "Speaker Out" line's
   volume slider has an ID of 0x00000000 */
mixerControlDetails.dwControlID = 0x00000000;

/* This is always 1 for a MIXERCONTROL_CONTROLF_UNIFORM control */
mixerControlDetails.cChannels = 1;

/* This is always 0 except for a MIXERCONTROL_CONTROLF_MULTIPLE control */
mixerControlDetails.cMultipleItems = 0;

/* Give mixerSetControlDetails() the address of the
   MIXERCONTROLDETAILS_UNSIGNED struct into which we place the value */
mixerControlDetails.paDetails = &value;

/* Tell mixerSetControlDetails() how big the MIXERCONTROLDETAILS_UNSIGNED is */
mixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);

/* Store the value */
value.dwValue = 31;

/* Set the value of the volume slider control for this line */
if ((err = mixerSetControlDetails((HMIXEROBJ)mixerHandle, &mixerControlDetails,
MIXER_SETCONTROLDETAILSF_VALUE)))
```

```
{
    /* An error */
    printf("Error #%d calling mixerSetControlDetails()\n", err);
}
```

Multi-channel controls

As mentioned, when a control's MIXERCONTROL_CONTROLF_UNIFORM flag bit is set, then it doesn't have individual values for each one of its channels. For example, with the "Speaker Out", there is not a separate volume for the left and right channels of this stereo line.

But if a control's MIXERCONTROL_CONTROLF_UNIFORM flag bit is <u>not</u> set, and it has more than one channel, then each channel has its own value. For this reason, you'll need more than one of the special structures to retrieve/set the values for all channels. For example, let's assume that the volume slider for the "Speaker Out" does not have its MIXERCONTROL_CONTROLF_UNIFORM flag bit set. Since the Speaker Out line has 2 channels, that means that we need 2 MIXERCONTROLDETAILS_UNSIGNED structs to retrieve/set the value for the Left and Right channels' volumes respectively. We need to use an array of MIXERCONTROLDETAILS_UNSIGNED structs. The first MIXERCONTROLDETAILS_UNSIGNED struct will be for the first (ie, Left) channel, and the second MIXERCONTROLDETAILS_UNSIGNED struct will be for the second (ie, Right) channel. Here is an example of retrieving the values of the left and right channels of our volume slider for the "Speaker Out" line:

```
/* We need 2 MIXERCONTROLDETAILS_UNSIGNED structs to retrieve the
   values of a stereo control that is not MIXERCONTROL_CONTROLF_UNIFORM */
MIXERCONTROLDETAILS_UNSIGNED value[2];
MIXERCONTROLDETAILS          mixerControlDetails;
MMRESULT                     err;

mixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);

/* Tell mixerGetControlDetails() which control whose value we
   want to retrieve. We do this by putting the desired control's
   ID number in dwControlID. Remember that the "Speaker Out" line's
   volume slider has an ID of 0x00000000 */
mixerControlDetails.dwControlID = 0x00000000;

/* We want to retrieve values for both channels */
mixerControlDetails.cChannels = 2;

/* This is always 0 except for a MIXERCONTROL_CONTROLF_MULTIPLE control */
mixerControlDetails.cMultipleItems = 0;

/* Give mixerGetControlDetails() the address of the
   MIXERCONTROLDETAILS_UNSIGNED array into which to return the values */
mixerControlDetails.paDetails = &value[0];

/* Tell mixerGetControlDetails() how big each MIXERCONTROLDETAILS_UNSIGNED is */
mixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);

/* Retrieve the current values of both channels */
if ((err = mixerGetControlDetails((HMIXEROBJ)mixerHandle, &mixerControlDetails,
MIXER_GETCONTROLDETAILSF_VALUE)))
{
    /* An error */
    printf("Error #%d calling mixerGetControlDetails()\n", err);
}
else
```

```c
{
    printf("The left channel's volume is %lu\n", value[0].dwValue);
    printf("The right channel's volume is %lu\n", value[1].dwValue);
}
```

To set the values of both channels, you fill in the values of both MIXERCONTROLDETAILS_UNSIGNED structs. Here is an example of setting the left channel's volume to 31 and the right channel's volume to 0.

```c
/* We need 2 MIXERCONTROLDETAILS_UNSIGNED structs to set the
   values of a stereo control that is not MIXERCONTROL_CONTROLF_UNIFORM */
MIXERCONTROLDETAILS_UNSIGNED value[2];
MIXERCONTROLDETAILS         mixerControlDetails;
MMRESULT                    err;

mixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);

/* Tell mixerSetControlDetails() which control whose value we
   want to set. We do this by putting the desired control's
   ID number in dwControlID. Remember that the "Speaker Out" line's
   volume slider has an ID of 0x00000000 */
mixerControlDetails.dwControlID = 0x00000000;

/* We want to set values for both channels */
mixerControlDetails.cChannels = 2;

/* This is always 0 except for a MIXERCONTROL_CONTROLF_MULTIPLE control */
mixerControlDetails.cMultipleItems = 0;

/* Give mixerSetControlDetails() the address of the
   MIXERCONTROLDETAILS_UNSIGNED structs into which we place the values */
mixerControlDetails.paDetails = &value[0];

/* Tell mixerSetControlDetails() how big each MIXERCONTROLDETAILS_UNSIGNED is */
mixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);

/* Store the left channel's value */
value[0].dwValue = 31;

/* Store the right channel's value */
value[1].dwValue = 0;

/* Set the left/right values of the volume slider control for this line */
if ((err = mixerSetControlDetails((HMIXEROBJ)mixerHandle, &mixerControlDetails,
MIXER_SETCONTROLDETAILSF_VALUE)))
{
    /* An error */
    printf("Error #%d calling mixerSetControlDetails()\n", err);
}
```

Of course, a control may have even more than 2 channels. You always need an array of special structures large enough to accomodate all channels for a given control, so typically, you'll allocate the array as needed.

It is not legal to retrieve or set only some of the channels. For example, if a control has 8 channels, it's not legal to try to retrieve the values of only the first 2 channels. You must always retrieve/set the values of all channels simultaneously. But there is one caveat to this rule. It concerns setting a value. If you set only the value for the first channel, then mixerSetControlDetails() automatically treats the control as if it was MIXERCONTROL_CONTROLF_UNIFORM. The net result is that all channels get set to that one value. So, you can quickly set all channels to the same value just by setting the

value of only the first channel.

Multi-item controls

You won't often encounter multi-item controls. A multi-item control is one that has several values associated with each channel. An example could be a graphic equalizer control. Let's study a simple example. Assume that you have the following 3 band graphic equalizer built into a sound card:

This control has three values associated with it -- the value for the "Low" band, the value for the "Mid" band, and the value for the "High" band. (ie, It is assumed that each band can be set to a different value, otherwise it would be a fairly useless graphic EQ). The way that this would be represented is a a multi-item control. It has 3 items (ie, values) associated with it.

Let's further assume that this control is in the "Speaker Out" line.

Let's examine its MIXERCONTROL struct. A multi-item control has the MIXERCONTROL_CONTROLF_MULTIPLE bit set of its MIXERCONTROL's *dwControlType* field. The MIXERCONTROL's *cMultipleItems* field will also tell how many items are in each channel.

```
MIXERCONTROL mixerctl_EQ = {
    sizeof(MIXERCONTROL),               /* size of a MIXERCONTROL */
    0x00000002,                         /* unique ID # for this control */
    MIXERCONTROL_CONTROLTYPE_EQUALIZER, /* type of control */
    MIXERCONTROL_CONTROLF_UNIFORM|MIXERCONTROL_CONTROLF_MULTIPLE,   /* flag bits */
    3,                                  /* # of items per channel */
    "EQ",                               /* Short name for this control */
    "Graphic Equalizer",                /* Long name for this control */
    0,                                  /* Minimum value to which this control can be
set */
    65535,                              /* Maximum value to which this control can be
set */
    0, 0, 0, 0,                         /* These fields are reserved for future use */
    31,                                 /* Step amount for the value */
    0, 0, 0, 0, 0,                      /* These fields are reserved for future use */
};
```

First of all, note that the ID is different than all of the other controls for this Mixer. Also, note that its MIXERCONTROL_CONTROLF_MULTIPLE flag bit is set. The *cMultipleItems* field is set to 3 to indicate that there are 3 items per channel. (But since I made this control MIXERCONTROL_CONTROLF_UNIFORM, it still has only 3 values total, even though the "Speaker Out" line is stereo. In other words, the value for each band affects both channels equally).

To query the values for all 3 bands, we need an array of 3 structures to fetch the values. What kind of structures? Well, the MIXERCONTROL_CONTROLTYPE_EQUALIZER control type is of the fader class, and you'll remember that all the types in that class use a MIXERCONTROLDETAILS_UNSIGNED struct to set/fetch values. Here's how we query the current values of the 3 bands:

```
/* We need 3 MIXERCONTROLDETAILS_UNSIGNED structs to retrieve the
   values of this control */
MIXERCONTROLDETAILS_UNSIGNED value[3];
MIXERCONTROLDETAILS          mixerControlDetails;
```

```c
MMRESULT                        err;

mixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);

/* Tell mixerGetControlDetails() which control whose value we
   want to retrieve */
mixerControlDetails.dwControlID = 0x00000002;

/* It's a MIXERCONTROL_CONTROLF_UNIFORM control, so the values
   for all channels are the same as the first */
mixerControlDetails.cChannels = 1;

/* There are 3 items per channel */
mixerControlDetails.cMultipleItems = 3;

/* Give mixerGetControlDetails() the address of the
   MIXERCONTROLDETAILS_UNSIGNED array into which to return the values */
mixerControlDetails.paDetails = &value[0];

/* Tell mixerGetControlDetails() how big each MIXERCONTROLDETAILS_UNSIGNED is */
mixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);

/* Retrieve the current values of all 3 bands */
if ((err = mixerGetControlDetails((HMIXEROBJ)mixerHandle, &mixerControlDetails,
MIXER_GETCONTROLDETAILSF_VALUE)))
{
    /* An error */
    printf("Error #%d calling mixerGetControlDetails()\n", err);
}
else
{
    printf("The Low band is %lu\n", value[0].dwValue);
    printf("The Mid band is %lu\n", value[1].dwValue);
    printf("The High band is %lu\n", value[2].dwValue);
}
```

To set the values of all 3 bands, you fill in the values of the MIXERCONTROLDETAILS_UNSIGNED structs. Here is an example of setting the Low band to 31, the Mid band to 0, and the High band to 62.

```c
/* We need 3 MIXERCONTROLDETAILS_UNSIGNED structs to set the
   values of the 3 bands */
MIXERCONTROLDETAILS_UNSIGNED value[3];
MIXERCONTROLDETAILS          mixerControlDetails;
MMRESULT                     err;

mixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);

/* Tell mixerSetControlDetails() which control whose values we
   want to set */
mixerControlDetails.dwControlID = 0x00000002;

/* The values for all channels are the same as the first channel */
mixerControlDetails.cChannels = 1;

/* We're setting all 3 bands */
mixerControlDetails.cMultipleItems = 3;
```

```c
/* Give mixerSetControlDetails() the address of the
   MIXERCONTROLDETAILS_UNSIGNED structs into which we place the values */
mixerControlDetails.paDetails = &value[0];

/* Tell mixerSetControlDetails() how big each MIXERCONTROLDETAILS_UNSIGNED is */
mixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);

/* Store the Low band's value */
value[0].dwValue = 31;

/* Store the Mid band's value */
value[1].dwValue = 0;

/* Store the High band's value */
value[2].dwValue = 62;

/* Set the values of the 3 bands for this control */
if ((err = mixerSetControlDetails((HMIXEROBJ)mixerHandle, &mixerControlDetails,
MIXER_SETCONTROLDETAILSF_VALUE)))
{
    /* An error */
    printf("Error #%d calling mixerSetControlDetails()\n", err);
}
```

Now let's remove that MIXERCONTROL_CONTROLF_UNIFORM flag bit of the MIXERCONTROL. Now, each channel's items have individual values. Since the "Speaker Out" has 2 channels, that means we have a total of 2 (channels) * 3 (items), or 6 values total for this control. Our graphic EQ now looks like this:

Left Channel   **Right Channel**
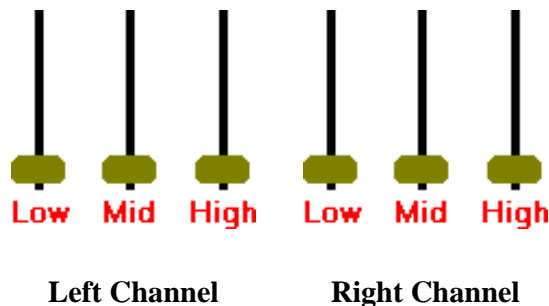
We're going to need 6 MIXERCONTROLDETAILS_UNSIGNED structs to fetch the values of all items in all channels. Oh, and in the preceding example, I assumed what the labels were for those items. What you really should do is ask the Mixer to provide you with the labels if you wish to print them out. To do this, you must supply an array of MIXERCONTROLDETAILS_LISTTEXT structs, just like how you use an array of MIXERCONTROLDETAILS_UNSIGNED structs to fetch the values of all items in all channels

```c
/* We need 6 MIXERCONTROLDETAILS_UNSIGNED structs to retrieve the
   values of this control */
MIXERCONTROLDETAILS_UNSIGNED value[6];
/* We need 6 MIXERCONTROLDETAILS_LISTTEXT structs to retrieve the
   labels of all items */
MIXERCONTROLDETAILS_LISTTEXT label[6];
MIXERCONTROLDETAILS              mixerControlDetails;
MMRESULT                         err;

mixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);
```

```c
/* Tell mixerGetControlDetails() which control whose value we
   want to retrieve */
mixerControlDetails.dwControlID = 0x00000002;

/* Our "Speaker Out" has 2 channels */
mixerControlDetails.cChannels = 2;

/* There are 3 items per channel */
mixerControlDetails.cMultipleItems = 3;

/* Give mixerGetControlDetails() the address of the
   MIXERCONTROLDETAILS_UNSIGNED array into which to return the values */
mixerControlDetails.paDetails = &value[0];

/* Tell mixerGetControlDetails() how big each MIXERCONTROLDETAILS_UNSIGNED is */
mixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_UNSIGNED);

/* Retrieve the current values of all 3 bands for both channels */
if ((err = mixerGetControlDetails((HMIXEROBJ)mixerHandle, &mixerControlDetails,
MIXER_GETCONTROLDETAILSF_VALUE)))
{
    /* An error */
    printf("Error #%d calling mixerGetControlDetails()\n", err);
}
else
{
    unsigned long    i,n;

    /* Let's fetch the labels of all the items */

    mixerControlDetails.cbStruct = sizeof(MIXERCONTROLDETAILS);
    mixerControlDetails.dwControlID = 0x00000002;
    mixerControlDetails.cChannels = 2;
    mixerControlDetails.cMultipleItems = 3;

    /* Give mixerGetControlDetails() the address of the
       MIXERCONTROLDETAILS_LISTTEXT array into which to return the labels */
    mixerControlDetails.paDetails = &label[0];

    /* Tell mixerGetControlDetails() how big each MIXERCONTROLDETAILS_LISTTEXT is */
    mixerControlDetails.cbDetails = sizeof(MIXERCONTROLDETAILS_LISTTEXT);

    /* Retrieve the labels of all items for both channels, Note
       that I specify MIXER_GETCONTROLDETAILSF_LISTTEXT */
    if ((err = mixerGetControlDetails((HMIXEROBJ)mixerHandle, &mixerControlDetails,
MIXER_GETCONTROLDETAILSF_LISTTEXT)))
    {
        /* An error */
        printf("Error #%d calling mixerGetControlDetails()\n", err);
    }
    else
    {
        /* Print the values of all items */
        for (i = 0; i < 2; i++)
        {
            printf("Channel %lu:\n", i+1);
```

```
        for (n = 0; n < 3; n++)
        {
            printf("\tThe %s item is %lu\n", label[3 * i + n].szName, value[3 *
i + n].dwValue);
        }
    }
}
}
```

It is not legal to retrieve or set only some of items of a control. For example, if a control has 8 items, it's not legal to try to retrieve the values of only the first 2 items. You must always retrieve/set the values of all items for all channels simultaneously. The one caveat to this rule is that if you set only the items for the first channel, then mixerSetControlDetails() automatically treats the control as if it was MIXERCONTROL_CONTROLF_UNIFORM. The net result is that the items for all channels get set to the same values as the first channel. So, you can quickly set all channels to the same values just by setting the items of only the first channel.

Notification of changes

In my above examples, I have shown that the mixer device is always opened with mixerOpen() and its handle retrieved for use with other mixer functions. This is not always necessary. In fact, the mixer API has been designed so that, instead of passing a handle to an open mixer wherever any mixer function specifies such an arg, you can instead pass the mixer ID number of the desired mixer. So, you do not need to open a mixer explicitly.

But there are advantages to explicitly opening a mixer (with mixerOpen()) for as long as you need to do operations upon it. First, this prevents the mixer from somehow being "unloaded" (presumably by the audio card's driver). Secondly, when you have a mixer open, you can instruct Windows to send you a special message (to the Window procedure of some window that you've created) whenever any line's state has changed (for example, if the line is muted), or the value of some control has been changed. You get sent such a message not only when you change a line's state or a control's value, but also when any other program also opens that mixer (ie, more than one program can open the same mixer simultaneously) and changes a line or control. So, you can keep your program in sync with any changes made to the mixer by any other program.

When you call mixerOpen(), you specify the handle to your own window which you wish Windows to send its special "mixer messages". Pass your window handle as the third arg to mixerOpen(). Also specify CALLBACK_WINDOW as the last arg.

There are 2 special "mixer messages". MM_MIXM_LINE_CHANGE is sent to your window procedure whenever a line's state is changed. MM_MIXM_CONTROL_CHANGE is sent whenever a control's value is changed.

For MM_MIXM_LINE_CHANGE, the WPARAM argument to your window procedure is the handle of the open mixer whose line has changed. The LPARAM argument is the ID number of the line whose state has changed.

For MM_MIXM_CONTROL_CHANGE, the WPARAM argument to your window procedure is the handle of the open mixer whose line has changed. The LPARAM argument is the ID number of the control whose value has changed.

Conclusion

The Mixer API is one of the most complicated APIs regarding Windows multi-media. It may take awhile to absorb this tutorial and apply it to your needs. But the Mixer API gives you a way to make adjustments to the settings of any sound card without needing to be specifically written for that card.

For more information about structures and APIs about mixers, see Microsoft Developer Network's reference upon audio mixers

Microsoft makes a freely downloadable example of using the Mixer API. But I found this code to be too terse in its use of comments, and also had a lot of code not really related to the Mixer API and not needed. I have pared down this example to mostly code pertinent to the Mixer API, and profusely commented the code. You can download my version of Microsoft's

[Mixer Device Example](#) to show how to display information on all Mixer devices and their lines/controls, as well as adjust controls' values. Included are the Project Workspace files for Visual C++ 4.0, but since it is an ordinary C Windowed app, any Windows C compiler should be able to compile it.

This information is not yet completed.

Windows' MultiMedia (File) I/O (MMIO) functions were created to provide programmers with some convenient routines to help read and write RIFF files, such as WAVE files. (You should read the article upon Interchange File Format before proceeding).

The MMIO functions allow file buffering (ie, much like the standard C functions such as fopen, fread, and fwrite), which is very handy given the fact that RIFF files often involve numerous reads and writes of a few bytes. Furthermore, the Windows functions take care of this file buffering completely transparently. (ie, Your program simply tells Windows to use file buffering, and Windows takes care of all details, including allocating a buffer if you do not wish to supply one yourself).

The MMIO functions allow quick and relatively easy locating, reading, and writing of chunks. That spares your program from having to do a lot of file seeking and parsing.

The MMIO functions can work with files on disks, or file images that have been completely loaded into memory.

Finally, you can install your own "custom I/O" procedures (that are called by Windows MMIO functions such as mmioRead or mmioWrite) to support streaming data to devices other than disks or memory files. In that way, any program that uses MMIO functions to read/write data can be made to transparently stream that data to/from your device.

Opening a RIFF file for reading or writing

NOTE: For the sake of demonstration, the following examples will be shown with a WAVE file, although there are other types of RIFF files to which the MMIO functions are applicable.

The first thing you must do is open a file for reading, writing, or both. This is accomplished by calling mmioOpen(). The first argument is a pointer to a filename.

The second argument is a pointer to an MMIOOPEN structure. This does not need to be supplied unless you desire some special operations such as parsing a file image in memory, or forcing Windows to use a file buffer that you've allocated for I/O buffering, or supplying the address of a custom I/O procedure.

The third argument is some flags OR'ed together. To open a file for reading, specify the flag MMIO_READ. To create a new file to which you'll be writing data, specify the flag MMIO_WRITE. To open a file that you'll be both reading and writing, specify the flag MMIO_READWRITE.

The MMIO_EXCLUSIVE, MMIO_DENYWRITE, and MMIO_DENYREAD flags are file-sharing flags. For example, to prevent another program from simultaneously writing to the file while your program has it open, specify MMIO_DENYWRITE. MMIO_EXCLUSIVE denies other programs read or write access simultaneously. MMIO_DENYREAD prevents other programs from reading (but not writing) to the file simultaneously

Specify the MMIO_ALLOCBUF flag if you wish Windows to allocate a buffer for buffered file I/O. The default size will be 8K, but if you wish a different size, then you must supply a MMIOOPEN structure with its cchBuffer field set to the desired buffer size in bytes. If cchBuffer is 0, the default buffer size is used. (If you are providing your own I/O buffer, do not set the MMIO_ALLOCBUF flag).

There are some other flags that will be discussed later.

If successful, mmioOpen() returns a handle to the open file. This handle can be used only with MMIO functions.

So, to open the file C:\Windows\Chord.wav for reading:

```
HMMIO hmmio;

/* Open the file for reading with buffered I/O. Let windows use its default internal
```

```
buffer */
if (!(hmmio = mmioOpen("C:\\WINDOWS\\CHORD.WAV", 0, MMIO_READ|MMIO_ALLOCBUF)))
{
        printf("An error opening the file!\n");
}
```

Locating and reading the Group Header

The first thing that you'll want to do after you open a file for reading is to check that it contains the desired type of data within it. For example, let's say that you want WAVE data. To check that there is a WAVE format within the open file (ie, it could be a collection of various types of data -- a CAT or LIST), and to locate the file pointer to the Group header (ie, the WAVE's RIFF header) and read in that Group Header, you need to allocate a "chunk information" (MMCKINFO) structure, set its fccType field to the ID that you wish to seek out (here it's 'WAVE'), and pass that structure to mmioDescend() with the flag MMIO_FINDRIFF. mmioDescend() then dives into the file and seeks to that RIFF WAVE header (if there is one in the file) and reads it in. (Windows docs refer to reading a chunk as "descending" into it).

Note that there is an MMIO function, mmioFOURCC(), to help you initialize the fccType field with an IFF ID. An IFF ID is 4 characters, but treated as one unsigned long in Motorola (big endian) order.

Here is an example of this:

```
MMCKINFO mmckinfoParent;    /* for the Group Header */

/* Tell Windows to locate a WAVE Group header somewhere in the file, and read it in.
 This marks the start of any embedded WAVE format within the file */
mmckinfoParent.fccType = mmioFOURCC('W', 'A', 'V', 'E');
if (mmioDescend(hmmio, (LPMMCKINFO)&mmckinfoParent, 0, MMIO_FINDRIFF))
{
        /* Oops! No embedded WAVE format within this file */
        printf("ERROR: This file doesn't contain a WAVE!\n");
}
else
{
        /* Here you may seek to, and read in other chunks */
}
```

Locating and reading a chunk

After you locate to the Group Header, the next thing that you'll likely want to do is read in the chunks of that group. For example, with the WAVE format, you'll at least want to read in the **fmt** chunk to get information about the WAVE, and then locate to the **data** chunk and read in the waveform data.

You locate the file pointer to a chunk header in much the same way as locating to a Group header. But, you'll need a second "chunk information" structure. You set its fccType field to the chunk ID that you wish to seek out (here we'll seek out a 'fmt ' chunk), and pass that structure to mmioDescend() with the flag MMIO_FINDCHUNK, and also pass the MMCKINFO that you filled in above (with the Group Header). mmioDescend() then dives into the file and seeks to that chunk's header (if there is one in the file), and read it in. (NOTE: Only the 8 byte chunk header is read in -- not the chunk's data).

Here is an example of reading the fmt chunk header:

```
MMCKINFO mmckinfoSubchunk;    /* for finding chunks within the Group */
```

```
/* Tell Windows to locate the WAVE's "fmt " chunk and read in its header */
mmckinfoSubchunk.ckid = mmioFOURCC('f', 'm', 't', ' ');
if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent, MMIO_FINDCHUNK))
{
        /* Oops! The required fmt chunk was not found! */
        printf("ERROR: Required fmt chunk was not found!\n");
}
else
{
        /* Here you may read in the chunk's data */
}
```

The mmckinfoSubchunk structure's cksize field now contains the chunk's chunkSize.

Reading a chunk's data

After reading in a chunk's header as above, you can then read in its data using the mmioRead() function.

It just so happens that a WAVEFORMATEX structure can hold all the data in WAVE's fmt chunk verbatim. So, you need to make only one call to mmioRead() to load in that chunk's data, as so:

```
WAVEFORMATEX waveFormat;    /* for reading a fmt chunk's data */

/* Tell Windows to read in the "fmt " chunk into a WAVEFORMATEX structure */
if (mmioRead(hmmio, (HPSTR)&waveFormat, mmckinfoSubchunk.cksize) != (LRESULT)
mmckinfoSubchunk.cksize)
{
        /* Oops! */
        printf("ERROR: reading the fmt chunk!\n");
}
```

Ascending out of a chunk

After you've used mmioDescend() to locate and read in a chunk's header and/or data, if you plan to read any other chunks in the file, you need to "ascend" out of any chunk that you've mmioDescend()'ed into. You can't mmioDescend() into another chunk without first ascending out of any chunk you're currently reading. You use the mmioAscend() function to do so. You pass it the MMCKINFO that you used to mmioDescend() into the current chunk.

For example, to ascend out of that fmt chunk whose data we were reading above:

```
mmioAscend(hmmio, &mmckinfoSubchunk, 0);
```

The purpose of this function is to allow the mmio functions to do some necessary, internal bookkeeping with the file position pointer inbetween you moving it from chunk to chunk via mmioDescend().

You can then use the same MMCKINFO with mmioDescend() to locate/read another chunk. (ie, You don't need a separate MMCKINFO with every chunk that you wish to read).

Closing a file

After you're done reading/writing to a file, you must close it. You do this by passing the handle which you obtained from mmioOpen() to the function mmioClose() as so:

```
mmioClose(hmmio, 0);
```

You can download my WaveParse C example to show how to use the MMIO functions to open, read, and display some information about a WAVE file. Included are the Project Workspace files for Visual C++ 4.0, but since it is a console app, any Windows C compiler should be able to compile it. Remember that all apps should include MMSYSTEM.H and link with WINMM.LIB (or MMSYSTEM.LIB if Win3.1). This is a ZIP archive. Use an unzip utility that supports long filenames.

```
;; Windows 2000 MPU-401 audio inf file for WDM core files.

[Version]
Signature="$CHICAGO$"
Class=MEDIA
ClassGUID={4d36e96c-e325-11ce-bfc1-08002be10318}
provider=%MSFT%
LayoutFile=layout.inf, layout1.inf, layout2.inf
DriverVer =08/24/1999,5.00.2113.1

;; Device Callouts for swenumed devices
[Manufacturer]
%MfgName%=Microsoft

[Microsoft]
%WDM_KMIXER.DeviceDesc%  = WDM_KMIXER,   SW\{B7EAFDC0-A680-11D0-96D8-00AA0051E51D}
%WDM_SYSAUDIO.DeviceDesc% = WDM_SYSAUDIO, SW\{A7C7A5B0-5AF3-11D1-9CED-00A024BF0407}
%WDM_SWMIDI.DeviceDesc%  = WDM_SWMIDI,   SW\{6C1B9F60-C0A9-11D0-96D8-00AA0051E51D}
%WDM_WDMAUD.DeviceDesc%  = WDM_WDMAUD,   SW\{CD171DE3-69E5-11D2-B56D-0000F8754380}
%WDM_DMUSIC.DeviceDesc%  = WDM_DMUSIC,   SW\{8C07DD50-7A8D-11d2-8F8C-00C04FBF8FEF}
%*PNPB006.DeviceDesc%     = MS_MPU401, *PNPB006

;; WDM Audio Registration Section can be called by OEM's to install a WDM driver
[WDMAUDIO.Registration]
AddReg   = DeviceRegistration
CopyFiles = WDM.CopyFiles.Drv, WDM.CopyFiles.Sys

;;for Win98 DDK backward compatibility
[WDMAUDIO.Registration.NT]
AddReg   = DeviceRegistration
CopyFiles = WDM.CopyFiles.Drv, WDM.CopyFiles.Sys

;; WDM Audio Install Section to be called once when system is installed
[WDMAUDIO.Install]

[DeviceRegistration]
;; Sysaudio swenum install
HKLM,%RunOnce%,"WDM_SYSAUDIO",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_SYSAUDIO.DeviceId%,%KSNAME_Filter%,%KSCATEGORY_SYSAUDIO%,%17%\WDMAUDIO.inf,WDM_SYSAUDIO.Interface.Install"

;; Kmixer swenum install
HKLM,%RunOnce%,"WDM_KMIXER0",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_KMIXER.DeviceId%,%KSNAME_Filter%,%KSCATEGORY_MIXER%,%17%\WDMAUDIO.inf,WDM_KMIXER.Interface.Install"
HKLM,%RunOnce%,"WDM_KMIXER1",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_KMIXER.DeviceId%,%KSNAME_Filter%,%KSCATEGORY_AUDIO%,%17%\WDMAUDIO.inf,WDM_KMIXER.Interface.Install"

;; SWMidi swenum install
HKLM,%RunOnce%,"WDM_SWMIDI0",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_SWMIDI.DeviceId%,%KSNAME_Filter%,%KSCATEGORY_DATATRANSFORM%,%17%\WDMAUDIO.inf,WDM_SWMIDI.Interface.Install"
HKLM,%RunOnce%,"WDM_SWMIDI1",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_SWMIDI.DeviceId%,%KSNAME_Filter%,%KSCATEGORY_SYNTHESIZER%,%17%\WDMAUDIO.inf,WDM_SWMIDI.Interface.Install"
HKLM,%RunOnce%,"WDM_SWMIDI2",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_SWMIDI.DeviceId%,%KSNAME_Filter%,%KSCATEGORY_AUDIO%,%17%\WDMAUDIO.inf,WDM_SWMIDI.Interface.Install"

;; DMusic swenum install
HKLM,%RunOnce%,"WDM_DMUSIC0",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_DMUSIC.DeviceId%,%KSNAME_DMUSIC%,%KSCATEGORY_DATATRANSFORM%,%17%\WDMAUDIO.inf,WDM_DMUSIC.Interface.Install"
HKLM,%RunOnce%,"WDM_DMUSIC1",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_DMUSIC.DeviceId%,%KSNAME_DMUSIC%,%KSCATEGORY_SYNTHESIZER%,%17%\WDMAUDIO.inf,WDM_DMUSIC.Interface.Install"
HKLM,%RunOnce%,"WDM_DMUSIC2",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_DMUSIC.DeviceId%,%KSNAME_DMUSIC%,%KSCATEGORY_AUDIO%,%17%\WDMAUDIO.inf,WDM_DMUSIC.Interface.Install"

;; WDMAud swenum install
HKLM,%RunOnce%,"WDM_WDMAUD",,"rundll32.exe streamci.dll,StreamingDeviceSetup %WDM_WDMAUD.DeviceId%,%KSNAME_Filter%,%KSCATEGORY_WDMAUD%,%17%\WDMAUDIO.inf,WDM_WDMAUD.Interface.Install"

;; copied to system32\drivers
[WDM.CopyFiles.Drv]
```

```
  portcls.sys
  stream.sys


;; copied to system
[WDM.CopyFiles.Sys]
wdmaud.drv
ksuser.dll


;; MPU401 Install Section can be called by OEM's to install MS MPU401 driver
[WDMAUDIO.MPU401]
CopyFiles  = MS_MPU401.CopyFiles
AddReg     = MS_MPU401.AddReg
UpdateInis = MS_MPU401.UpdInis
DelReg     = MS_MPU401.DelRegs
DelFiles   = MS_MPU401.DelKernFiles, MS_MPU401.DelUserFiles


[MS_MPU401.CopyFiles]
msmpu401.sys


[MS_MPU401.AddReg]
HKR,,AssociatedFilters,,"wdmaud"
HKR,,Driver,,msmpu401.sys
HKR,,NTMPDriver,,msmpu401.sys

HKR,Drivers,SubClasses,,"midi"
HKR,Drivers\midi\wdmaud.drv, Driver,,wdmaud.drv
HKR,Drivers\midi\wdmaud.drv, Description,,%MS_MPU401%


[MS_MPU401.UpdInis]
system.ini,drivers32,"midi*=mpu401.dll",,1


[MS_MPU401.DelRegs]
HKLM,SYSTEM\CurrentControlSet\Enum\Root\LEGACY_MPU401
HKLM,SYSTEM\CurrentControlSet\Services\mpu401


[MS_MPU401.DelKernFiles]
mpu401.sys,,,1


[MS_MPU401.DelUserFiles]
mpu401.dll,,,1


[WDMAUDIO.MPU401.Services]
AddService = ms_mpu401,0x00000002, ms_mpu401_Service_Inst


[ms_mpu401_Service_Inst]
DisplayName   = %ms_mpu401.SvcDesc%
ServiceType   = 1              ; SERVICE_KERNEL_DRIVER
StartType     = 3              ; SERVICE_DEMAND_START
ErrorControl  = 1              ; SERVICE_ERROR_NORMAL
ServiceBinary = %10%\system32\drivers\msmpu401.sys


;; FDMA Install Section can be called by OEM's to turn on FDMA
[FDMA.AddReg]
;; nothing to add for NT (yet)
[WDMAUDIO.FDMA]
AddReg=FDMA.AddReg


;; Generic MPU401 driver install
[MS_MPU401]
```

```
    Include=ks.inf
    Needs=KS.Registration, WDMAUDIO.Registration, WDMAUDIO.MPU401
    LogConfig  = MS_MPU401.LogConfig

    [MS_MPU401.FactDef]
    ConfigPriority=NORMAL
    IOConfig=330-331
    IRQConfig=9

    [MS_MPU401.LogConfig]
    ConfigPriority = HARDRECONFIG
    IOConfig=2@200-271%fff0(3ff::),300-301,310-311,320-321,330-331,332-333,334-335,336-337,340-341,350-351,360-361,370-371,390-391,3A0-3A1,3D0-3D1,3E0-3E1
    IRQConfig=3, 4, 5, 7, 9

    [MS_MPU401.Interfaces]
    AddInterface = %KSCATEGORY_AUDIO%,%KSNAME_UART%,WDM.Interface.UART

    [WDM.Interface.UART]
    AddReg=WDM.Interface.UART.AddReg
    [WDM.Interface.UART.AddReg]
    HKR,,CLSID,,%Proxy.CLSID%
    HKR,,FriendlyName,,%WDM.UART.szPname%

    [MS_MPU401.Services]
    Needs=WDMAUDIO.MPU401.Services

    ;; System Audio Install
    [WDM_SYSAUDIO.Interface.Install]
    AddReg=WDM_SYSAUDIO.Interface.AddReg

    [WDM_SYSAUDIO.Interface.AddReg]
    HKR,,CLSID,,%Proxy.CLSID%
    HKR,,FriendlyName,,%WDM_SYSAUDIO.FriendlyName%

    [WDM_SYSAUDIO]
    CopyFiles = WDM_SYSAUDIO.CopyFiles

    [WDM_SYSAUDIO.CopyFiles]
    sysaudio.sys

    [WDM_SYSAUDIO.Services]
    AddService = sysaudio,0x00000002, sysaudio_Service_Inst

    [sysaudio_Service_Inst]
    DisplayName   = %sysaudio.SvcDesc%
    ServiceType   = 1              ; SERVICE_KERNEL_DRIVER
    StartType     = 3              ; SERVICE_DEMAND_START
    ErrorControl  = 1              ; SERVICE_ERROR_NORMAL
    ServiceBinary = %10%\system32\drivers\sysaudio.sys

    ;; Kernel Mixer Install
    [WDM_KMIXER.Interface.Install]
    AddReg=WDM_KMIXER.Interface.AddReg

    [WDM_KMIXER.Interface.AddReg]
    HKR,,CLSID,,%Proxy.CLSID%
    HKR,,FriendlyName,,%WDM_KMIXER.FriendlyName%

    [WDM_KMIXER]
```

```
    CopyFiles = WDM_KMIXER.CopyFiles

    [WDM_KMIXER.CopyFiles]
    kmixer.sys

    [WDM_KMIXER.Services]
    AddService = kmixer,  0x00000002, kmixer_Service_Inst

    [kmixer_Service_Inst]
    DisplayName    = %kmixer.SvcDesc%
    ServiceType   = 1              ; SERVICE_KERNEL_DRIVER
    StartType     = 3              ; SERVICE_DEMAND_START
    ErrorControl  = 1              ; SERVICE_ERROR_NORMAL
    ServiceBinary = %10%\system32\drivers\kmixer.sys

    ;; Software MIDI Install
    [WDM_SWMIDI.Interface.Install]
    AddReg=WDM_SWMIDI.Interface.AddReg

    [WDM_SWMIDI.Interface.AddReg]
    HKR,,CLSID,,%Proxy.CLSID%
    HKR,,FriendlyName,,%WDM_SWMIDI.FriendlyName%

    [WDM_SWMIDI]
    CopyFiles = WDM_SWMIDI.CopyFiles

    [WDM_SWMIDI.CopyFiles]
    swmidi.sys

    [WDM_SWMIDI.Services]
    AddService = swmidi,0x00000002, swmidi_Service_Inst

    [swmidi_Service_Inst]
    DisplayName    = %swmidi.SvcDesc%
    ServiceType   = 1              ; SERVICE_KERNEL_DRIVER
    StartType     = 3              ; SERVICE_DEMAND_START
    ErrorControl  = 1              ; SERVICE_ERROR_NORMAL
    ServiceBinary = %10%\system32\drivers\swmidi.sys

    ;; DMusic Install
    [WDM_DMUSIC.Interface.Install]
    AddReg=WDM_DMUSIC.Interface.AddReg

    [WDM_DMUSIC.Interface.AddReg]
    HKR,,CLSID,,%Proxy.CLSID%
    HKR,,FriendlyName,,%WDM_DMUSIC.FriendlyName%

    [WDM_DMUSIC]
    CopyFiles = WDM_DMUSIC.CopyFiles

    [WDM_DMUSIC.CopyFiles]
    DMusic.sys

    [WDM_DMUSIC.Services]
    AddService = DMusic, 0x00000002, DMus_Synth_Service_Inst

    [DMus_Synth_Service_Inst]
    DisplayName    = %WDM_DMUSIC.SvcDesc%
    ServiceType   = 1              ; SERVICE_KERNEL_DRIVER
```

```
 StartType    = 3              ; SERVICE_SYSTEM_START
 ErrorControl = 1              ; SERVICE_ERROR_NORMAL
 ServiceBinary = %10%\system32\drivers\DMusic.sys

 ;; WDMAUD Install
 [WDM_WDMAUD.Interface.Install]
 AddReg=WDM_WDMAUD.Interface.AddReg

 [WDM_WDMAUD.Interface.AddReg]
 HKR,,CLSID,,%Proxy.CLSID%
 HKR,,FriendlyName,,%WDM_WDMAUD.FriendlyName%

 [WDM_WDMAUD]
 CopyFiles = WDM_WDMAUD.CopyFiles

 [WDM_WDMAUD.CopyFiles]
 wdmaud.sys

 [WDM_WDMAUD.Services]
 AddService = wdmaud,0x00000002, wdmaud_Service_Inst

 [wdmaud_Service_Inst]
 DisplayName   = %wdmaud.SvcDesc%
 ServiceType   = 1              ; SERVICE_KERNEL_DRIVER
 StartType     = 3              ; SERVICE_DEMAND_START
 ErrorControl  = 1              ; SERVICE_ERROR_NORMAL
 ServiceBinary = %10%\system32\drivers\wdmaud.sys

 ;; Destination Directory List
 [DestinationDirs]
 WDM.CopyFiles.Drv    = 10,system32\drivers ; %SystemRoot%\system32\drivers
 WDM.CopyFiles.Sys    = 11              ; %SystemRoot%\system

 MS_MPU401.CopyFiles   = 10,system32\drivers ; %SystemRoot%\system32\drivers

 WDM_SYSAUDIO.CopyFiles = 10,system32\drivers
 WDM_KMIXER.CopyFiles  = 10,system32\drivers
 WDM_SWMIDI.CopyFiles  = 10,system32\drivers
 WDM_WDMAUD.CopyFiles  = 10,system32\drivers
 WDM_DMUSIC.CopyFiles  = 10,system32\drivers

 MS_MPU401.DelKernFiles = 10,system32\drivers
 MS_MPU401.DelUserFiles = 10,system32

 ;; NT Class Install
 [ClassInstall32.NT]
 Addreg    = ClassInstall.AddReg
 UpdateInis = ClassInstall.DelOldInis

 [ClassInstall.AddReg]
 HKR,,,,%MediaClassName%
 HKR,,Installer32,,"MmSys.Cpl,MediaClassInstaller"
 HKR,,EnumPropPages32,,"MmSys.Cpl,MediaPropPageProvider"
 HKR,,TroubleShooter-0,,"tshoot.chm,hdw_multi.htm"
 HKR,,Icon,,"3004"

 [ClassInstall.DelOldInis]
 system.ini,drivers32,"wave*=sndblst.dll",,1
 system.ini,drivers32,"midi*=sndblst.dll",,1
```

```
system.ini,drivers32,"mixer*=sndblst.dll",,1
system.ini,drivers32,"aux*=sndblst.dll",,1

system.ini,drivers32,"wave*=sb16snd.dll",,1
system.ini,drivers32,"midi*=sb16snd.dll",,1
system.ini,drivers32,"mixer*=sb16snd.dll",,1
system.ini,drivers32,"aux*=sb16snd.dll",,1

system.ini,drivers32,"wave*=sbawe32.drv",,1
system.ini,drivers32,"midi*=sbawe32.drv",,1
system.ini,drivers32,"mixer*=sbawe32.drv",,1
system.ini,drivers32,"aux*=sbawe32.drv",,1

system.ini,drivers32,"wave*=auddrive.dll",,1
system.ini,drivers32,"midi*=auddrive.dll",,1
system.ini,drivers32,"mixer*=auddrive.dll",,1
system.ini,drivers32,"aux*=auddrive.dll",,1

system.ini,drivers32,"wave*=sndsys32.dll",,1
system.ini,drivers32,"midi*=sndsys32.dll",,1
system.ini,drivers32,"mixer*=sndsys32.dll",,1
system.ini,drivers32,"aux*=sndsys32.dll",,1

system.ini,drivers32,"wave*=opl3sa32.dll",,1
system.ini,drivers32,"midi*=opl3sa32.dll",,1
system.ini,drivers32,"mixer*=opl3sa32.dll",,1
system.ini,drivers32,"aux*=opl3sa32.dll",,1

system.ini,drivers32,"wave*=maestro.dll",,1
system.ini,drivers32,"midi*=maestro.dll",,1
system.ini,drivers32,"mixer*=maestro.dll",,1
system.ini,drivers32,"aux*=maestro.dll",,1

system.ini,drivers32,"wave*=sndglxy.dll",,1
system.ini,drivers32,"midi*=sndglxy.dll",,1
system.ini,drivers32,"mixer*=sndglxy.dll",,1
system.ini,drivers32,"aux*=sndglxy.dll",,1

system.ini,drivers32,"wave*=ydsxg32.dll",,1
system.ini,drivers32,"midi*=ydsxg32.dll",,1
system.ini,drivers32,"mixer*=ydsxg32.dll",,1
system.ini,drivers32,"aux*=ydsxg32.dll",,1

system.ini,drivers32,"wave*=adisound.dll",,1
system.ini,drivers32,"midi*=adisound.dll",,1
system.ini,drivers32,"mixer*=adisound.dll",,1
system.ini,drivers32,"aux*=adisound.dll",,1

system.ini,drivers32,"wave*=essaudio.dll",,1
system.ini,drivers32,"midi*=essaudio.dll",,1
system.ini,drivers32,"mixer*=essaudio.dll",,1
system.ini,drivers32,"aux*=essaudio.dll",,1

system.ini,drivers32,"wave*=cs32ba11.dll",,1
system.ini,drivers32,"midi*=cs32ba11.dll",,1
system.ini,drivers32,"mixer*=cs32ba11.dll",,1
system.ini,drivers32,"aux*=cs32ba11.dll",,1

system.ini,drivers32,"wave*=cwbaudll.dll",,1
```

```
system.ini,drivers32,"midi*=cwbaudll.dll",,1
system.ini,drivers32,"mixer*=cwbaudll.dll",,1
system.ini,drivers32,"aux*=cwbaudll.dll",,1

system.ini,drivers32,"wave*=alsnd32.dll",,1
system.ini,drivers32,"midi*=alsnd32.dll",,1
system.ini,drivers32,"mixer*=alsnd32.dll",,1
system.ini,drivers32,"aux*=alsnd32.dll",,1

system.ini,drivers32,"wave*=es1371.dll",,1
system.ini,drivers32,"midi*=es1371.dll",,1
system.ini,drivers32,"mixer*=es1371.dll",,1
system.ini,drivers32,"aux*=es1371.dll",,1

system.ini,drivers32,"wave*=ctmm32.dll",,1
system.ini,drivers32,"midi*=ctmm32.dll",,1
system.ini,drivers32,"mixer*=ctmm32.dll",,1

system.ini,drivers32,"midi*=ctsyn32.dll",,1


system.ini, driver32, "wave*=cwcmmsys.dll",,1
system.ini, driver32, "midi*=cwcmmsys.dll",,1
system.ini, driver32, "mixer*=cwcmmsys.dll",,1
system.ini, driver32, "midi* =cwcfm3.dll",,1


system.ini,drivers32,"wave*=aztaud.dll",,1
system.ini,drivers32,"midi*=aztaud.dll",,1
system.ini,drivers32,"mixer*=aztaud.dll",,1
system.ini,drivers32,"aux*=aztaud.dll",,1


system.ini,drivers32,"wave*=audpci40.dll",,1
system.ini,drivers32,"midi*=audpci40.dll",,1
system.ini,drivers32,"mixer*=audpci40.dll",,1
system.ini,drivers32,"aux*=audpci40.dll",,1


system.ini,drivers32,"wave*=au88nt32.dll",,1
system.ini,drivers32,"midi*=au88nt32.dll",,1
system.ini,drivers32,"mixer*=au88nt32.dll",,1
system.ini,drivers32,"aux*=au88nt32.dll",,1

system.ini,drivers32,"wave*=au88nt30.dll",,1
system.ini,drivers32,"midi*=au88nt30.dll",,1
system.ini,drivers32,"mixer*=au88nt30.dll",,1
system.ini,drivers32,"aux*=au88nt30.dll",,1


system.ini,drivers32,"wave*=waveart.dll",,1
system.ini,drivers32,"midi*=waveart.dll",,1
system.ini,drivers32,"mixer*=waveart.dll",,1
system.ini,drivers32,"aux*=waveart.dll",,1

[Strings]
;Non-localized strings
;; ???? These keys weren't used anywhere. Delete?
;;KEY_WARNVER="SYSTEM\CurrentControlSet\Control\SessionManager\WarnVerDLLs"
```

```
;;KEY_CHKVER="SYSTEM\CurrentControlSet\Control\SessionManager\CheckVerDLLs"
;;KEY_FIL="SYSTEM\CurrentControlSet\Control\MediaResources\FilterList"
KEY_IOC="SYSTEM\CurrentControlSet\Control\MediaResources\InstallOptionalComponents"
MSFT="Microsoft"
MfgName="Microsoft"

RunOnce = "SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce"
Proxy.CLSID        = "{17CCA71B-ECD7-11D0-B908-00A0C9223196}"
KSNAME_Filter      = "{9B365890-165F-11D0-A195-0020AFD156E4}"
KSNAME_DMUSIC      = "dmusic"

WDM_SYSAUDIO.DeviceId = "{A7C7A5B0-5AF3-11D1-9CED-00A024BF0407}"
KSCATEGORY_SYSAUDIO   = "{A7C7A5B1-5AF3-11D1-9CED-00A024BF0407}"

WDM_KMIXER.DeviceId   = "{B7EAFDC0-A680-11D0-96D8-00AA0051E51D}"
KSCATEGORY_MIXER      = "{AD809C00-7B88-11D0-A5D6-28DB04C10000}"

WDM_SWMIDI.DeviceId   = "{6C1B9F60-C0A9-11D0-96D8-00AA0051E51D}"
WDM_DMUSIC.DeviceId   = "{8C07DD50-7A8D-11d2-8F8C-00C04FBF8FEF}"

KSCATEGORY_DATATRANSFORM = "{2EB07EA0-7E70-11D0-A5D6-28DB04C10000}"
KSCATEGORY_SYNTHESIZER   = "{DFF220F3-F70F-11D0-B917-00A0C9223196}"
KSCATEGORY_AUDIO         = "{6994AD04-93EF-11D0-A3CC-00A0C9223196}"

WDM_WDMAUD.DeviceId = "{CD171DE3-69E5-11D2-B56D-0000F8754380}"
KSCATEGORY_WDMAUD   = "{3E227E76-690D-11D2-8161-0000F8775BF1}"

;Localized strings
MediaClassName="Sound, video and game controllers"
*PNPB006.DeviceDesc = "MPU-401 Compatible MIDI Device"
MS_MPU401="MPU401"
WDM.UART.szPname="MPU-401"
KSNAME_UART="UART"

WDM_SYSAUDIO.DeviceDesc   = "Microsoft Kernel System Renderer"
WDM_SYSAUDIO.FriendlyName = "Microsoft Kernel System Renderer"
sysaudio.SvcDesc          = "Microsoft System Audio Device"

WDM_KMIXER.DeviceDesc     = "Microsoft Kernel Audio Mixer"
WDM_KMIXER.FriendlyName   = "Microsoft Kernel Audio Mixer"
kmixer.SvcDesc            = "Microsoft Kernel Wave Audio Mixer"

WDM_SWMIDI.DeviceDesc     = "Microsoft Kernel GS Wavetable Synthesizer"
WDM_SWMIDI.FriendlyName   = "Microsoft Kernel GS Wavetable Synthesizer"
swmidi.SvcDesc            = "Microsoft Kernel GS Wavetable Synthesizer"

WDM_DMUSIC.DeviceDesc     = "Microsoft DirectMusic SW Synth (WDM)"
WDM_DMUSIC.FriendlyName   = "Microsoft DirectMusic SW Synth (WDM)"
WDM_DMUSIC.SvcDesc        = "Microsoft DirectMusic SW Synth (WDM)"

WDM_WDMAUD.DeviceDesc     = "Microsoft WINMM WDM Audio Compatibility Driver"
WDM_WDMAUD.FriendlyName   = "Microsoft WINMM WDM Audio Compatibility Driver"
wdmaud.SvcDesc            = "Microsoft WINMM WDM Audio Compatibility Driver"

ms_mpu401.SvcDesc         = "Microsoft MPU-401 MIDI UART Driver"
```

RAP-10

Output Channels	2 at 44.1, 22.05, or 11.025kHz, 16 or 8 bit. (Any combination of the above, for example play 1 channel at 22.05kHz 8-bit, and the other at 44.1kHz 16-bit).
Output Rate/ Resolution	44.1 kHz (max.), 16-bit
Digital Input	Stereo jack, 44.1, 22.05, or 11.025kHz, 16 or 8 bit. (Any combination of the above, for example, record 1 channel at 22.05kHz 8-bit, and the other at 44.1kHz 16-bit simultaneously).
MIDI Synth	WaveTable synth, GM Patches (128) in ROM. 6 Drum kits. 26 voice polyphony. (Digital audio has its own 2 voices). 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	No information
Custom Sample Upload	None
Effects	Digital reverb/delay. Chorus.
Connections	Joystick (MIDI), stereo line/mic in, stereo aux in, stereo line out, stereo headphone. All jacks are mini plugs. CD audio internal connector.
System Req.	1 port, 1 IRQ, 2 DMA channels.
Driver Support	Windows 3.x. (will also work on Win 95/98), Window NT/2000 (without digital audio -- GM Synth and MIDI I/O only, via MPU-401 driver)
Compatibility	MPU-401 (UART). Windows Drivers.
Street Price (U.S.)	$249

The RAP-10 is essentially the SCC-1's MIDI Synth (first 128 patches only, like the SCD-10) with 2 tracks of CD-quality digital audio added. As such, it sports the much-acclaimed Roland Sound Canvas waveform set as found in cards like the SCC-1, SCD-10, SCD-15, and SC-7 (but not the newer Sound Canvasses such as the SC-88, which have newer waveform sets and improved playback circuitry). The RAP-10 does not support replacing any of its GM waveforms.

The RAP-10 has a digital reverb (with choices of various rooms or halls, decay time, and level) which can alternately be used as a delay (with repeats and delay time adjustable). Furthermore, the RAP-10 has a chorus (with feedback, delay, rate and depth). These effects can be applied both to the MIDI Synth Parts as well as digital audio (with separate reverb/delay and chorus levels for each Part and each of the two tracks).

In fact, the RAP-10's digital audio tracks each have a number of settings including tuning (in both half steps and cents), panning, reverb level, chorus level, volume, brightness, and sample rate. For playback, these can be adjusted via System Exclusive messages (ie, so any software that supports

System Exclusive can be used to adjust all of these parameters) even during digital audio playback. In essense, the RAP-10 allows a lot of the realtime control over digital audio tracks that it allows for one of its GM MIDI Synth Parts (but by using SysEx instead of Controller messages).

The MIDI Synth Parts understand and respond to Channel Pressure (which can be used to control Brightness, Volume, Vibrato Depth, and/or Vibrato Rate, with each Part responding to Pressure in any or no combination of the above ways, independently of other Parts), Modulation (which can be used to control Vibrato Speed and/or Depth), Volume Controller, Effect1 Controller (ie, reverb level), Effect3 Controller (ie, chorus level), Hold Pedal Controller, Expression Controller, Pan Controller, Pitch Bend, Program Change, Reset All Controllers, All Sounds Off, and All Notes Off.

Furthermore, using Registered Parameter Number Controller, each Part's Pitch Bend Range, as well as tuning (in half steps and cents) can be controlled. Using Non-registered Parameter Number Controller, you can control the pan, volume, tuning, chorus level, and reverb level for each drum sound in a kit (in addition to being able to control the entire kit's overall volume, pan, chorus level, and reverb level).

The RAP-10 has a master Volume and master tuning (ie, for all 16 Parts and 2 digital audio tracks) in both half steps and cents.

The RAP-10 ships with a Windows MCI driver that will work under Win95, and also some software that graphically mimics a multi-track mixer for recording/playback. The bundled program is more geared for hobbyist and multi-media work than pro musician's use. The RAP-10 can be used with pro software such as CakeWalk, SAW, Sound Forge, or other Windows titles, utilizing both its MIDI Synth and digital audio simultaneously. You can also use OS/2's RTMIDI MPU-401 driver, but this will not enable digital audio tracks (ie, GM sound module control only). (You could even use Win95's MPU-401 driver, but this also won't enable digital audio tracks).

In order to obtain the actual MIDI In/Out/Thru jacks, you need to attach a "breakout box" to the joystick port. The joystick port is compatible with the Sound Blaster, so any breakout box for the SB will suffice (ie, you don't need the overpriced MCB-10).

SC-7

Output Channels	2
Output Rate/ Resolution	16-bit @ 44.1KHz
Digital Input	None
MIDI Synth	WaveTable synth, GM Patches (128) in ROM. 6 Drum kits. 24 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	No information

Custom Sample Upload	None
Effects	Digital reverb/delay. Chorus.
Connections	COM port, MIDI In/Out, stereo line outs (RCA jacks), headphone (mini), stereo line inputs (mini), AC power (uses a wall-wart).
Controls	Volume knob.
System Req.	Attaches to computer's serial port.
Driver Support	Windows 3.x., Windows 95/98 (if you use the Roland Win95/98 serial driver available on Roland's web site).
Compatibility	Windows Drivers.
Street Price (U.S.)	$207

The SC-7 is essentially the SCC-1's MIDI Synth (first 128 patches only, like the SCD-10) in an external box that attaches to the computer's serial port. (It can attach to both the IBM and Mac serial ports, and instructions are given on using the SC-7 with either. This is a cheap way for Mac owners to get a nice, low cost Roland GM sound module). As such, it sports the much-acclaimed Roland Sound Canvas waveform set as found in cards like the SCC-1, SCD-10, SCD-15, and RAP-10 (but not the newer Sound Canvasses such as the SC-88, which have newer waveform sets and improved playback circuitry). The SC-7 does not support replacing any of its GM waveforms, nor does it record/play digital audio.

See the RAP-10 description for remarks about the SC-7's MIDI Synth engine. It's the same as the RAP-10's, so all RAP-10 remarks (except as concern digital audio tracks) are pertinent (ie, its effects and support for MIDI messages is the same).

The SC-7 is a complete, standalone unit. It can be used without the computer. It has its own power supply and case. An external keyboard (or other MIDI controller) can be plugged directly into the SC-7 MIDI In to play it as a 16-Part multi-timbral module.

The SC-7 ships with a Windows MCI driver, and also some software (Ballade, a sequencer, and Band in a Box). It can be used with pro sequencers such as CakeWalk.

Furthermore, because the SC-7 offers MIDI In and Out jacks to which you can attach even more MIDI gear, it becomes a MIDI interface between your computer and all other external gear. This makes the SC-7 nice for laptop use since it allows connecting external MIDI gear via a clone or Mac serial port, is a 16-part multi-timbral unit also, and can even be used without the computer. It's very small, light, and therefore portable.

SCC-1 (SCC-1B)

Output Channels	2
Output Rate/ Resolution	16-bit @ 44.1KHz
Digital Input	None
MIDI Synth	WaveTable synth, GS Patches (354) in ROM. 9 Drum kits. 24 voice polphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	4 MEG (compressed) ROM
Custom Sample Upload	None
Effects	Digital reverb/delay. Chorus.
Connections	MIDI In/Out, 2 line outs (RCA), stereo headphone (mini).
System Req.	1 port, 1 IRQ.
Driver Support	Windows 3.x., Win 95/98, Window NT/2000, or any operating system that has an MPU-401 driver.
Compatibility	MPU-401 (UART and Intelligent). MT-32. Windows Drivers.
Street Price (U.S.)	$275

The SCC-1 is a MIDI Synth only (ie, no digital audio, ie, WAVE recording/playback). It sports the much-acclaimed Roland Sound Canvas waveform set as found in cards like the RAP-10, SCD-10, SCD-15, and SC-7 (but not the newer Sound Canvasses such as the SC-88, which have newer waveform sets and improved playback circuitry). The SCC-1 does not support replacing any of its GS waveforms.

See the RAP-10 description for remarks about the SCC-1's MIDI Synth engine. It's the same as the RAP-10's (with extra patches and drum kits, plus recognition of Bank Select Controller to select such), so all RAP-10 remarks (except as concern digital audio tracks) are pertinent (ie, its effects and support for MIDI messages is the same). The SCC-1's Parts do have a few extra parameters that can be adjusted, which the RAP-10 and SC-7 do not have, such as Attack Time and Filter Resonance.

Early SCC-1 units had less than 354 patches (ie, didn't have the extra MT-32 sounds). The only difference between new SCC-1 units and the SCC-1B appears to be that the latter includes bundled software, such as a nice patch editor (ie, although you can't change the GM waveforms used in a patch, you can adjust VCA and VCF envelopes to contour the sound. This is different than the RAP-10 or SC-7 more hard-wired GM Patch sets). The SCC-1 also offers MT-32 sounds (but not without some of the patch customization that the MT-32 offered). MT-32 sounds are mediocre by today's audio standards, but some older games were designed to support MT-32 rather than GM.

Furthermore, the SCC-1 has a fully intelligent MPU-401 compatibility, so it understands all MPU-401 commands. (That's not as important today since most software, especially Windows stuff, no longer uses MPU-401 Intelligent Mode).

The SCC-1 ships with a Windows MCI driver, (and the SCC-1B with some apps), which will also work under Windows 95 (but you're better off using Win95's MPU-401 driver instead). It can be used with pro sequencers such as CakeWalk. You can also use OS/2's RTMIDI MPU-401 driver.

Roland has replaced the SCC-1 with a package containing the MPU-401/AT and SCD-15 daughterboard. It's essentially the same thing as an SCC-1B. (I forget what they call this combo deal).

SC-55

Output Channels	2
Output Rate/ Resolution	16-bit @ 44.1KHz
Digital Input	None
MIDI Synth	WaveTable synth, GS Patches (317) in ROM. 9 Drum kits. 24 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	No information
Custom Sample Upload	None
Effects	Digital reverb/delay. Chorus.
Connections	COM port, MIDI In/Out, stereo line outs (RCA jacks), headphone (mini), stereo line inputs (RCA), AC power (uses a wall-wart).
Controls	Volume knob, LCD, parameter buttons (ie, Volume, Pan, Reverb, Chorus, Transpose, Instrument, MIDI CH, Part #, Mute Part).
System Req.	Attaches to computer's serial port.
Driver Support	Windows 3.x., Windows 95/98 (if you use the Roland Win95/98 serial driver available on Roland's web site).
Compatibility	Windows Drivers. MT-32.
Street Price (U.S.)	???

The SC-55 is essentially the SCC-1B (with a few less patches) in an external box that attaches to the computer's serial port. (It can attach to both the IBM and Mac serial ports).

See the RAP-10 description for remarks about the SC-55's MIDI Synth engine. It's the same as the SCC-1B's, so all SCC-1B remarks are pertinent.

The SC-55 is a complete, standalone unit. See the SC-7 description for what advantages Roland standalone units offer.

SC-155

Same as SC-55 except for:

Controls	Volume knob, 8 sliders, LCD, parameter buttons.

The SC-155 is the SC-55 in a table top unit (ie, the SC-55 is a half-rack unit) with the addition of 8 sliders to control various parameters of 8 parts (plus a button to switch the sliders to control the other 8 parts). With sliders, the SC-155's parameters can be more easily manipulated while playing the unit. Sliders also can send MIDI messages to record their movements with a sequencer.

CM-300

Same as SC-55 except for:

Controls	Volume knob.

The CM-300 is the SC-55 with all controls except the Volume knob removed. As such, you need computer software to change parameters (via MIDI).

Roland also made the CM-500 which is the CM-300 with an MT-32 added inside of it.

SC-35

Same as SC-55 except for:

MIDI Synth	WaveTable synth, GS Patches (223) in ROM. 8 Drum kits. 28 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Controls	Volume and Aux In Vol knobs, 3.5" floppy drive, parameter and sequencer (Play, Stop, Rec, Rew, FF, Pause, Tempo, etc) buttons.

The SC-35 is essentially the SC-55 plus a Sound Brush (ie, a sequencer that records and plays standard MIDI files off of a built-in 3.5 inch floppy drive) in one box. As such, the SC-35 is a

standalone General MIDI file player/recorder. It also has 4 extra voices of polyphony, but omits the MT-32 patches.

See the SC-55 description for remarks about the SC-35's features.

SC-33

Same as SC-55 except for:

MIDI Synth	WaveTable synth, GS Patches (226) in ROM. 8 Drum kits. 28 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Controls	Volume knob, LCD, editing buttons.

The SC-33 is the SC-55 but with editing capabilities over the "tones" (ie, waveform parameters such as VCA envelopes, VCF envelopes, etc) to design your own patches. It also has 4 extra voices of polyphony, but omits the MT-32 patches.

SC-50

Output Channels	2
Output Rate/ Resolution	18-bit @ 44.1KHz
Digital Input	None
MIDI Synth	WaveTable synth, GS Patches (226) in ROM. 8 Drum kits. 28 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	No information
Custom Sample Upload	None
Effects	Digital reverb/delay. Chorus.
Connections	COM port, MIDI In/Out, stereo line outs (RCA jacks), headphone (mini), stereo line inputs (RCA), AC power (uses a wall-wart).
Controls	Volume knob, LCD, editing buttons.
System Req.	Attaches to computer's serial port.
Driver Support	Windows 3.x., Windows 95/98 (if you use the Roland Win95/98 serial driver available on Roland's web site).

Compatibility	Windows Drivers.
Street Price (U.S.)	???

The SC-50 is essentially the SC-55 without the MT-32 sounds, but appears to have improved playback circuitry (including 18-bit DACs). The MT-32 sounds are mediocre by today's audio standards, so are really only useful for older games that support MT-32 but not GM.

Also, the SC-50 has 4 extra voices of polyphony over the SC-55.

See the SC-55 description for remarks about the SC-50's features.

The SC-50 does have an LCD panel and knobs, so that SC-50 Patch parameters can be edited from the SC-50 itself (as opposed to the SC-7 and SC-55 requiring a software "patch editor" to change various parameters).

SC-88

Output Channels	2
Output Rate/ Resolution	18-bit @ 44.1KHz
Digital Input	None
MIDI Synth	WaveTable synth, GS Patches (654) in ROM. 24 Drum kits. 64 voice polyphony. 32-Part multi-timbral via 2 MIDI inputs.
Sound ROM Size	8 MEG compressed. (16 MEG)
Custom Sample Upload	None
Effects	Digital reverb. Delay. Chorus. EQ.
Connections	COM port, 2 MIDI In/Out (THRU), stereo line outs (RCA jacks), headphone (mini), stereo line inputs (RCA), AC power cord.
Controls	Volume knob, LCD, editing buttons.
System Req.	Attaches to computer's serial port.
Driver Support	Windows 3.x., Windows 95/98 (if you use the Roland Win95/98 serial driver available on Roland's web site).
Compatibility	Windows Drivers. MT-32

Street Price (U.S.)	???

The SC-88 is an external box that attaches to the computer's serial port. (It can attach to both the IBM and Mac serial ports). It contains not only the SCC-1's waveform set, but also newer waveforms (over 300 more) and improved playback circuitry (including 18-bit DACs). It also has many more effects (and reverb and delay are separate units now, so Reverb, Delay, Chorus, and EQ are simultaneous). The SC-88 therefore offers a substantially different and improved MIDI Synth than all other Roland Sound Canvas cards (except the SC-55mkII). The SC-88 does not support replacing any of its waveforms, nor does it record/play digital audio.

The SC-88 is a complete, standalone unit. It can be used without the computer. It has its own power supply and case. An external keyboard (or other MIDI controller) can be plugged directly into the SC-88 MIDI In to play it as a 16-Part multi-timbral module. A second controller can be plugged into the second MIDI In to play 16 more multi-timbral parts. (In order to use all 32 mult-timbral parts with the computer, you have to have the latest version of Roland's MIDI serial driver, available from Roland's web site, under Products/Software updates. This driver turns the serial port into 2 separate MIDI Outs, each with 16 MIDI channels).

Furthermore, because the SC-88 offers MIDI In and Out jacks to which you can attach even more MIDI gear, it becomes a MIDI interface between your computer and all other external gear. This makes the SC-88 nice for laptop use since it allows connecting external MIDI gear via a clone or Mac serial port, is a multi-timbral unit also, and can even be used without the computer.

The SC-88 does have an LCD panel and knobs, so that its Patch parameters can be edited from the SC-88 itself (as opposed to the SC-7 and SC-55 requiring a software "patch editor" to change various parameters). The SC-88VL is the same as the SC-88 expect that it doesn't have the patch editing knobs (ie, like the SC-7 and SC-55, it requires computer software to change its patch settings).

SC-55mkII

Output Channels	2
Output Rate/ Resolution	18-bit @ 44.1KHz
Digital Input	None
MIDI Synth	WaveTable synth, GS Patches (354) in ROM. 8 Drum kits. 28 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	No Information

Custom Sample Upload	None
Effects	Digital reverb. Delay. Chorus. Flange.
Connections	COM port, MIDI In/Out, stereo line outs (RCA jacks), headphone (mini), stereo line inputs (RCA), AC power (uses a wall-wart).
Controls	Volume knob, LCD, editing buttons.
System Req.	Attaches to computer's serial port.
Driver Support	Windows 3.x., Windows 95/98 (if you use the Roland Win95/98 serial driver available on Roland's web site).
Compatibility	Windows Drivers.
Street Price (U.S.)	???

The SC-55mkII is sort of like half of an SC-88 (in terms polyphony and parts, but also with less waveforms too), and has improved waveforms and circuitry over older sound canvasses.

SCD-10

Output Channels	2
Output Rate/ Resolution	16-bit @ 44.1KHz
Digital Input	None
MIDI Synth	WaveTable synth, GM Patches (128) in ROM. 6 Drum kits. 28 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	No information
Custom Sample Upload	None
Effects	Digital reverb/delay. Chorus.
Connections	None. (Uses host card's output stage).
System Req.	Attaches to host card. (ie, It's a Sound Blaster compatible daughterboard).
Driver Support	Uses host card's driver.
Compatibility	Depends upon host card. Designed for an SB or compatible.

Street Price (U.S.)	$149

The SCD-10 is essentially the SCC-1's MIDI Synth (first 128 patches only) in the form of a daughterboard that attaches to a Sound Blaster's Waveblaster connector. (It is not a self-contained card). The SCD-10 does not support replacing any of its GM waveforms, nor does it record/play digital audio (which is left up to the host card).

See the RAP-10 description for remarks about the SCD-10's MIDI Synth engine. It's the same as the RAP-10's, so all RAP-10 remarks (except as concern digital audio tracks) are pertinent (ie, its effects and support for MIDI messages is the same).

Unlike self-contained Roland cards such as the SCC-1, RAP-10, or SC-7, the SCD-10 uses its host's output stage, which in typical Sound Blaster cards and compatibles, is not quite as noise and distortion free as Roland cards.

The SCD-10 ships with some strictly hobbyist software (DoReMi and Easy Juke). It can be used with pro sequencers such as CakeWalk, provided your host card has proper driver support.

One real advantage of the SCD-10 (and SCD-15) over other Roland cards is that, since these cards attach to an SB compatible host, they offer the best audio solution to a game player. The RAP-10's digital audio hardware is not SB compatible, and therefore not supported by game software (which is typically MS-DOS software that writes directly to hardware rather than Windows software that uses MCI drivers). The other Roland cards do not offer digital audio playback. So they can't be used to play voice and sound effects in games (which are usually done with WAVE files. On the other hand, since most game software can play its background music via an MPU-401 port, all Roland cards can be used to play background music for those games). The SCD-10, in combo with its host, offers a complete solution, as the host plays voice and sound effects, and the SCD-10 with its renowned Roland waveform set plays the background music, without requiring two separate sound cards.

SCD-15

Output Channels	2
Output Rate/ Resolution	16-bit @ 44.1KHz
Digital Input	None
MIDI Synth	WaveTable synth, GS Patches (354) in ROM. Numerous Drum kits. 28 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	No information

Custom Sample Upload	None
Effects	Digital reverb/delay. Chorus.
Connections	None. (Uses host card's output stage).
System Req.	Attaches to host card. (ie, It's a Sound Blaster compatible daughterboard).
Driver Support	Uses host card's driver.
Compatibility	Depends upon host card. Designed for an SB or compatible.
Street Price (U.S.)	$185

The SCD-15 is the same thing as the SCD-10 except that it has the full set of GS patches (ie, 354 instead of only 128) plus some MT-32 sounds. It has a few extra drum kits too. Otherwise, all other remarks concerning the SCD-10 apply to the SCD-15.

The extra patches are really not of much use to game players since most game software sticks to the smaller GM set of patches. A musician seeking a greater variety of sounds for MIDI file recording/ playback may find the extra GS patches more useful, although the extra MT-32 sounds are mediocre by today's audio standards.

MPU-401/AT

Output Channels	None
Output Rate/ Resolution	None
Digital Input	None
MIDI Synth	None
Sound ROM Size	None
Custom Sample Upload	None
Effects	None
Connections	MIDI In and Out. SB compatible WaveBlaster connector. Stereo line outs (RCA).
System Req.	1 Port, 1 IRQ.
Driver Support	Win 3.x., Windows 95/98, Windows NT/2000, OS/2, or any other OS with an MPU-401 device driver.

Compatibility	MPU-401 (UART and Intelligent). Windows drivers.
Street Price (U.S.)	$68 (With the SCD-15 daughterboard, $228)

The MPU-401/AT is a MIDI interface card without any built-in sound module nor digital audio tracks. It is meant to control external MIDI units as well as any SB compatible daughterboard plugged into the MPU-401/AT's internal connector (such as the SCD-10 or SCD-15. In fact, putting an SCD-15 into an MPU-401/AT gives you essentially an SCC-1B). But, the MPU-401 itself is not SB compatible (ie, it accomodates SB daughterboards, but is not an SB compatible card itself, and therefore will not work with software requiring SB compatibility).

It offers MPU-401 Intelligent and Uart modes. An advantage of using this card as the host for an SCD-10 or SCD-15 is that this card's audio output circuitry is cleaner and quieter than what is normally found in an SB type card.

Earlier MPU-401 units did not have the daughterboard connector. The very first MPU-401 was actually an external box containing all of the Roland circuitry and MIDI jacks. This box attached into a secondary card that plugged into the ISA bus. Later, Roland reduced its chipset size, and was able to fit everything onto one ISA card that fit into the computer. This was the MPU-IPC. Recently, Roland added the daughterboard connector, for the MPU-401/AT. (I recommend the MPU-401/AT, as there were also some improvements in MIDI throughput made).

SCP-55B

Output Channels	2 at 44.1, 22.05, or 11.025kHz, 16 or 8 bit. (Any combination of the above, for example play 1 channel at 22.05kHz 8-bit, and the other at 44.1kHz 16-bit).
Output Rate/ Resolution	44.1 kHz (max.), 16-bit
Digital Input	Stereo jack, 44.1, 22.05, or 11.025kHz, 16 or 8 bit. (Any combination of the above, for example, record 1 channel at 22.05kHz 8-bit, and the other at 44.1kHz 16-bit simultaneously).
MIDI Synth	WaveTable synth, GS Patches (354) in ROM. Numerous Drum kits. 28 voice polyphony. 16-Part multi-timbral, with the option to set Parts to any channels or disable Parts.
Sound ROM Size	No information
Custom Sample Upload	None
Effects	Digital reverb/delay. Chorus.

Connections	With MCB-3 connector, it has MIDI IN and OUT, stereo mic in, stereo line in, and stereo line out (mini jack).
System Req.	PCMCIA port.
Driver Support	Windows 3.x. Windows 95/98.
Compatibility	Windows Drivers.
Street Price (U.S.)	$315

The SCP-55B is essentially the SCC-1's MIDI Synth in the form of a PCMCIA card that plugs into a PCMCIA slot on a computer. The SCP-55B does not support replacing any of its GM waveforms. Unlike the SCC-1, the SCP-55B incorporates the RAP-10's digital audio capabilities.

See the RAP-10 description for remarks about the SCP-55B's MIDI Synth engine. It's the same as the SCC-1B's, so all SCC-1B remarks are pertinent. Furthermore, see the RAP-10 description for remarks about the SCP-55B's digital audio tracks.

The SCP-55B is ideal for laptop computers with a PCMCIA slot. The SCP-55B can simultaneously play full MIDI scores along with 2 tracks of CD-quality audio (recorded/played to the Hard Drive). Therefore, multi-media presentations can be made with full background music scores and sound effects/voice narration happening simultaneously, at very good sound quality.

The SCP-55B needs the MCB-3 to provide external MIDI jacks and output/input jacks.

The SCP-55B ships with several Windows titles, including CakeWalk Express (create/play MIDI files), and Macromedia Action (a multi-media authoring tool).

NOTES:

All cards use Roland's own proprietary chipsets.

One nice thing about the Roland cards which they have over cheap cards is a hardware buffered MIDI output. This allows the computer's CPU to typically stream MIDI messages to the card without requiring the CPU to wait for the card to accept MIDI bytes. With the RAP-10, this can even happen at the full bus speed of 8MHz, resulting in a more stable timing for sequenced playback, especially under CPU-taxing OS's such as Windows. Other cards typically make the CPU wait until the card has shifted out a MIDI byte at 32KHz before the card will accept another byte.

Street prices were obtained from ComputAbility, a typical computer mail order firm (800-554-9948) during the time when these products were available new.

The sales/distribution for Roland's line of computer cards has been taken over by Editol.

There have probably been times when you play back a MIDI arrangement that you're working on, and you find yourself saying "This lacks punch. It sounds flat and lifeless. It lacks depth". You decide that you need to add more musical parts to the arrangement (or more percussion, or sound effects, or whatever). But as you overdub more, new musical parts, all you seem to end up with is a musical "soup" where there are so many different parts happening that now you can barely hear what each instrument is playing (ie, a too "busy" arrangement), and yet you still haven't gotten a playback that sounds any less flat and lifeless. The problem is likely that you've taken the wrong approach to solve your problem. There may be nothing wrong with your musical arrangement. Rather, it could be that your mix (ie, the blending and processing of all your MIDI modules' sounds) needs some work.

Consider this example. Take an acoustic guitar and play it in a closet. What you're probably going to hear is a rather "claustrophobic" and "confined" sound. Even if you were listening to a recording of such, and weren't told that the guitarist was playing in a closet, you're going to know that the guitarist is playing in a small room because your ear can detect the characteristics of that room's sound (as recorded by the microphone). What are the characteristics? They're the echoes of the guitar's notes quickly bouncing off of the nearby walls and ceiling and being recorded with the microphone. Furthermore, a room can boost and muffle certain frequencies (ie, sort of like an acoustic "Tone" control). Your ear instinctively knows that such short, sparse echoes indicate a small room. And this does affect the way that you perceive the musical performance. It doesn't "sound" spacious. It doesn't sound "big". It doesn't sound "powerful". It sounds puny and cramped. Now, have that guitarist play the exact same musical performance in a large concert hall. Your ear is going to hear an entirely different sound, and you're going to get an entirely different impression of the performance. The complex "ambience" of the large hall will make the performance sound more powerful, bigger, and even more dynamic.

Musicians are often taught all about music theory and arranging techniques, but seldom taught about acoustics (ie, the principles of how sound is made and what affects it). And that often leads them to seek the solution for a "bland, flat, lifeless" mix by adding more musical parts rather than by what they may really need; more complexity in blending and processing the sounds of their MIDI modules.

You may playback a MIDI song file that someone else made, and hear a wonderfully "thick" and complex string sound out of your sound card. "Why don't I hear that string sound out of my card when I play my own MIDI song files? What do I have to do?", you may ask.

What may likely be happening is that the MIDI file is sending the same MIDI data to 2 or more similiar sounding String Patches (Presets, Instruments, whatever), and they are all playing the exact same notes simultaneously. These patches may be panned in stereo, and have different effects applied to them. But because these patches are all playing the exact same musical notes in perfect (to the human ear) sync and have tonal characteristics that blend together well, your ear doesn't hear separate Patches so much as it hears one, really complex Patch. This is an example of using the capabilities of a MIDI module to add depth to the musical performance (rather than adding more musical parts to the arrangement).

Here's an experiment. Follow these steps:

1. Setup some **Analog** (ie, real) Strings Patch (on your sound module) to play upon MIDI Channel 1. (If you've got a GM Patch set, use Patch 49).
2. Pan the Patch to the center.
3. Set your controller keyboard to play only on channel 1.
4. Play some notes and chords on the keyboard, and listen to the sound.

Now try this:

1. Setup some **Synth** Strings Patch to play upon MIDI Channel 2. (For a GM Patch set, try Patch 51). Leave the Analog Strings Patch set to MIDI Channel 1.
2. Pan the Analog Strings hard left (ie, so it only sounds upon the left speaker).
3. Pan the Synth Strings hard right.
4. Set your controller keyboard to play both channels 1 and 2 simultaneously.
5. Play some notes and chords on the keyboard, and listen to the sound.

You'll note a fuller string sound with the second example (assuming that you don't have a crummy game audio card that uses the same waveform for both patches). That's because each note is playing 2 Patches simultaneously, but because they are similiar sounding Instruments and therefore blend well together, and because both Patches are playing in perfect sync with each other, your ear tends to perceive them as only one Patch. But because there are <u>some</u> differences in the sounds of the Patches, and because they are panned in stereo, your ear does detect that there is something more complex (and therefore "fuller", "thicker", etc) about this sound than with the first example. In essense, we've got a stereo string sound here, whereas before we had a mono string sound. We're getting into **Psycho-Acoustics** here, which include ways to fool the ear into hearing things a certain way.

Note: Taking 2 of the "multitimbral parts" of a GM module, perhaps altering them in slightly different ways (ie, panning them in opposite directions, assigning different Patches to them, Fine Tuning them slightly apart, applying different levels of Reverb or Chorus, etc), and then <u>making them play the exact same MIDI data</u> is the basis for getting fuller, "stereo" sounds out of your MIDI modules. How do you get them to play the same MIDI data? Well, you can either assign both parts to the same MIDI channel if your multitimbral module supports that (which will make it impossible to setup different Patches using MIDI Program change. You'll have to set the Patches by hand). Or, you can set the Parts to 2 different MIDI channels, copy the track in your sequencer to another track (ie, duplicate it), and set the 2 sequencer tracks to those 2 different MIDI channels.

If you're still unfamiliar with the structure of multitimbral modules (ie, what a "Part" is), you may want to read the article ["What's multitimbral?"](#).

Of course, there is a price to pay for such stereo shenanigans. When you had 16 multitimbral Parts playing in mono (ie, each playing its own MIDI part), you could have 16 musical parts. Now, if you pair up Parts to play the same musical part, the multitimbral module can play less musical parts. And there's the module's voice limit (ie, polyphony) to consider. With a stereo sound, each MIDI note is going to use up twice as many voices as with a mono part.

Here are my favorite techniques for getting more complex or "stereo" sounds out of a sound module.

All of them involve 2 Parts, usually panned in stereo (ie, one Part is hard left and the other is hard right). You can pan them not so far apart if you want to make the sound "tighter" or more focused toward center without losing much complexity. Also, who's to say that you can't add a third or fourth Part with slightly different settings, and make it play the same MIDI data, in order to thicken the sound even more? In any event, I think that, beyond 2 Parts with a wavetable-based card, you get diminishing returns.

Note: In sophisticated MIDI modules, each Patch can often play several different waveforms panned in stereo with different filter and volume contours, etc, such that you can create a single, stereo Patch. Contrast this to a typical, low-cost GM module which typically uses only one, mono waveform per Patch. In the former case, you don't need to apply the techniques that I'm about to describe to 2 different (mono) Patches playing the same MIDI data. Rather, you can apply these techniques to create a stereo Patch itself, and then let it play the one sequencer track.

The following techniques should work on any modern MIDI module. You can mix and match these Techniques. For example, you can use both Technique 1 and Technique 2 on the same pair of Parts for an even more dramatic stereo effect.

In all of these examples, we'll assume using Parts 1 and 2 of your multitimbral module. One is panned hard left, and the other is hard right.

Technique 1

1. Increase Part 1's tuning sharp by a few cents (ie, a small fraction of a half step. You don't want to be tuning in entire half-steps. Hopefully, your module has a "fine tuning" parameter).
2. Decrease Part 2's tuning to an opposite, lower value (ie, flat).

This is my favorite when using the same Patch for both Parts. Sometimes, if I can't find another Patch that blends well with the desired Patch, or I want the sound to be distinctly perceived to be one Patch, I'll assign the same Patch to both Parts, pan in stereo, and alter the tuning. This is particularly good at improving a piano by using **Grand Piano** for both Parts and making a stereo piano. Of course, that doesn't mean that this Technique is any less effective combined with Technique 2. Indeed, the combo is the most effective way to get the fullest stereo sounds.

You can increase the difference between the 2 Parts' tunings for more depth to the sound, but the farther that you widen the difference, the more of a "detuned" sound you get (which may be what you want). Always make the tuning settings complimentary so that the ear hears a "centered" tuning.

Technique 2

1. Pick out the desired Patch for Part 1.
2. Now find a different Patch for Part 2, but one that blends well with (ie, sounds similiar to) Part 1.

When creating "stereo sounds" with a GM module, I often pair up patches 49/51 (String section), 17/18 (Organ), 5/12 (percussive electric piano), 26/27 (Clean guitar), 30/31 (Distorted guitar), and 34/35 (electric bass). There are lots of other good combos, particularly with the synth patches.

Technique 3

1. Transpose one of the Parts 12 or 24 half-steps above the other Part (or -12 or -24 steps below).

If you want to create some sort of "overtone" sound, generally you'll make the higher transposed Part much softer volume than the lower Part; just enough to flavor the sound. It's also more subtle to not pan the Parts in stereo.

Technique 4

1. Set both Part's Channel Pressure to control Brightness (ie, Filter cutoff), but use opposite polarity (if your module supports that). For example, set increasing Channel Pressure to make Part 1 brighter (ie, increase filter cutoff), then set increasing Channel Pressure to make Part 2 less bright (ie, decrease filter cutoff).

As varying MIDI Channel Pressure is sent to the module, each Part will respond differently and contrary, creating a sensation of "movement" to the sound.

Alternately, you could send only one Channel Pressure message to the Parts <u>before</u> playing any notes, and then do not allow any more Channel Pressure to be sent to the Parts. For example, if you send Channel Pressure with a value of 100 or so to both Parts, then Part 1 will be brighter than Part 2 throughout the performance of a song. The difference in the timbre of the 2 Parts yields a stereo effect that doesn't shift about. This is particularly useful for when both Parts use the same Patch.

In fact, Channel Pressure is often useful for controlling other things such as volume. For example, choose to make increased Channel Pressure raise the volume of one Patch while lowering the volume of the other Patch. (If you choose 2 different Patches, you may wish to pan the 2 Parts to the same position if you don't want to hear the sound "moving" across the stereo spectrum, but rather simply want the character of the sound to change "in place"). Channel Pressure can often control LFO speed and depth on some sound modules. The thing to remember when you want to use Channel Pressure to create a fuller, more spacious stereo blending of the two Parts is to set the Parts to do <u>different</u> (and often exactly opposite) things when you increase Pressure.

Note that besides Channel Pressure, some modules allow certain MIDI controllers (such as MOD Wheel) to control a variety of parameters (ie, volume, panning, brightness, etc). By using Channel Pressure along with other controllers to vary lots of parameters while playing, you can create complex (ie, spacious, "big" sounding) stereo effects, and thus liven up your arrangements without needing to add more musical parts to it.

Technique 5

1. Add lots of Reverb to one Part, but no Reverb to the other Part.

This can yield a rather "spacey" sound since it's kind of unnatural, but that may be what you're after.

If your module's reverb unit is mono (ie, the reverb sound always ends up in the center), this is not as effective as it could be. In that case, to get something useful out of this Technique, you should combine it with one of the other Techniques.

This technique can be applied with other effects. For example, add Chorus to one Part, but no Chorus to the other Part. Again, a stereo effect unit is preferable to get the full benefit of that stereo sound.

A particularly good stereo effect can be gotten by panning a Patch hard left or right, and then panning a delay of it to the opposite side. With a single, short delay (ie, 20 to 30 milliseconds), this thickens the sound without creating a distinct echo that the human ear will detect as such. If you don't have a delay device, you can simulate that with two sequencer tracks; one of them delayed. (See the article "Can I create effects with MIDI?").

Before proceeding with this article, you may want to read my article on The architecture of Roland Sound Modules. It has a general overview of the design of typical Roland Sound Modules.

In the typical Roland user's manual, there is a chapter entitled **MIDI Implementation**. This chapter lists the details of how the unit responds to MIDI messages sent to it, as well as what MIDI messages the unit is capable of generating (if any) and sending to some other MIDI device (such as a sequencer).

I'm now going to walk you through the MIDI Implementation chapter of the Roland JV-90, to show you how to interpret a typical Roland MIDI Implementation chapter. The JV-90 has a built-in multi-timbral sound module which is capable of responding to many different MIDI messages. It also has a keyboard which is capable of generating many different kinds of MIDI messages (to its built-in sound module or other MIDI units). Many Roland sound modules share somewhat similiar architectures, and Roland tends to use the same syntax in its user manuals, so this exercise is applicable to many Roland sound modules. Although the Roland remarks and MIDI particulars I'll be showing you here are specific to the JV-90, you'll likely find many similiarities to the MIDI Implementation chapter in the manual for other Roland modules.

I'll be showing you exactly what is printed in the manual, and then adding my own comments immediately to the left of each line that I've taken from the manual. My comments will have a different color background than the line from the manual. Here's a typical example:

The exact line in the manual:	My comment about this line:
Control Change	This indicates the start of the sub-section where all of the MIDI controllers are listed.

The MIDI Implementation chapter is divided into 4 main sections.

The first main section is entitled RECEIVE DATA. This is the section that tells what MIDI messages your unit can understand being sent to it by another device, and what it can do with each of those received messages. (ie, This section lists what MIDI messages it can receive and do something with).

The second main section is entitled TRANSMIT DATA. This lists all of the MIDI messages that your unit is capable of generating when you operate it. (ie, It lists all of the MIDI messages that your unit can create and send to other MIDI units). Upon Roland sound modules that can't function as a controller of other devices, or which have no capability to dump their settings to another device, there may be no TRANSMIT DATA section. (ie, The sound module has no capability to generate MIDI messages on its own).

Each of these main sections is divided into subsections, where each subsection is dedicated to a type of MIDI message, starting with "Note Off" type of MIDI messages, and going all of the way to "System Exclusive" (and sometimes "System Realtime" messages such as MIDI Start, Stop, and Clock if the unit has some sort of onboard sequencer or arpeggiator). For example, the first subsection will most likely be entitled "Note Off", and it will list information about what your unit

does when it receives or transmits Note Off messages.

Incidentally, an "H" appended to some numeric value means that it is expressed as hexadecimal, not a decimal, number. For example, 7FH means 7F hexadecimal, which is 127 decimal. Use your Windows calculator in scientific mode to easily enter hex values and convert them to decimal, and vice versa).

The third and fourth main sections are dedicated to giving the excruciating details of the JV-90's System Exclusive data dumps. The JV-90 can dump all of its patch, performance, effects, etc, settings over MIDI using System Exclusive messages. Those settings can then be saved upon some storage device. The JV-90 can also accept such a dump to restore its settings to those previously saved values. In fact, every single parameter on the JV-90 which can be changed manually by you from the JV-90's front panel can also be changed over MIDI by sending a System Exclusive message to the JV-90. This System Exclusive message will contain info that tells what parameter to change, and what its new value should be.

1. RECEIVE DATA

This indicates the start of the RECEIVE DATA main section.

Channel Voice Message

This indicates that what follows is info about the types of messages that can be broadcast upon particular MIDI channels. Such types of messages are Note Off and On, Aftertouch, Channel Pressure, all of the Controller messages, Program Change, and Pitch Wheel.

Note off

Indicates that what follows is how my JV-90 handles Note Off messages it receives.

Status	Second	Third	
			Shows the format for a MIDI Note Off message. There are two versions of it. Both have 3 bytes. The Status is the first byte. The Second is the note number. The third is the velocity amount for the note.
8n**H**	kk**H**	vv**H**	Here's the first version of a Note Off message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The kk means that it can be any note number within a range of values. The allowable range of values will be shown later. The vv means that it can be any velocity within a range of values. The allowable range of values will be shown later.

9n**H**	kk**H**	00**H**	Here's the second version of a Note Off. It's actually a Note On message with the velocity byte always set to 0.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Note Off messages upon any of the 16 MIDI channels).
kk = Note number	:00**H** - 7F**H** (0 - 127)		Indicates that the allowable range for note number is 0 to 127. (ie, The JV-90 can recognize the full range of note numbers).
vv = velocity	:00**H** - 7F**H** (0 - 127)		Indicates that the allowable range for velocity is 0 to 127. (ie, The JV-90 can recognize the full range for note velocity).
In the Performance mode, ignored when the MIDI receive switch is OFF at each Part.			This is a comment about how the JV-90 handles MIDI Note Off messages it receives. All this is saying is that if you turn a Part off, it no longer "plays" MIDI Note Off messages. Duh!
In the Rhythm Part, ignored when "ENV Mode" is "NO-SUSTAIN" at each Rhythm Tone.			Another comment about how the JV-90 handles MIDI Note Off messages it receives. All this is saying is that unless you set a particular drum sound to sustain, then the drum kit doesn't really concern itself with Note Off messages.

Note on

Indicates that what follows is how my JV-90 handles Note On messages it receives.

Status	Second	Third	
			Shows the format for a MIDI Note On message. It has 3 bytes. The Status is the first byte. The Second is the note number. The third is the velocity amount for the note.
───────	───────	───────	
9n**H**	kk**H**	vv**H**	The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The kk means that it can be any note number within a range of values. The allowable range of values will be shown later. The vv means that it can be any velocity within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Note On messages upon any of the 16 MIDI channels).

kk = Note number	:00**H** - 7F**H** (0 - 127)		Indicates that the allowable range for note number is 0 to 127. (ie, The JV-90 can recognize the full range of note numbers).
vv = velocity	:00**H** - 7F**H** (0 - 127)		Indicates that the allowable range for velocity is 0 to 127. (ie, The JV-90 can recognize the full range for note velocity).
In the Performance mode, ignored when the MIDI receive switch is OFF at each Part.			This is a comment about how the JV-90 handles MIDI Note On messages it receives. All this is saying is that if you turn a Part off, it no longer "plays" MIDI Note On messages. Duh!
Control Change			Indicates that what follows is how my JV-90 handles Controller messages it receives. This subsection is typically quite long because it is divided up into numerous sub-subsections, where each sub-subsection is a different MIDI Controller, for example Bank Select, Mod Wheel, Pan, Volume, etc.
Bank Select MSB/LSB			Indicates that what follows is how my JV-90 handles Bank Select Controller messages it receives (both the coarse message and the fine message).
Status	Second	Third	Shows the format for a Bank Select Controller message. There are two messages -- one for coarse adjustment and one for fine adjustment. Both have 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
Bn**H**	00**H**	mm**H**	Here's the Bank Select message for coarse adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 00. The mm means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
Bn**H**	20**H**	ll**H**	Here's the Bank Select message for fine adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 20. The ll means that the fine value can be any value within a range of values. The allowable range of values will be shown later.

n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)	Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
mm = MSB of bank number	:50**H** - 54**H** (80 - 84)	Indicates the allowable range for coarse value. There are 5 allowable values, 80, 81, 82, 83, and 84. The chart below shows you which one of those coarse values, when used in conjunction with one of the allowable fine values, selects which bank.
ll = LSB of bank number	:00**H** - 7F**H** (0 - 127)	Indicates that the allowable range for the fine value is 0 to 127.
The Bank Select is suspended until receiving a program change.		This is a comment about how the JV-90 handles Bank Select messages it receives. All this is saying is that until you actually follow up the Bank Select messages with a Program Change message, the patch doesn't change.
Ignored when "Program bank sel" of the system common is OFF.		Another comment about how the JV-90 handles Bank Select messages it receives. All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Program bank sel". If you set this to off, then the JV-90 ignores Bank Select messages it receives.
In the patch mode, selected a bank of the patch memory. In the performance mode, selected a bank of the performance part memory. And specified the control channel, selected a bank of the performance part memory.		If you're in patch mode, and the JV-90 receives a Bank Select, it will use this (in conjunction with a following Program Change message) to select a new patch that the keyboard plays. If you're in Performance mode, and the JV-90 receives a Bank Select, what happens depends upon whether you have the Control Channel enabled and then whether the Bank Select is transmitted on that channel. If so, then the Bank Select is used to change to another Performance among several Banks of Performances. Otherwise, the Bank Select is used to change the patch assigned to one of the Parts in the currently selected Performance. Which part? Well, the part that is assigned to the same MIDI channel as the Bank Select message was sent upon.

This is just telling you that below this is now a chart that shows you what specific coarse and fine bank numbers to use, in conjunction with what Program Change numbers, to select specific patches in certain components upon the JV-90. For example, what Bank Select fine and coarse numbers do you use, and what Program Change number do you use, to select the first patch in the "Preset B" bank? According to the chart below, that would be a coarse bank number of 81, a fine bank number of 1, and a Program Change number of 65. The article Changing patches over MIDI using Bank Select Controller has a more indepth explanation and example using the below JV-90 Patch chart. The JV-90's use of Bank Select controller is similiar to other Roland Sound Modules, and may help you in understanding how to utilize your own manual's Patch chart.

The bank number specified as follow.

80	0	1 - 64	Internal	(#1 - #64)
80	0	65 - 128	Data Card	(#1 - #64)
81	0	1 - 64	JV-80 Preset A	(#1 - #64)
81	0	65 - 128	JV-80 Preset B	(#1 - #64)
81	1	1 - 64	Preset A	(#1 - #64)
81	1	65 - 128	Preset B	(#1 - #64)
81	2	1 - 64	Preset C	(#1 - #64)
81	2	65 - 128	Preset D	(#1 - #64)
82	0	1 - 64	Data Card	(#1 - #64)
83	0	1 - 128	PCM Card	(#1 - #128)
84	0	1 - 128	Expansion Board	(#1 - #128)
84	1	1 - 128	Expansion Board	(#129 - #256)

Modulation

Indicates that what follows is how my JV-90 handles Modulation Controller messages it receives.

Shows the format for a Modulation Controller message. The JV-90 uses only the message for coarse adjustment. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.

Status	Second	Third

BnH	01**H**	vv**H**	Here's the Modulation message for coarse adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 01. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Modulation depth	:00**H** - 7F**H** (0 - 127)		Indicates the allowable range for coarse value. Supports the full coarse range of Mod depth.
The effect of the modulation depends on the value of "Mod1 - 4" of the patch tone.			This is a comment about how the JV-90 handles Modulation messages it receives. All this is saying is that the modulation can be set to affect upto 4 different Tone parameters simultaneously, for example the Tone's tuning in order to achieve a vibrato effect, and/or the Tone's filter cutoff to achieve a growl effect, etc.
Ignored when "Receive Modulation" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Modulation". If you set this to off, then the JV-90 ignores Modulation messages it receives.
Portamento time			Indicates that what follows is how my JV-90 handles Portamento time Controller messages it receives.
Status	Second	Third	Shows the format for a Portamento time Controller message. The JV-90 uses only the message for coarse adjustment. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
————	————	————	
BnH	05**H**	vv**H**	Here's the Portamento time message for coarse adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 05. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.

n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)	Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Portamento time	:00**H** - 7F**H** (0 - 127)	Indicates the allowable range for coarse value. Supports the full coarse range of Portamento time.
You can adjust the portamento time of the patch common.		This is a comment about how the JV-90 Portamento time setting. All this is saying is that there is a setting in the "Patch Common" screen to set the default Portamento time for a patch (although the Portamento time can be changed at any time by a Portamento time controller message).
Ignored when "Receive Control change" of the system common is OFF.		All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Control change". If you set this to off, then the JV-90 ignores Portamento time messages it receives.

Volume

Indicates that what follows is how my JV-90 handles Volume Controller messages it receives.

Status	Second	Third	
			Shows the format for a Volume Controller message. The JV-90 uses only the message for coarse adjustment. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
Bn**H**	07**H**	vv**H**	Here's the Volume message for coarse adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 07. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Volume	:00**H** - 7F**H** (0 - 127)		Indicates the allowable range for coarse value. Supports the full coarse range of Volume.

You can adjust the volume of specified channel.	This is a comment about Volume controller messages. All this is saying is that this volume controls the volume of all Note messages upon the same channel as the Volume controller is sent upon.
Ignored when "Receive Volume" of the system common is OFF.	All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Volume". If you set this to off, then the JV-90 ignores Volume messages it receives.
In the performance mode, ignored when volume receive switch is OFF at each part.	All this is saying is that you can choose to have an individual Part (of a Performance) ignore Volume controller messages (whereas the other Parts in the same Performance perhaps may not). There is such a setting in one of the screens for editing a Performance.
Ignored when "Volume switch" of the patch tone is OFF.	All this is saying is that you can choose to have a Tone (used in a patch) itself ignore Volume controller messages (whereas the other Tones in the same patch perhaps may not). There is such a setting in one of the screens for editing a Tone.

Pan

Indicates that what follows is how my JV-90 handles Pan Controller messages it receives.

Status	Second	Third	
			Shows the format for a Pan Controller message. The JV-90 uses only the message for coarse adjustment. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
BnH	0A**H**	vv**H**	Here's the Pan message for coarse adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 0A. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Pan	:00**H** - 7F**H** (0 - 127)		Indicates the allowable range for coarse value. Supports the full coarse range of Pan.

"0" represents left end, "64" represents the center, and "127" represents the right end.			This is a comment about Pan controller messages. All this is saying is that a 0 value pans the sound all of the way to the left speaker. 64 pans it center. (And therefore values inbetween 64 and 0 would pan it increasingly to the left). 127 pans it all of the way to the right speaker. (And therefore values inbetween 65 and 127 would pan it increasingly to the right).
Ignored when "Receive Control change" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Control change". If you set this to off, then the JV-90 ignores Pan messages it receives.
Expression			Indicates that what follows is how my JV-90 handles Expression Controller messages it receives.
Status	Second	Third	Shows the format for an Expression Controller message. The JV-90 uses only the message for coarse adjustment. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
Bn**H**	0B**H**	vv**H**	Here's the Expression message for coarse adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 0B. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Expression	:00**H** - 7F**H** (0 - 127)		Indicates the allowable range for coarse value. Supports the full coarse range of Expression.
The effect of the expression depends on the value of "Exp1 - 4" of the patch tone.			This is a comment about how the JV-90 handles Expression messages it receives. All this is saying is that the expression can be set to affect upto 4 different Tone parameters simultaneously, for example the Tone's tuning in order to achieve a vibrato effect, and/or the Tone's filter cutoff to achieve a growl effect, etc.

Ignored when "Receive Control change" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Control change". If you set this to off, then the JV-90 ignores Expression messages it receives.
Hold 1			Indicates that what follows is how my JV-90 handles Hold 1 Controller messages it receives.
Status	Second	Third	Shows the format for a Hold 1 Controller message. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
———	———	———	
Bn**H**	40**H**	vv**H**	Here's the Hold 1 message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 40. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Control value	:00**H** - 7F**H** (0 - 127) 0 - 63 = OFF 64 - 127 = ON		Indicates the allowable range. Any value from 0 to 63 means that the Hold pedal is off. Any value from 64 to 127 means Hold pedal is on.
Notes played can be sustained for as long as the time that elapses between turning hold on and turning hold off.			All this is saying is that notes get sustained for as long as the Hold 1 pedal is on. Duh!
Ignored when "Receive Control change" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Control change". If you set this to off, then the JV-90 ignores Hold 1 messages it receives.
In the rhythm part (part8), ignored when "ENV mode" is "NO - SUSTAIN" at each rhythm tone.			All this is saying is that for any drum sound that you set to "NO SUSTAIN", then the Hold 1 pedal has no effect upon the drum sound (ie, can't be used to sustain the drum sound).

Ignored when "Hold - 1 switch" of the patch tone is OFF.			All this is saying is that you can choose to have a Tone (used in a patch) itself ignore Hold 1 controller messages (whereas the other Tones in the same patch perhaps may not). There is such a setting in one of the screens for editing a Tone.
Portamento			Indicates that what follows is how my JV-90 handles Portamento (On/Off) Controller messages it receives.
Status	Second	Third	Shows the format for a Portamento Controller message. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
Bn**H**	41**H**	vv**H**	Here's the Portamento message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 41. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Control value	:00**H** - 7F**H** (0 - 127) 0 - 63 = OFF 64 - 127 = ON		Indicates the allowable range. Any value from 0 to 63 means that the Portamento is off. Any value from 64 to 127 means Portamento is on.
Switched over "Portamento sw" of patch common.			This is a comment about how the JV-90 handles received Portamento messages. All this is saying is that the Portamento On/Off controller message overrides the default setting in the patch common screen for whether portamento is turned on or off.
Ignored when "Receive Control change" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Control change". If you set this to off, then the JV-90 ignores Portamento messages it receives.
Effect1 depth (Reverb send level)			Indicates that what follows is how my JV-90 handles Effect1 Controller messages it receives. The JV-90 uses this controller to turn the reverb on or off.

Status	Second	Third	
			Shows the format for a Effect1 Controller message. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
Bn**H**	5B**H**	vv**H**	Here's the Effect1 message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 5B. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Control value	:00**H** - 7F**H** (0 - 127) 0 - 63 = OFF 64 - 127 = ON		Indicates the allowable range. Any value from 0 to 63 means that Reverb is off. Any value from 64 to 127 means Reverb is on.
In the patch mode, switched over reverb switch of the system common.			This is a comment about how the JV-90 handles received Effect1 messages. All this is saying is that, when you're in patch play mode, the Effect1 controller message overrides the default setting in the system common screen for whether reverb is turned on or off for the patch.
In the performance mode, switched over reverb switch of the performance.			Each part (of a performance) has a default setting for whether its reverb is turned on or off. The Effect1 controller message overrides this default setting.
Ignored when "Receive Control change" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Control change". If you set this to off, then the JV-90 ignores Effect1 messages it receives.
Effect3 depth (Chorus send level)			Indicates that what follows is how my JV-90 handles Effect3 Controller messages it receives. The JV-90 uses this controller to turn the Chorus on or off.
Status	Second	Third	Shows the format for a Effect3 Controller message. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.

Bn**H**	5D**H**	vv**H**	Here's the Effect1 message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 5D. The vv means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = Control value	:00**H**... 7F**H** (0 - 127) 0 - 63 = OFF 64 - 127 = ON		Indicates the allowable range. Any value from 0 to 63 means that Chorus is off. Any value from 64 to 127 means Chorus is on.
In the patch mode, switched over chorus switch of the system common.			This is a comment about how the JV-90 handles received Effect3 messages. All this is saying is that, when you're in patch play mode, the Effect3 controller message overrides the default setting in the system common screen for whether chorus is turned on or off for the patch.
In the performance mode, switched over chorus switch of the performance.			Each part (of a performance) has a default setting for whether its chorus is turned on or off. The Effect3 controller message overrides this default setting.
Ignored when "Receive Control change" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Control change". If you set this to off, then the JV-90 ignores Effect3 messages it receives.
RPN MSB/LSB			Indicates that what follows is how my JV-90 handles Registered Parameter Number Controller messages it receives (both the coarse message and the fine message). See the article [Registered and Non-registered Parameter Number controllers](#) for more information on using Data Entry in conjunction with RPN or NRPN.

Status	Second	Third	
			Shows the format for an RPN Controller message. There are two messages -- one for coarse adjustment and one for fine adjustment. Both have 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller (ie, the low or high bits of the parameter number).
Bn**H**	65**H**	mm**H**	Here's the RPN message for coarse adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 65. The mm means that the coarse value can be any value within a range of values. The allowable range of values will be shown later.
Bn**H**	64**H**	ll**H**	Here's the RPN message for fine adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 64. The ll means that the fine value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
mm = MSB of the specified parameter by RPN			The Registered Parameter numbers that the JV-90 recognizes will be listed later in the manual.
ll = LSB of the specified parameter by RPN			The Registered Parameter numbers that the JV-90 recognizes will be listed later in the manual.
Data entry MSB/LSB			Indicates that what follows is how my JV-90 handles Data entry (slider) Controller messages it receives (both the coarse message and the fine message). See the article Registered and Non-registered Parameter Number controllers for more information on using Data Entry in conjunction with RPN or NRPN.

Status	Second	Third	
			Shows the format for an Data entry Controller message. There are two messages -- one for coarse adjustment and one for fine adjustment. Both have 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller (ie, the low or high bits of a value for some RPN or NRPN).
BnH	06H	mmH	Here's the Data entry message for coarse adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 06. The mm means that the coarse value can be any value within a range of values.
BnH	26H	llH	Here's the Data entry message for fine adjust. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 26. The ll means that the fine value can be any value within a range of values.
n = MIDI Channel	:0H - FH (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
mm = MSB of the value of the parameter specified with RPN			The "high" bits of the value for some RPN.
ll = LSB of the value of the parameter specified with RPN			The "low" bits of the value for some RPN.
** RPN **			The chart that follows lists all of the Registered Parameter Numbers that the JV-90 understands. It understands 4 RPNs -- Pitch Range, Fine Tuning, Coarse Tuning, and RPN Reset. The values of any of these 4 RPN settings can be changed with Data entry controller messages.

RPN	Data entry	Description	The chart had 3 columns. The first shows you the coarse and fine controller numbers to send (via the 2 RPN controllers) in order to select this Parameter. The second shows you the allowable range of coarse and fine values to which you can set this Parameter (using the 2 Data Slider controllers). The last column tells you what the parameter is, gives additional info about its range of allowable values, and any remarks about the Parameter.
MSB LSB	MSB LSB		
00H 00H	mmH --	Pitch bend sensitivity	This is the Parameter to set the range of the pitch wheel.
		mm : 00H - 0CH (0 - 12 semitones)	You can set its value anywhere between 0 and 12, where 0 means no pitch wheel transpose at all, and 12 gives you the maximum range of up and down an octave. For example, a 2 gives you a range up and down a whole step.
		ll : Ignored	The Data Slider fine controller is not used to set pitch range. Only the Data Slider coarse controller (ie, 06) is used to set the range.
		(Up to one octave)	As noted, the maximum value is 12, which gives a range of up and down an octave.
		* You can adjust "Bend range up" and "Bend range down" at the same time	When you send a value for Pitch bend range, it sets both the amount that the wheel transposes up as well as down by the same amount. In other words, you can't set it to transpose up an octave, but down only a whole step.
		* In the Rhythm part, this message is not recognized.	You can't set the pitch wheel transpose range for the drum part. You can set it only for the other parts of a performance (and each part has its own range setting), or a patch.
00H 01H	mmH llH	Fine tuning	This is the Parameter to set the Master tuning in cents.

mm, ll : 20**H**, 00**H** - 40**H**, 00**H** - 60**H**, 00**H** (-50 - 0 - +50 cents)	You can set its value a maximum of 50 cents up or down. For -50 cents, you send a coarse value of 20 and fine value of 00. For centered tuning (ie, 0 offset), send coarse = 40 and fine = 00. For +50 cents, send coarse = 60 and fine = 00.
* In the patch mode, adjusted the master tune.	When in patch play mode, you set the master tune. (ie, When you switch patches, the same tuning setting will stick until you set a different value for Fine tuning).
* In the performance mode, adjusted fine tune for each part.	In performance play, each part has its own Fine tuning setting. So although all of the patches you use with that part will be fine tuned by this amount, those same patches used with other parts will not be.
* In the performance mode, specified control channel, changed the master tune.	In performance play, if you've got the control channel enabled, and you send RPN and Data Sliders on that channel, THEN it sets the master tune. In other words, all parts will be fine tuned by that amount. Otherwise, you'd have to fine tune each part separately.

00**H** 00**H**	mm**H** --	Coarse tuning	This is the Parameter to set the Master transpose in half steps. I prefer to call this parameter "Transpose Amount".
		mm : 10**H** - 40**H** - 70**H** (-48 - 0 - +48 semitones)	You can set its value up or down a maximum of 48 half steps. For -48 steps, you send a coarse value of 10. For centered tuning (ie, 0 transpose), send coarse = 40. For +48 half steps, send coarse = 70.
		ll : Ignored	The Data Slider fine controller is not used to set Transpose Amount. Only the Data Slider coarse controller (ie, 06) is used to set the Transpose Amount.
		* In the patch mode, this message is not recognized.	When in patch play mode, this message doesn't change the Transpose Amount. You can't set this parameter over MIDI when in patch play mode.
		* In the performance mode, adjusted coarse tune for each part.	In performance play, each part has its own Transpose Amount setting. So although all of the patches you use with that part will be transposed by this amount, those same patches used with other parts will not be.

7F**H** 7F**H**	-- --	RPN reset	This is sort of a non-existant Parameter. When you select this parameter, all it means is that any subsequent Data Slider controllers will be ignored. That way, you won't accidentally have them change any one of the above 3 Parameters that can be changed via Data Slider controllers.
		mm, ll : Ignored	Like I said, the whole point of selecting this Parameter is just to force the JV-90 to ignore all subsequent Data Slider messages.
		* Return to no specified parameter of RPN. Current setting value is no change.	This pigeon english is struggling to tell you what I just did above. Honest.

Program Change

Program Change			Indicates that what follows is how my JV-90 handles Program Change messages it receives. The JV-90 uses this message in conjunction with Bank Select controllers to change patches.
Status	Second		Shows the format for a Program Change message. It has 2 bytes. The Status is the first byte. The second is the program number.
Cn**H**	pp		Here's the Program Change message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The pp means that the value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Program Change messages upon any of the 16 MIDI channels).
pp = Program number	:00**H** - 7F**H** (prog.1 - prog.128)		Indicates the allowable range. The full range of program numbers is supported. Note that program #1 is sent in the Program Change message as a 0 value.
Ignored when "Receive Program change" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Program change". If you set this to off, then the JV-90 ignores Program Change messages it receives.

When the JV-90 receives a program change on a part receive channel while in the performance mode, it changes the patches of that part: the new patch value being the program number plus 1. If the JV-90 receives the program change on the control channel, it changes the performance.

Pretty self-explanatory. In patch mode, Program Change changes to another patch. In performance play mode, each Part has its own patch assignment. So you can use a Program Change message to change the patch for each part individually. If you got the control channel enabled, and you send the Program Change on that channel, then it changes to another Performance instead.

Channel Pressure

Indicates that what follows is how my JV-90 handles Channel Pressure messages it receives. The JV-90 can use this message for a variety of purposes from adjusting the volume of notes while they are being sustained, to adding vibrato, etc.

Status	Second

Shows the format for a Channel Pressure message. It has 2 bytes. The Status is the first byte. The second is the pressure amount.

Dn**H**	vv

Here's the Channel Pressure message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The vv means that the value can be any value within a range of values. The allowable range of values will be shown later.

n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)

Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Channel Pressure messages upon any of the 16 MIDI channels).

vv = Pressure value	:00**H**... 7F**H** (0 - 127)

Indicates the allowable range. The full range of Channel Pressure is supported.

The effect of Channel pressure depends on the value of "After1 - 4" of the patch tone.

This is a comment about how the JV-90 handles Channel Pressure messages it receives. All this is saying is that the pressure amount can be set to affect upto 4 different Tone parameters simultaneously, for example the Tone's tuning in order to achieve a vibrato effect, and/or the Tone's filter cutoff to achieve a growl effect, etc.

Ignored when "Receive Aftertouch" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Aftertouch". If you set this to off, then the JV-90 ignores Channel Pressure messages it receives.
Pitch bend change			Indicates that what follows is how my JV-90 handles Pitch Wheel messages it receives.
Status	Second	Third	Shows the format for a Pitch Wheel message. It has 3 bytes. The Status is the first byte. The Second is the coarse pitch change. The third is the fine pitch change.
En**H**	mm**H**	ll**H**	Here's the Pitch Wheel message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The mm means that the coarse value can be any value within a range of values. The allowable range of values will be shown later. The ll means that the fine value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Pitch Wheel messages upon any of the 16 MIDI channels).
mm, ll = Pitch bend change	:00**H**, 00**H** - 7F**H**, 7F**H** (-8192 - 0 - +8191)		Indicates the allowable range. Supports the full range for Pitch Wheel.
Ignored when "Receive Pitch bend" of the system common is OFF.			All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Pitch bend". If you set this to off, then the JV-90 ignores Pitch Wheel messages it receives.
Channel Mode Message			This indicates that what follows is info about the types of messages that can change the JV-90's MIDI modes. This subsection is sort of out of place since these are all controller messages that can be broadcast upon particular MIDI channels. They really should have been listed under the Controller messages.

Reset All Controllers

Indicates that what follows is how my JV-90 handles a Reset All Controllers message it receives. This message is used to reset several other Controllers to their default values.

Status	Second	Third	
			Shows the format for a Reset All Controllers message. It has 3 bytes.
Bn**H**	79**H**	00**H**	Here's a Reset All Controllers message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 79 and the third byte is always 00.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).

Received this message. The controllers is set the following.

Ouch! More pigeon english! All this is saying is that the following chart shows you which Controllers are reset by a Reset All Controllers message, and what their default values are.

Controller	Value	
		The chart lists the Controller's name on the right, and its respective default value on the left.
Modulation	0 (off)	The Mod wheel is reset to its minimum position.
Volume	127 (maximum)	Volume is reset to maximum.
Pan	64 (center)	Pan is reset to center.
Expression	0 (off)	Expression is reset to minimum (ie, no effect applied).
Hold1	0 (off)	The Hold pedal is released. (ie, Notes stop sustaining).
Channel pressure	0 (off)	Channel pressure is reset to minimum (ie, no effect applied).
Pitch bend change	+/- 0 (center)	The Pitch Wheel is reset to center position.
RPN	No specified parameter	None of the Registered Parameter Numbers are reset by a Reset All Controllers message. For example, the Pitch Wheel Bend Range isn't reset to its default state (ie, when the unit was turned on).

Local control

Indicates that what follows is how my JV-90 handles a Local control message it receives. This message is used to turn the keyboard on or off (ie, have it directly play its internal sound module or not).

Status	Second	Third	
			Shows the format for a Local control message. It has 3 bytes.
BnH	7AH	vvH	Here's a Local control message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 7A. (The JV-90 manual has a typo and shows it incorrectly as 7B). The third byte is a 0 for "off" or 127 for "on".
n = MIDI Channel	:0H - FH (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = value	:00H, 7FH (0, 127) 0=OFF 127=ON		See above.

All notes off

Indicates that what follows is how my JV-90 handles an All notes off message it receives. This message is used to turn off all sustaining notes (although if the Hold1 controller is still on, they won't stop sounding).

Status	Second	Third	
			Shows the format for an All notes off message. It has 3 bytes.
BnH	7BH	00H	Here's an All notes off message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 7B, and the third byte is always 00.
n = MIDI Channel	:0H - FH (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).

When this message is recognized, all the notes which have been turned on by MIDI note on message are turned off.

Yeah, I already told you that above.

OMNI OFF

Indicates that what follows is how my JV-90 handles an OMNI OFF message it receives. This message is normally used to allow a unit to not respond to all 16 MIDI channels. But, the JV-90 merely uses it as if it were nothing more than an All notes off message.

Status	Second	Third

Shows the format for an OMNI OFF message. It has 3 bytes.

Status	Second	Third
BnH	7CH	00H

Here's an OMNI OFF message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 7C, and the third byte is always 00.

n = MIDI Channel   :0H - FH (ch1 - ch16)

Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).

Recognized as All notes off.

I already told you that above too.

OMNI ON

Indicates that what follows is how my JV-90 handles an OMNI ON message it receives. This message is normally used to allow a unit to respond to all 16 MIDI channels. But, the JV-90 merely uses it as if it were nothing more than an All notes off message.

Status	Second	Third

Shows the format for an OMNI ON message. It has 3 bytes.

Status	Second	Third
BnH	7DH	00H

Here's an OMNI ON message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 7D, and the third byte is always 00.

n = MIDI Channel   :0H - FH (ch1 - ch16)

Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).

Recognized as All notes off.

Guess what? I already told you that above.

MONO

Indicates that what follows is how my JV-90 handles a MONO Controller message it receives. This message is used to set the sound module in monophonic state (ie, plays only 1 note at a time). In the JV-90's case, this is referred to as "Solo mode", and each part (of a Performance) can be set to solo mode individually. When in solo mode, a part's portamento and legato settings become effective. Otherwise, they are ignored.

Status	Second	Third	
			Shows the format for a MONO controller message. It has 3 bytes.
BnH	7EH	vvH	Here's a MONO controller message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 7E. The third byte can be any value in a range of values. The allowable range is shown later.
n = MIDI Channel	:0H - FH (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
vv = number of mono	:0H - FH (0 - 16)		Normally this value determines how many voices you wish the JV-90 to stack. But only a value of 1 here is honored.
Switched over "Assign mode" of patch common.			All this is saying is that you have a default "Assign mode" setting in the patch common screen. You can set a patch to play in Solo mode, or not, but this message overrides that default setting.
Recognized as All notes off, and set MODE4 (M = 1) at each part.			Besides changing a part to "Solo mode", it also turns off all notes that the part may have been sustaining up to that point. Also, what is being said here is that, no matter how many voices you tell the JV-90 to stack, it always uses only 1 voice when it receives a MONO controller message.

POLY			Indicates that what follows is how my JV-90 handles a POLY Controller message it receives. This message is used to set the sound module in polyphonic state (ie, plays more than 1 note at a time). In the JV-90's case, this is referred to as "Poly mode", and each part (of a Performance) can be set to poly mode individually. When in poly mode, a part's portamento and legato settings are ignored.
Status	Second	Third	Shows the format for a POLY controller message. It has 3 bytes.
BnH	7FH	00H	Here's a POLY controller message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The second byte is always 7F and the third byte is always 00.
n = MIDI Channel	:0H - FH (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can recognize Controller messages upon any of the 16 MIDI channels).
Switched over "Assign mode" of patch common.			All this is saying is that you have a default "Assign mode" setting in the patch common screen. You can set a patch to play in Poly mode, or not, but this message overrides that default setting.
Recognized as All notes off, and set MODE3 at each part.			Besides changing a part to "Poly mode", it also turns off all notes that the part may have been sustaining up to that point.
System Realtime message			This indicates that what follows is info about System Realtime types of messages. Such messages include MIDI Clock, MIDI Time Code quarter-frame messages, Song Select, etc. The JV-90 ignores all of them except Active Sense.
Active sensing			Indicates that what follows is how my JV-90 handles an Active Sense message it receives. Read the article What's Active Sense? for an indepth explanation of what Active Sense MIDI messages are and what they do.
Status			Shows the format for an Active Sense message. It is 1 byte.

FE**H**		The byte is always FE.
When the JV-90 receive "Active sensing"... blah blah blah		This is just an explanation of what Active Sense messages do. Read my above article instead.
System Exclusive Message		Indicates that what follows is the general format for a JV-90 System Exclusive message. There are many permutations of it since a variety of System Exclusive messages can be sent to the JV-90 to alter nearly any parameter that you can change from the front panel. The details for constructing those messages will be excruciatingly detailed in a later section entitled **Exclusive communications**. Right here, you're shown the general form of all SysEx messages that the JV-90 recognizes, but some of the "data" fields will have different values depending upon which parameter you're trying to change.
Status	Data	All this chart is showing you is that a System Exclusive message starts with a F0 byte, then has one or more data bytes following the F0, and finally ends with an F7. That's the MIDI standard for System Exclusive. Hopefully you already know this if you're going to be screwing around with System Exclusive messages.
F0**H**	ii**H** dd**H** .... ee**H**	Don't worry about this ii, dd, ee stuff. All it means is that, according to the MIDI standard for System Exclusive, there can be any number of data bytes here, each set to any value within an allowable range (ie, 0 to 127).
F7**H**		And then of course, there's a final F7 byte.
F0**H**	: System exclusive	Whereas the above showed you the MIDI standard for a System Exclusive message, now you're going to see the Roland general form. In Roland SysEx, after the opening F0 byte, the first data byte is always a Manufacturer ID of 41.
ii = Manufacturer ID	: 41**H** (65)	Roland's Manufacturer ID is always the first data byte after the F0, and is always 41.
dd .....ee = Data	: 00**H** - 7F**H** (0 to 127)	All this means is that there can be any number of data bytes here, each set to any value within the range 0 to 127.

F7**H**	: EOX (End of exclusive)	There's that final F7 byte.
Ignored when "Receive Exclusive" of the system common is OFF.		All this is saying is that if you go into the "System Common" settings of the JV-90, there's a setting called "Receive Exclusivee". If you set this to off, then the JV-90 ignores System Exclusive messages it receives.

2. TRANSMIT DATA

This indicates the start of the TRANSMIT DATA main section. This is divided into subsections exactly like the RECEIVE DATA main section. In other words, it typically begins with a NOTE OFF subsection. But here, the info is about how the unit creates Note Off messages. For example, the info here will tell you what range of Note numbers the unit's keyboard is capable of generating, as well as what range of velocities, for Note Off messages. Roland units that have no built-in controller, such as rackmount modules, may not be able to generate MIDI messages, and therefore the manual will omit this section.

Channel Voice Data

This indicates that what follows is info about the types of messages that the JV-90 can generate upon particular MIDI channels. Such types of messages are Note Off and On, Aftertouch, Channel Pressure, all of the Controller messages, Program Change, and Pitch Wheel.

Note off

Indicates that what follows is how my JV-90 keyboard generates Note Off messages.

Status	Second	Third

Shows the format for a MIDI Note Off message. (Note that the JV-90 generates real Note Off messages, rather than using Note On with 0 velocity). It has 3 bytes. The Status is the first byte. The Second is the note number. The third is the velocity amount for the note.

8n**H**	kk**H**	vv**H**	Here's a Note Off message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The kk means that it can be any note number within a range of values. The allowable range of values will be shown later. The vv means that it can be any velocity within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 keyboard can generate Note Off messages upon any of the 16 MIDI channels).
kk = Note number	:00**H** - 7F**H** (0 - 127)		Indicates that the allowable range for note number is 0 to 127. (ie, The JV-90 keyboard can generate the full range of note numbers. Even though it only has 76 keys, the keyboard can be "transposed" to generate any range of MIDI note numbers).
vv = velocity	:00**H** - 7F**H** (0 - 127)		Indicates that the allowable range for velocity is 0 to 127. (ie, The JV-90 keyboard can generate the full range for note velocity).
In the Performance mode, not transmit when the MIDI transmit switch is OFF at each Part.			This is a comment about how the JV-90 keyboard generates MIDI Note Off messages. All this is saying is that the keyboard generates Note Offs only for those MIDI Parts that have their MIDI transmit switch set to on. For example, assume that MIDI Part 1 is assigned to MIDI channel 1 and has its transmit siwtch on. Part 2 is assigned to channel 2 and has its transmit switch off. When you play the JV-90 keyboard, it generates Note Off messages only upon channel 1.
Note on			Indicates that what follows is how my JV-90 keyboard generates Note On messages.
Status	Second	Third	Shows the format for a MIDI Note On message. It has 3 bytes. The Status is the first byte. The Second is the note number. The third is the velocity amount for the note.
────────	────────	────────	

9n**H**	kk**H**	vv**H**	The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The kk means that it can be any note number within a range of values. The allowable range of values will be shown later. The vv means that it can be any velocity within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 keyboard can generate Note On messages upon any of the 16 MIDI channels).
kk = Note number	:00**H** - 7F**H** (0 - 127)		Indicates that the allowable range for note number is 0 to 127. (ie, The JV-90 can recognize the full range of note numbers).
vv = velocity	:00**H** - 7F**H** (0 - 127)		Indicates that the allowable range for velocity is 0 to 127. (ie, The JV-90 keyboard can generate the full range for note velocity).
In the Performance mode, not transmit when the MIDI transmit switch is OFF at each Part.			This is a comment about how the JV-90 keyboard generates MIDI Note On messages. All this is saying is that the keyboard generates Note Ons only for those MIDI Parts that have their MIDI transmit switch set to on. For example, assume that MIDI Part 1 is assigned to MIDI channel 1 and has its transmit siwtch on. Part 2 is assigned to channel 2 and has its transmit switch off. When you play the JV-90 keyboard, it generates Note On messages only upon channel 1.
Control Change			Indicates that what follows is how my JV-90 generates Controller messages. The JV-90 is capable of generating several types of controller messages. What follows is info about each type of controller it generates.
The function of the Modulation lever is determined by control number 1 (modulation).			This is a comment about the JV-90's Modulation lever. All this is saying is that it is hardwired to being controller number 1.
The function of the Hold1 pedal is determined by control number 64 (hold1).			This is a comment about the JV-90's Hold1 pedal (jack). All this is saying is that it is hardwired to being controller number 64.

Control numbers 0 - 95 can be assigned to Pedal1, Pedal2, and C1.

This is a comment about the other jacks and levers upon the JV-90. All this is saying is that they aren't hardwired to a particular controller number (like the Mod lever and Hold pedal jack are). So, you can change the function that these control. For example, you can take the C1 lever and assign it to controller number 07. That's the controller for Volume. So when you move the C1 lever, it generates MIDI Volume controller messages (and can therefore change the volume of the patch that the keyboard is playing). Or, you can assign the C1 lever to controller number 0A. That's the controller for Pan. So when you move the C1 lever, it now generates MIDI Pan controller messages instead (and can therefore change the pan position of the patch that the keyboard is playing). You can assign the C1 lever to any controller number in the range 0 to 95. See the chart Defined MIDI controllers for a list of all of the defined controller numbers, and what they control. (But note that the JV-90's built-in module responds only to a few of the defined controllers -- only those that were listed under "Control change" in the RECEIVE DATA section. So too, the Pedal 1 and Pedal 2 jacks can be assigned to any of those controller numbers. But note that the Pedal 1 jack can only set a controller to either the value 0 (ie, the pedal is released) or 127 (ie, the pedal is pressed down). Unlike the C1 lever or Pedal 2 jack, the Pedal 1 jack can't generate values inbetween 0 and 127.

Status	Second	Third

Shows the format for a Controller message that the JV-90 can generate when the controller number is 0 to 63 or 70 to 95. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.

Status	Second	Third	
Bn**H**	cc**H**	vv**H**	The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The cc means that the controller number can be any number within the range of 0 to 63 or 70 to 90. The vv means that the value can be any number within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can generate Controller messages upon any of the 16 MIDI channels).
cc = Control number	: 0**H** - 3F**H**, 46**H** - 5F**H** (0 - 63, 70 - 95)		Indicates the allowable range for controller number. The JV-90 can have its lever and/or pedals assigned to any controller number in the range of 0 to 63 or 70 - 95.
vv = Control value	:00**H** - 7F**H** (0 - 127)		Indicates the allowable range for the controller value. The JV-90 supports full controller range for controller numbers 0 to 63 or 70 to 95.
Status	Second	Third	Shows the format for a Controller message that the JV-90 can generate when the controller number is 64 to 69. It has 3 bytes. The Status is the first byte. The Second is the controller number. The third is the value for the controller.
Bn**H**	cc**H**	vv**H**	The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The cc means that the controller number can be any number within the range 64 to 69. The vv means that the value can be any number within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can generate Controller messages upon any of the 16 MIDI channels).
cc = Control number	: 40**H** - 45**H** (64 - 69)		Indicates the allowable range for controller number. The JV-90 can have its lever and/or pedals also assigned to any controller number in the range of 64 to 69.

vv = Control value	:00**H**, 7F**H** (0, 127)	Indicates the allowable range for the controller value. The JV-90 sends only the value 0 (ie, Off) or 127 (ie, On) for controller numbers 64 to 69. It doesn't send any of the inbetween values, even if you're using a lever.
Control change messages will not be transmitted when the following parameter are set:.		This is a comment about the JV-90's generation of controller messages. If you set some lever or pedal to use one of the following control numbers, and yet set that controller number to OFF in the TRANSMIT MIDI screen, then the JV-90 does not generate MIDI messages (although the levers or pedals may still affect the built-in sound module).
Control number	TRANSMIT MIDI	The chart lists the Controller number on the right, and its respective TRANSMIT MIDI screen setting on the left.
0, 32 (Bank select)	Bnk=OFF	If you set a lever/pedal to transmit Bank select coarse or fine, but set TRANSMIT MIDI screen's Bnk setting to OFF, then no such MIDI messages are generated.
1, 32 (Modulation)	Mod=OFF	If you set a lever/pedal to transmit Modulation coarse or fine, but set TRANSMIT MIDI screen's Mod setting to OFF, then no such MIDI messages are generated.
7, 39 (Volume)	Vol=OFF	If you set a lever/pedal to transmit Volume coarse or fine, but set TRANSMIT MIDI screen's Vol setting to OFF, then no such MIDI messages are generated.
2 - 6, 8 - 31, 34 - 38, 40 - 95	C.C=OFF	If you set a lever/pedal to transmit one of these controller numbers, but set TRANSMIT MIDI screen's C.C setting to OFF, then no such MIDI messages are generated.
Program Change		Indicates that what follows is how my JV-90 generates Program Change messages (as you manually push the Group, Patch, and other buttons that change between patches.
Status	Second	Shows the format for a Program Change message. It has 2 bytes. The Status is the first byte. The second is the program number.

CnH	pp	Here's the Program Change message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The pp means that the value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0H - FH (ch1 - ch16)	Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 can generate Program Change messages upon any of the 16 MIDI channels).
pp = Program number	:00H - 7FH (prog.1 - prog.128)	Indicates the allowable range. The full range of program numbers is supported. Note that program #1 is sent in the Program Change message as a 0 value.
Not transmit when "Transmit Program Change" of the System Common is OFF.		This is a comment about the JV-90's generation of Program Change messages. In the System Common screen, there is a setting called "Transmit Program Change", and if you set this to OFF, then the JV-90 generates no MIDI Program Change messages when you manually switch patches.

Channel Pressure		Indicates that what follows is how my JV-90 keyboard generates Channel Pressure messages.
Status	Second	Shows the format for a Channel Pressure message. It has 2 bytes. The Status is the first byte. The second is the pressure amount.
DnH	vv	Here's the Channel Pressure message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The vv means that the value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0H - FH (ch1 - ch16)	Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 keyboard can generate Channel Pressure messages upon any of the 16 MIDI channels).
vv = Pressure value	:00H... 7FH (0 - 127)	Indicates the allowable range. The full range of Channel Pressure is supported.

Not transmit when "Transmit Aftertouch" of the System Common is OFF.			This is a comment about the JV-90's generation of Channel Pressure messages. In the System Common screen, there is a setting called "Transmit Aftertouch", and if you set this to OFF, then the JV-90 generates no MIDI Channel Pressure messages when you press harder/softer on the keys.

Pitch bend change

Indicates that what follows is how my JV-90 Pitch lever generates Pitch Wheel messages.

Status	Second	Third	
			Shows the format for a Pitch Wheel message. It has 3 bytes. The Status is the first byte. The Second is the coarse pitch change. The third is the fine pitch change.
────	────	────	
EnH	**mm**H	**ll**H	Here's the Pitch Wheel message. The n means that the Status byte's channel can be any value within a range of values. The allowable range of values will be shown later. The mm means that the coarse value can be any value within a range of values. The allowable range of values will be shown later. The ll means that the fine value can be any value within a range of values. The allowable range of values will be shown later.
n = MIDI Channel	:0**H** - F**H** (ch1 - ch16)		Indicates that the allowable range for MIDI channel is 0 to 15. (ie, The JV-90 pitch lever can generate Pitch Wheel messages upon any of the 16 MIDI channels).
mm, ll = Pitch bend change	:00**H**, 00**H** - 7F**H**, 7F**H** (-8192 - +8191)		Indicates the allowable range. Supports the full range for Pitch Wheel.
Not transmit when "Transmit Pitch bend" of the System Common is OFF.			This is a comment about the JV-90's generation of Pitch Wheel messages. In the System Common screen, there is a setting called "Transmit Pitch bend", and if you set this to OFF, then the JV-90 generates no MIDI Pitch Wheel messages when you move the pitch lever.

System Realtime message

This indicates that what follows is info about System Realtime types of messages. Such messages include MIDI Clock, MIDI Time Code quarter-frame messages, Song Select, etc. The JV-90 does not generate any of those except Active Sense.

Active sensing

	Indicates that what follows is how my JV-90 generates Active Sense messages.
Status	Shows the format for an Active Sense message. It is 1 byte.
FE**H**	The byte is always FE.
This message transmit at about 300 milli-second interval.	If the JV-90 hasn't been generating any other MIDI messages for awhile, it starts sending an Active Sense byte every 300 milli-seconds.

System Exclusive Message	Indicates that what follows is the general format for a JV-90 System Exclusive message. There are many permutations of it since the JV-90 can generate a variety of System Exclusive messages when it dumps various parameter settings over MIDI. The details for constructing those messages will be excruciatingly detailed in a later section entitled **Exclusive communications**. Right here, you're shown the general form of all SysEx messages that the JV-90 generates, but some of the "data" fields will have different values depending upon which parameter(s) is being dumped.
Status Data	All this chart is showing you is that a System Exclusive message starts with a F0 byte, then has one or more data bytes following the F0, and finally ends with an F7. That's the MIDI standard for System Exclusive. Hopefully you already know this if you're going to be screwing around with System Exclusive messages.

F0**H**	ii**H** dd**H** .... ee**H**	Don't worry about this ii, dd, ee stuff. All it means is that, according to the MIDI standard for System Exclusive, there can be any number of data bytes here, each set to any value within an allowable range (ie, 0 to 127).
F7**H**		And then of course, there's a final F7 byte.
F0**H**	: System exclusive	Whereas the above showed you the MIDI standard for a System Exclusive message, now you're going to see the Roland general form. In Roland SysEx, after the opening F0 byte, the first data byte is always a Manufacturer ID of 41.

ii = Manufacturer ID	: 41**H** (65)	Roland's Manufacturer ID is always the first data byte after the F0, and is always 41.
dd .....ee = Data	: 00**H** - 7F**H** (0 to 127)	All this means is that there can be any number of data bytes here, each set to any value within the range 0 to 127.
F7**H**	: EOX (End of exclusive)	There's that final F7 byte.
Refer to section 3, 4.		All this is saying is that you should also read the sections entitled "Exclusive communications" and "Parameter Address Map". These give the excruciating details of the JV-90's System Exclusive capabilities.

What follows here are the "Exclusive communications" and "Parameter address map" sections. These are covered in another article entitled Roland System Exclusive Implementation.

When the MIDI spec was first developed, it wasn't foreseen that anyone would need more than 128 patches on a given module. (Back then, most didn't even have anywhere near that number of patches). So, the MIDI Program Change message was hardwired to have a limit of counting only from 1 to 128.

Later, modules with more than 128 patches came on the market. People needed some way of being able to switch to these extra patches, but which was still compatible with the old way of switching patches. The manufacturers adopted a scheme of arranging patches in "banks" (ie, groups of usually 128 patches). For example, the first 128 patches in a module may be "bank 1". The next 128 patches may be "bank 2". Etc. Theoretically, there can be up to 16,384 banks on a module.

The technique that the manufacturers adopted for MIDI control over patch changing is to have the musician first select the bank that contains his desired patch, and then select the patch within that bank. For example, assume that a musician wants to select the patch "Gungho" which happens to be the third patch in the second bank. First the musician would have to send one or two (depending upon how the manufacturer arranged patches into banks) MIDI messages to select the second bank (MIDI counts this as bank 1, since MIDI considers bank number 0 to actually be the first bank). Then, the musician sends a MIDI message to select the third patch (again, MIDI considers patch number 0 to be the first patch in a bank, so the third patch would actually be number 2). So, selecting a patch is a two-step (ie, 2 or 3 message) process. First, you send the Bank Select message(s) to switch to the desired bank. Then you send an ordinary Program Change message to select which one of the 128 possible patches in that bank you desire.

The Bank Select messages are actually MIDI Controller messages, just like Volume, Pan, Sustain Pedal, Wind, and other controllers. Specifically, the controller number for the "Most Significant Byte" (ie, MSB) of Bank Select is controller 0. The controller number for "Least Significant Byte" (ie, LSB) of Bank Select is controller 32. The data for these messages are the bank number you want to select. (Sometimes the MSB Bank Select is referred to as the coarse adjustment for bank, and the LSB Bank Select is referred to as the fine adjustment).

NOTE: We need to use 2 messages to contain the bank number because, due to MIDI's design, it's not possible to transmit a value greater than 128 in just one controller message. Remember that a bank number can go as high as 16,384, and you need 2 MIDI controller messages to send such a large value. But, since most modules do not have more than 128 banks anyway, these modules typically only use the MSB message (ie, controller number 0) to select bank, and ignore any LSB message (ie, controller number 32). So then, here are the two messages (in hexadecimal, assuming MIDI channel 1, and assuming that the module only uses the MSB Bank Select controller) to select that "Gungho" patch:

B0 00 01	Switch to bank 2 (NOTE: only the MSB message needed)
C0 02	Switch to the third patch in this bank

The patch doesn't actually change until the module receives the Program Change message. The Bank Select message(s) must precede the Program Change message. Otherwise, the module would select the third patch in whatever bank the module was currently set to. (In other words, when the module receives Bank Select messages, it doesn't change any patches. Rather, it simply notes what the "current bank" is, and any subsequent Program Change messages will switch to patches within that bank. What this also means is that you don't necessarily need to send Bank Select before every individual Program Change. If you're switching to a patch in the same bank as whatever patch you've currently selected, then you can omit the Bank Select messages since the module will then assume the same bank).

The following, real-world example concerns a Roland module. Before proceeding, you should read The architecture of Roland sound modules if you're unfamiliar with the way that Roland arranges Patches into "Banks".

Let's look at the JV-90 as an example. Look under the "MIDI Implementation" chapter of the manual. Follow the "Receive Data" section until you get to "Control Change" and "Bank Select". (The article Understanding the MIDI Implementation in your Roland manual helps guide you through this chapter of the manual). As is so typical of Roland, the company uses the MIDI spec in a clever way. Rather than using the data bytes of both the MSB and LSB messages "added" together to make up a single bank number, Roland sort of uses the MSB message to select the "component" that

you wish to access, and the LSB message to select the bank within that component (ie, some components contain several banks, such as the factory patches in ROM).

Look at the following list: (Note that I'm using a fixed pitch font to line up the columns)

```
Bank    Bank    Program       Media
MSB     LSB     Change
80       0      1 - 64        Internal (ie, User custom presets in RAM)
80       0      65 - 128      Data Card
81       0      1 - 64        JV-90 Preset A (ie, factory patches on Voice Expansion)
81       0      65 - 128      JV-90 Preset B (ie, factory patches on Voice Expansion)
81       1      1 - 64        Preset A (ie, factory patches in ROM)
81       1      65 - 128      Preset B (ie, factory patches in ROM)
81       2      1 - 64        Preset C (ie, factory patches in ROM)
81       2      65 - 128      Preset D (ie, factory patches in ROM)
82       0      1 - 64        Data Card
83       0      1 - 128       PCM Card
84       0      1 - 128       Expansion Board (ie, factory patches on Wave Expansion)
84       1      1 - 128       Expansion Board (129 - 256)
```

Let's say that you want to select the third patch on the factory presets "B" group. Look at "Preset B". In order to select from these patches, first you must send an MSB Bank Select with the decimal value 81 (ie, hexadecimal 51). Next, you must send an LSB Bank Select with the value 1. Finally, you can send a Program Change. The patches in this bank are numbered 65 to 128 (ie, the first patch in the "Preset B" is number 65 -- actually it is 64 in our Program Change message since we always subtract 1 to account for MIDI assuming 0 as the first patch in a bank). So the third patch in Preset B is #67 (really 66, or hexadecimal 42). Here are the 3 messages (assuming MIDI channel 1):

B0 00 51	MSB (Coarse) Bank select
B0 20 01	LSB (Fine) Bank select
C0 42	Program Change

Let's select the first patch in User RAM (ie, "Internal" as Roland calls it in the above chart). We need an MSB Bank Select of 80 (hexadecimal 50), an LSB Bank Select of 0, and then a Program Change to patch 1 (actually 0) since the patches are numbered 1 to 64 here.

B0 00 50	MSB (Coarse) Bank select
B0 20 00	LSB (Fine) Bank select
C0 00	Program Change

Of course, if you subsequently want to select another patch in this same bank, then you can skip sending the 2 Bank Select messages. After all, "User RAM" is now the current bank.

Although most Roland sound modules use Bank Select controller in ways similiar to the JV-90, your unit may have different components and different bank numbers used to select them. You need to look up the appropriate chart in your Roland manual. It will be found in the "MIDI Implementation" chapter, under the RECEIVE DATA main section, CONTROL CHANGE subsection, and "Bank Select" sub-subsection.

Some sequencer programs allow you to setup an "instrument definition" wherein you simply enter the above 3 messages and then specify the name of the patch. For example, maybe this patch is called "Gungho". You can tell your sequencer that whenever you select "Gungho" from a list of patch names, the sequencer will then automatically insert the above 3 messages into your track at some specified point where you want to switch to the "Gungho" patch. It's a lot easier to setup that instrument definition once, detailing the messages to insert for each patch name, and then from there, only deal with selecting patch names from some list rather than needing to remember what Bank Select and Program Change messages need to be inserted. But of course, in order to create such a definition, you need to understand the above (once anyway -- and then you can forget about it all -- after the nightmares subside).

For example, you can go into CakeWalk's Settings->Assign Instruments menu item, and click on "Define Instruments" to create a new instrument definition for your MIDI module. Unfortunately, I think that CakeWalk's instrument definition is most unintuitive and convoluted because you have to first dig your way down to the "Define names" dialog box, skipping a whole dialog of settings, in order to enter all the banks of patch names prior to even defining what sort of bank select your module uses and what those banks are. This "bottom up" approach may save some time if you're dealing with a module whose banks tend to include a lot of duplicate patches, but it's really a pain for modules whose banks do not contain duplicate patches. I'd prefer to first choose the "bank select method", and then have CakeWalk guide me through the naming/numbering of a bank, immediately followed by each patch name/number in that bank, and so on for however many more banks there are. Worse, it appears that you have to enter the Banks by editing a rather freeform text file. For this reason, the creation of CakeWalk instrument definitions is a perilous and difficult undertaking for all but the most patient and experienced.

Once you have an instrument definition, you can insert a Bank Select (MSB and LSB) and Program Change in a CakeWalk track simply using the Insert menu's "Bank/Patch change".

The very first Bank Select and Program Change events, if they appear before any note events in a track, are automatically extracted by CakeWalk and used to set the "Bank" and "Patch" fields in CakeWalk's track display. So, you use these two fields to enter your values for any initial Bank and Program Change at playback start. Then, if you want to change the Bank and/or Patch somewhere in the middle of the track (ie, after notes have already been played, you'll have use CakeWalk's Insert menu to insert "Bank/Patch change". Choose a Bank and a Patch, and enter the desired time where it should be inserted, and CakeWalk will insert the Bank Select and Program Change events in the track at that desired point. You should see these shown in the Event List. If you don't have a CakeWalk instrument definition file for your MIDI module, then you'll need to manually insert the desired Bank controller events yourself using Insert menu's "Series of Controllers". Do this once to choose "Bank Select" controller (ie, the coarse, or MSB one) and its desired Value and Time. (If your unit also requires the LSB or "fine" value for Bank Select, you'll need to repeat this to choose "Bank Select (fine)" controller and set its value/time). Then use the "Bank/Patch change" just to pick out one of the 128 patches in the bank.

Windows provides a High level API for playing both digital audio and MIDI. Windows also provides a high level API for recording digital audio, but not yet a corresponding API for recording MIDI. The goal of the high level API is for the operating system to do as much work as possible in playing or recording digital audio or MIDI data, and your program merely gives overall "instructions" to the operating system such as telling it the name of the MIDI or WAVE file to load and play.

Furthermore, one of the goals of the High level API is to allow the playback or recording to be done by the operating system in the background while your program goes on to do other things. In order to implement such a scheme, the Windows operating system incorporates a software entity (ie, a software library/driver/program/however-you-want-to-think-of-it) known as the "MCI Wave Device" which can play or record a digital audio waveform all by itself after receiving a few instructions from your program. It does all of the Low level API stuff on your behalf. The operating system also incorporates a software entity known as the "MCI Sequencer Device" which can play a MIDI song (ie, an entire song stored as a series of MIDI messages) all by itself after receiving a few instructions from your program.

In fact, because multimedia programmers often have to support both digital audio and MIDI in their programs, Windows tries to make things even simpler by having you pass your instructions (ie, commands) to the MCI Wave Device in the same way (and using the same API function) as you pass instructions to the MCI Sequencer Device. In other words, if you learn how to send commands to the MCI Wave Device, then you already know how to send commands to the MCI Sequencer Device (or vice versa). (You also will know how to send commands to other MCI devices such as the Audio CD Player and Video Player).

Indeed, many of the documented commands are the same for both. For example, you can tell the MCI Wave Device to "start playback" (of a waveform) just like you can tell the MCI Sequencer Device to "start playback" (of a MIDI song). It doesn't matter to you that these devices are dealing with entirely different data and/or audio hardware, because after all, those devices are managing the actual data and hardware entirely on their own. You need give them only some general instructions to follow, and one such command for both of them could indeed be to "start playback". In other words, if you know the commands to tell the MCI Wave Device to do something simple such as start playback, then you already know the same commands for the MCI Sequencer Device (or vice versa).

Message or String interfaces

In order to make it even easier for a programmer to use these devices, they offer two different ways to send them a command.

In one approach, you can pass them binary values, or addresses of structures filled in with values. For example, let's say that you want the MCI Wave Device to record a waveform. Prior to doing this, you need to tell the MCI Wave Device your desired sample rate, bit resolution (ie, 8 for 8-bit, 16 for 16-bit), and number of channels (ie, 1 for MONO, 2 for STEREO, etc). You can fill in a structure that has an unsigned long field where you store your desired sample rate, plus another unsigned long field

where you store the number of channels of digital audio you wish recorded, and another field where you store your desired bit resolution. And then you call a function that passes this structure to the MCI Wave Device, as well as passing a "Command" value that indicates to the MCI Wave Device that it should expect this structure, and to use those values to **set** its record parameters. Passing binary values and structures to an MCI device is referred to as using the "Command **message** interface". You use the function mciSendCommand() to send commands using this approach.

Furthermore, when you use mciSendCommand() to command the MCI Sequencer or Wave Audio Device to return some information to you, (for example if you ask it to tell you what bit resolution and sample rate is currently set for recording), then these devices will fill in structures with binary values. For example you can pass that above structure and have the Wave Audio device fill in its sample rate, channels, and bit resolution fields.

In the other approach, you instead pass null-terminated strings. For example, let's take that same job of telling the MCI Wave Device to set itself up for recording a 11025 KHz sample rate, stereo, 8-bit waveform. You instead format a null-terminated string that looks like "**set** *bitspersample* 8 *channels* 2 *samplespersec* 11025" where **set** is the command and the other (italicized) words are the record parameters followed by the desired value for each. It's really the same command that you issued above, but now it takes the form of a string instead of passed binary values/structures. Passing strings to an MCI device is referred to as using the "Command **string** interface". You use the function mciSendString() to send commands using this approach.

When you use mciSendString() to command the MCI Sequencer or Wave Audio Device to return some information back to you, (for example if you ask it to tell you what bit resolution and sample rate is currently set for recording), then these devices will return strings like what you see above. For example it may return a string that looks like "bitspersample 8 channels 1 samplespersec 11025". Your program would then have to parse that string to extract the individual record parameters.

Why 2 different approaches to do the same thing? The MCI Sequencer and Wave Audio devices are meant to be controlled by programs written in languages where it is easier to deal with simply sending (and receiving back) binary values and structures (such as C), as well as languages where it is much easier to deal with strings (such as Basic). And there are other cases where you may find it easier yourself to use strings, or easier to use binary values. You've got your choice (or you can use both).

Overview of using the device

To use the MCI Sequencer or Wave Audio Device, you need to first open that device. To open either of these devices, you issue an "open" command using mciSendString() or mciSendCommand() (depending upon whether you want to specify your open command by passing formatted strings, or binary values/structures, respectively).

After opening the device, you can then issue commands directly to it using mciSendString() or

mciSendCommand(). Some commands you issue can cause the device to return data to you.

After you're done using the device (and have no further use for it), you must close that device. To close the device, you send it a "close" command using mciSendString() or mciSendCommand().

The Wave Audio and Sequencer Devices are referred to as *compound devices*. This is just a fancy way of saying that when you open the device, it has to be associated with a WAVE file or a MIDI file (for the Wave Audio or Sequencer Device respectively). In essense, when you open the device, you tell it what WAVE or MIDI file to also open. If you're going to subsequently do a play operation, this will be the file that gets played. If you're going to do a record operation, this will be file into which recorded data is stored. Indeed, for any operation that you do, this is the file that is operated upon.

In fact, you can open either device several times (simultaneously) if you want to open several WAVE or MIDI files. Since you need to specify a particular WAVE or MIDI file each time that you open the device, the act of opening the device is sort of equivalent to opening a MIDI or WAVE file. The device ID (or alias) you obtain from each instance of opening the device is akin to a file handle. In essense, when you open the Wave Audio device, think of it as if you were opening a WAVE file that can perform operations upon itself, such as to playback any waveform that is stored in it (using whatever default Digital Audio Out hardware device is in the system), or to store (ie, record) the digital audio input (of the default Digital Audio In hardware device). When you open the Sequencer device, think of it as if you were opening a MIDI file that also can perform operations upon itself, such as to playback any MIDI data that is stored in it (using whatever default MIDI Out hardware device is in the system). You can also give commands to these files to have them perform other operations upon themselves.

And of course, if you opened several files, you'd have to close each one. So too with opening multiple instances of the Wave Audio or Sequencer Devices.

Although many of the commands are the same for both the Wave Audio and Sequencer Devices, and therefore this article could detail the common operations such as open, play, record, etc. for both devices, instead the Wave Audio Device is discussed further in High level Digital Audio API, and the Sequencer Device is discussed further in High level MIDI API.

WAVE File Format is a file format for storing digital audio (waveform) data. It supports a variety of bit resolutions, sample rates, and channels of audio. This format is very popular upon IBM PC (clone) platforms, and is widely used in professional programs that process digital audio waveforms. It takes into account some peculiarities of the Intel CPU such as little endian byte order.

This format uses Microsoft's version of the Electronic Arts Interchange File Format method for storing data in "chunks". You should read the article About Interchange File Format before proceeding.

Data Types

A C-like language will be used to describe the data structures in the file. A few extra data types that are not part of standard C, but which will be used in this document, are:

pstring   Pascal-style string, a one-byte count followed by that many text bytes. The total number of bytes in this data type should be even. A pad byte can be added to the end of the text to accomplish this. This pad byte is not reflected in the count.

ID   A chunk ID (ie, 4 ASCII bytes) as described in About Interchange File Format.

Also note that when you see an array with no size specification (e.g., char ckData[];), this indicates a variable-sized array in our C-like language. This differs from standard C arrays.

Constants

Decimal values are referred to as a string of digits, for example 123, 0, 100 are all decimal numbers. Hexadecimal values are preceded by a 0x - e.g., 0x0A, 0x1, 0x64.

Data Organization

All data is stored in 8-bit bytes, arranged in Intel 80x86 (ie, little endian) format. The bytes of multiple-byte values are stored with the low-order (ie, least significant) bytes first. Data bits are as follows (ie, shown with bit numbers on top, "lsb" stands for "least significant byte" and "msb" stands for "most significant byte"):

```
         7   6   5   4   3   2   1   0
       +-----------------------+
 char: | lsb                   |
       +-----------------------+

         7   6   5   4   3   2   1   0 15  14  13  12  11  10   9   8
       +-----------------------+-----------------------+
short: | lsb      byte 0       |        byte 1    msb  |
       +-----------------------+-----------------------+

         7   6   5   4   3   2   1   0 15  14  13  12  11  10   9   8 23  22  21  20  19  18  17  16 31
30 29 28 27 26 25 24
       +-----------------------+-----------------------+----------------------
+-----------------------+
 long: | lsb      byte 0       |         byte 1        |         byte 2
|      byte 3      msb  |
       +-----------------------+-----------------------+----------------------
+-----------------------+
```
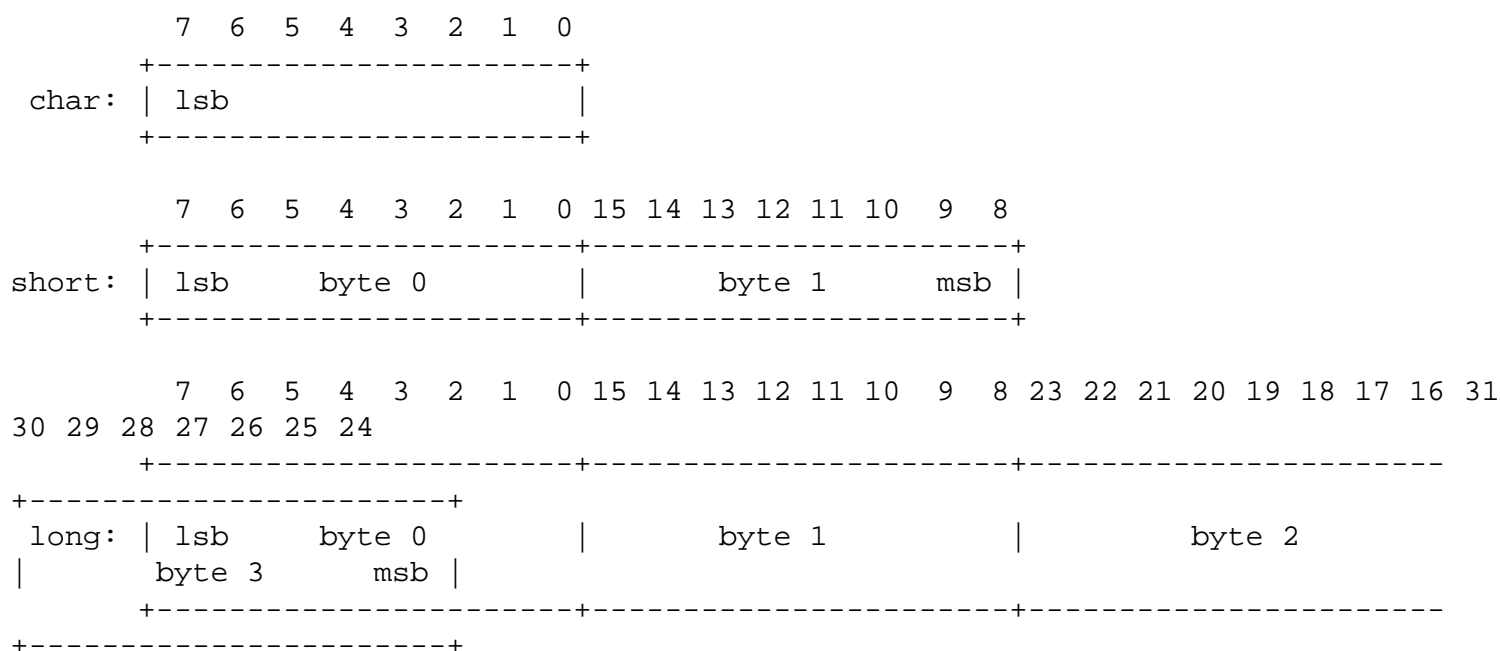
File Structure

A WAVE file is a collection of a number of different types of chunks. There is a required Format ("fmt ") chunk which contains important parameters describing the waveform, such as its sample rate. The Data chunk, which contains the actual waveform data, is also required. All other chunks are optional. Among the other optional chunks are ones which define cue points, list instrument parameters, store application-specific information, etc. All of these chunks are described in detail in the following sections of this document.

All applications that use WAVE must be able to read the 2 required chunks and can choose to selectively ignore the optional chunks. A program that copies a WAVE should copy all of the chunks in the WAVE, even those it chooses not to interpret.

There are no restrictions upon the order of the chunks within a WAVE file, with the exception that the Format chunk must precede the Data chunk. Some inflexibly written programs expect the Format chunk as the first chunk (after the RIFF header) although they shouldn't because the specification doesn't require this.

Here is a graphical overview of an example, minimal WAVE file. It consists of a single WAVE containing the 2 required chunks, a Format and a Data Chunk.

```
 _____
|  RIFF WAVE Chunk               |
|     groupID  = 'RIFF'          |
|     riffType = 'WAVE'          |
|     _____     |
|    |  Format Chunk        |    |
|    |     ckID = 'fmt '    |    |
|    |_____|    |
|     _____     |
|    |  Sound Data Chunk    |    |
|    |     ckID = 'data'    |    |
|    |_____|    |
|_____|
```
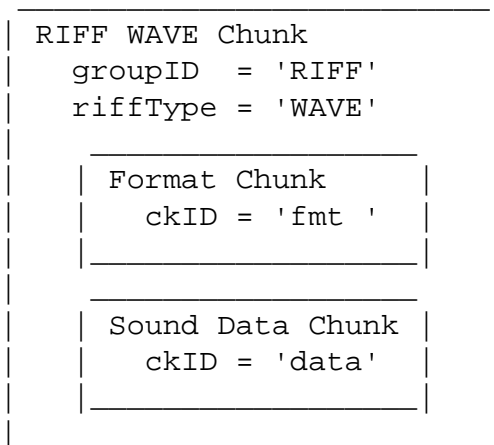
A Bastardized Standard

The WAVE format is sort of a bastardized standard that was concocted by too many "cooks" who didn't properly coordinate the addition of "ingredients" to the "soup". Unlike with the AIFF standard which was mostly designed by a small, coordinated group, the WAVE format has had all manner of much-too-independent, uncoordinated aberrations inflicted upon it. The net result is that there are far too many chunks that may be found in a WAVE file -- many of them duplicating the same information found in other chunks (but in an unnecessarily different way) simply because there have been too many programmers who took too many liberties with unilaterally adding their own additions to the WAVE format without properly coming to a concensus of what everyone else needed (and therefore it encouraged an "every man for himself" attitude toward adding things to this "standard"). One example is the Instrument chunk versus the Sampler chunk. Another example is the Note versus Label chunks in an Associated Data List. I don't even want to get into the totally irresponsible proliferation of compressed formats. (ie, It seems like everyone and his pet Dachshound has come up with some compressed version of storing wave data -- like we need 100 different ways to do that). Furthermore, there are lots of inconsistencies, for example how 8-bit data is unsigned, but 16-bit data is signed.

I've attempted to document only those aspects that you're very likely to encounter in a WAVE file. I suggest that you concentrate upon these and refuse to support the work of programmers who feel the need to deviate from a standard with inconsistent, proprietary, self-serving, unnecessary extensions. Please do your part to rein in half-assed programming.
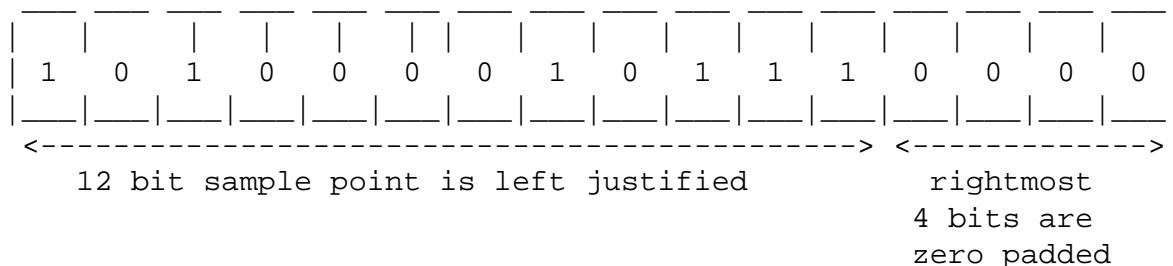
Sample Points and Sample Frames

A large part of interpreting WAVE files revolves around the two concepts of sample points and sample frames.

A sample point is a value representing a sample of a sound at a given moment in time. For waveforms with greater than 8-bit
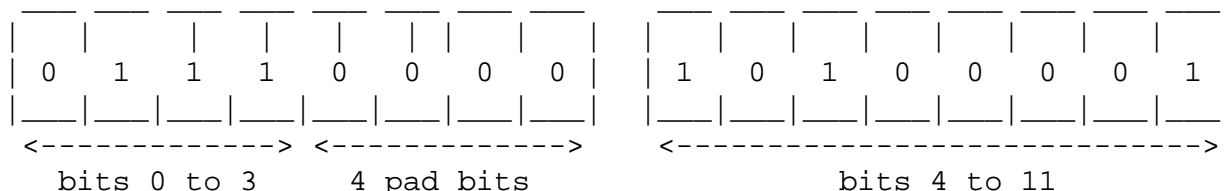
resolution, each sample point is stored as a linear, 2's-complement value which may be from 9 to 32 bits wide (as determined by the wBitsPerSample field in the Format Chunk, assuming PCM format -- an uncompressed format). For example, each sample point of a 16-bit waveform would be a 16-bit word (ie, two 8-bit bytes) where 32767 (0x7FFF) is the highest value and -32768 (0x8000) is the lowest value. For 8-bit (or less) waveforms, each sample point is a linear, unsigned byte where 255 is the highest value and 0 is the lowest value. Obviously, this signed/unsigned sample point discrepancy between 8-bit and larger resolution waveforms was one of those "oops" scenarios where some Microsoft employee decided to change the sign sometime after 8-bit wave files were common but 16-bit wave files hadn't yet appeared.

Because most CPU's read and write operations deal with 8-bit bytes, it was decided that a sample point should be rounded up to a size which is a multiple of 8 when stored in a WAVE. This makes the WAVE easier to read into memory. If your ADC produces a sample point from 1 to 8 bits wide, a sample point should be stored in a WAVE as an 8-bit byte (ie, unsigned char). If your ADC produces a sample point from 9 to 16 bits wide, a sample point should be stored in a WAVE as a 16-bit word (ie, signed short). If your ADC produces a sample point from 17 to 24 bits wide, a sample point should be stored in a WAVE as three bytes. If your ADC produces a sample point from 25 to 32 bits wide, a sample point should be stored in a WAVE as a 32-bit doubleword (ie, signed long). Etc.

Furthermore, the data bits should be left-justified, with any remaining (ie, pad) bits zeroed. For example, consider the case of a 12-bit sample point. It has 12 bits, so the sample point must be saved as a 16-bit word. Those 12 bits should be left-justified so that they become bits 4 to 15 inclusive, and bits 0 to 3 should be set to zero. Shown below is how a 12-bit sample point with a value of binary 101000010111 is formatted left-justified as a 16-bit word.
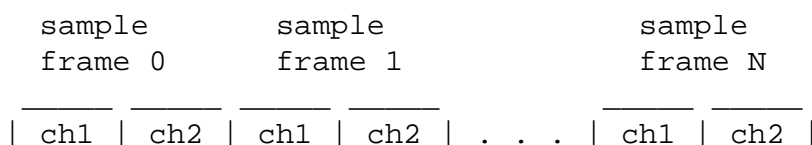
```
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1   0   1   0   0   0   0   1   0   1   1   1   0   0   0   0 |
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
 <-------------------------------------------> <------------->
    12 bit sample point is left justified         rightmost
                                                  4 bits are
                                                  zero padded
```

But note that, because the WAVE format uses Intel little endian byte order, the LSB is stored first in the wave file as so:

```
 ___ ___ ___ ___ ___ ___ ___ ___      ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |   |
| 0   1   1   1   0   0   0   0 |    | 1   0   1   0   0   0   0   1 |
|___|___|___|___|___|___|___|___|    |___|___|___|___|___|___|___|___|
 <------------->  <------------->     <----------------------------->
  bits 0 to 3       4 pad bits                  bits 4 to 11
```

For multichannel sounds (for example, a stereo waveform), single sample points from each channel are interleaved. For example, assume a stereo (ie, 2 channel) waveform. Instead of storing all of the sample points for the left channel first, and then storing all of the sample points for the right channel next, you "mix" the two channels' sample points together. You would store the first sample point of the left channel. Next, you would store the first sample point of the right channel. Next, you would store the second sample point of the left channel. Next, you would store the second sample point of the right channel, and so on, alternating between storing the next sample point of each channel. This is what is meant by interleaved data; you store the next sample point of each of the channels in turn, so that the sample points that are meant to be "played" (ie, sent to a DAC) simultaneously are stored contiguously.

The sample points that are meant to be "played" (ie, sent to a DAC) simultaneously are collectively called a **sample frame**. In the example of our stereo waveform, every two sample points makes up another sample frame. This is illustrated below for that stereo example.

```
  sample          sample                    sample
  frame 0         frame 1                   frame N

 _____ _____ _____ _____              _____ _____
| ch1  | ch2  | ch1  | ch2  | . . .      | ch1  | ch2  |
```

```
|_____|_____|_____|_____|        |_____|_____|

 _____
|      | = one sample point
|_____|
```

For a monophonic waveform, a sample frame is merely a single sample point (ie, there's nothing to interleave). For multichannel waveforms, you should follow the conventions shown below for which order to store channels within the sample frame. (ie, Below, a single sample frame is displayed for each example of a multichannel waveform).

```
 channels        1          2
               _____ _____
              | left    | right   |
 stereo       |         |         |
              |_____|_____|


                1          2          3
               _____ _____ _____
              | left    | right   | center  |
 3 channel    |         |         |         |
              |_____|_____|_____|


                1          2          3          4
               _____ _____ _____ _____
              | front   | front   | rear    | rear    |
 quad         | left    | right   | left    | right   |
              |_____|_____|_____|_____|


                1          2          3          4
               _____ _____ _____ _____
              | left    | center  | right   | surround|
 4 channel    |         |         |         |         |
              |_____|_____|_____|_____|


                1          2          3          4          5          6
               _____ _____ _____ _____ _____ _____
              | left    | left    | center  | right   | right   |surround |
 6 channel    | center  |         |         | center  |         |         |
              |_____|_____|_____|_____|_____|_____|
```

The sample points within a sample frame are packed together; there are no unused bytes between them. Likewise, the sample frames are packed together with no pad bytes.

Note that the above discussion outlines the format of data within an uncompressed data chunk. There are some techniques of storing compressed data in a data chunk. Obviously, that data would need to be uncompressed, and then it will adhere to the above layout.

Format chunk

The Format (fmt) chunk describes fundamental parameters of the waveform data such as sample rate, bit resolution, and how many channels of digital audio are stored in the WAVE.

```
#define FormatID 'fmt '   /* chunkID for Format Chunk. NOTE: There is a space at the
end of this ID. */
```

```
typedef struct {
  ID              chunkID;
  long            chunkSize;

  short           wFormatTag;
  unsigned short  wChannels;
  unsigned long   dwSamplesPerSec;
  unsigned long   dwAvgBytesPerSec;
  unsigned short  wBlockAlign;
  unsigned short  wBitsPerSample;

/* Note: there may be additional fields here, depending upon wFormatTag. */

} FormatChunk;
```

The ID is always "**fmt** ". The chunkSize field is the number of bytes in the chunk. This does not include the 8 bytes used by ID and Size fields. For the Format Chunk, chunkSize may vary according to what "format" of WAVE file is specified (ie, depends upon the value of wFormatTag).

WAVE data may be stored without compression, in which case the sample points are stored as described in **Sample Points and Sample Frames**. Alternately, different forms of compression may be used when storing the sound data in the Data chunk. With compression, each sample point may take a differing number of bytes to store. The wFormatTag indicates whether compression is used when storing the data.

If compression is used (ie, WFormatTag is some value other than 1), then there will be additional fields appended to the Format chunk which give needed information for a program wishing to retrieve and decompress that stored data. The first such additional field will be an unsigned short that indicates how many more bytes have been appended (after this unsigned short). Furthermore, compressed formats must have a Fact chunk which contains an unsigned long indicating the size (in sample points) of the waveform after it has been decompressed. There are (too) many compressed formats. Details about them can be gotten from Microsoft's web site.

If no compression is used (ie, wFormatTag = 1), then there are no further fields.

The wChannels field contains the number of audio channels for the sound. A value of 1 means monophonic sound, 2 means stereo, 4 means four channel sound, etc. Any number of audio channels may be represented. For multichannel sounds, single sample points from each channel are interleaved. A set of interleaved sample points is called a sample frame.

The actual waveform data is stored in another chunk, the Data Chunk, which will be described later.

The dwSamplesPerSec field is the sample rate at which the sound is to be played back in sample frames per second (ie, Hertz). The 3 standard MPC rates are 11025, 22050, and 44100 KHz, although other rates may be used.

The dwAvgBytesPerSec field indicates how many bytes play every second. dwAvgBytesPerSec may be used by an application to estimate what size RAM buffer is needed to properly playback the WAVE without latency problems. Its value should be equal to the following formula rounded up to the next whole number:

dwSamplesPerSec * wBlockAlign

The wBlockAlign field should be equal to the following formula, rounded to the next whole number:

wChannels * (wBitsPerSample / 8)

Essentially, wBlockAlign is the size of a sample frame, in terms of bytes. (eg, A sample frame for a 16-bit mono wave is 2 bytes. A sample frame for a 16-bit stereo wave is 4 bytes. Etc).

The wBitsPerSample field indicates the bit resolution of a sample point (ie, a 16-bit waveform would have wBitsPerSample = 16).

One, and only one, Format Chunk is required in every WAVE.

Data chunk

The Data (data) chunk contains the actual sample frames (ie, all channels of waveform data).

```
#define DataID 'data'  /* chunk ID for data Chunk */

typedef struct {
  ID              chunkID;
  long            chunkSize;

  unsigned char   waveformData[];
} DataChunk;
```

The ID is always **data**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields nor any possible pad byte needed to make the chunk an even size (ie, chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing pad byte).

Remember that the bit resolution, and other information is gotten from the Format chunk.

The following discussion assumes uncompressed data.

The waveformData array contains the actual waveform data. The data is arranged into what are called *sample frames*. For more information on the arrangment of data, see "Sample Points and Sample Frames".

You can determine how many bytes of actual waveform data there is from the Data chunk's chunkSize field. The number of sample frames in waveformData is determined by dividing this chunkSize by the Format chunk's wBlockAlign.

The Data Chunk is required. One, and only one, Data Chunk may appear in a WAVE.

Another way of storing waveform data

So, you're thinking "This WAVE format isn't that bad. It seems to make sense and there aren't all that many inconsistencies, duplications, and inefficiencies". You fool! We're just getting started with our first excursion into unnecessary inconsistencies, duplications, and inefficiency.

Sure, countless brain-damaged programmers have inflicted literally dozens of compressed data formats upon the Data chunk, but apparently someone felt that even this wasn't enough to make your life difficult in trying to support WAVE files. No, some half-wit decided that it would be a good idea to screw around with storing waveform data in something other than one Data chunk. NOOOOOOOOOOOOOOO!!!!!!

For some god-forsaken reason, someone came up with the idea of using an imbedded IFF List inside of the WAVE file. NOOOOOOOOOOOOOOOOOOO!!!!!!!!! And this "Wave List" would contain multiple 'data' and 'slnt' chunks. NOOOOOOOOOOOOOOOOOO!!!! The Type ID for this List is 'wavl'.

I strongly suggest that you refuse to support any WAVE file that exhibits this Wave List nonsense. There's no need for it, and hopefully, the misguided programmer who conjured it up will be embarrassed into hanging his head in shame when nobody agrees to support his foolishness. Just say "NOOOOOOOOOOOOOOO!!!!"

Cue chunk

The Cue chunk contains one or more "cue points" or "markers". Each cue point references a specific offset within the waveformData array, and has its own CuePoint structure within this chunk.

In conjunction with the Playlist chunk, the Cue chunk can be used to store looping information.

CuePoint Structure

```
typedef struct {
  long     dwIdentifier;
  long     dwPosition;
  ID       fccChunk;
  long     dwChunkStart;
  long     dwBlockStart;
  long     dwSampleOffset;
} CuePoint;
```

The dwIdentifier field contains a unique number (ie, different than the ID number of any other CuePoint structure). This is used to associate a CuePoint structure with other structures used in other chunks which will be described later.

The dwPosition field specifies the position of the cue point within the "play order" (as determined by the Playlist chunk. See that chunk for a discussion of the play order).

The fccChunk field specifies the chunk ID of the Data or Wave List chunk which actually contains the waveform data to which this CuePoint refers. If there is only one Data chunk in the file, then this field is set to the ID 'data'. On the other hand, if the file contains a Wave List (which can contain both 'data' and 'slnt' chunks), then fccChunk will specify 'data' or 'slnt' depending upon in which type of chunk the referenced waveform data is found.

The dwChunkStart and dwBlockStart fields are set to 0 for an uncompressed WAVE file that contains one 'data' chunk. These fields are used only for WAVE files that contain a Wave List (with multiple 'data' and 'slnt' chunks), or for a compressed file containing a 'data' chunk. (Actually, in the latter case, dwChunkStart is also set to 0, and only dwBlockStart is used). Again, I want to emphasize that you can avoid all of this unnecessary crap if you avoid hassling with compressed files, or Wave Lists, and instead stick to the sensible basics.

The dwChunkStart field specifies the byte offset of the start of the 'data' or 'slnt' chunk which actually contains the waveform data to which this CuePoint refers. This offset is relative to the start of the first chunk within the Wave List. (ie, It's the byte offset, within the Wave List, of where the 'data' or 'slnt' chunk of interest appears. The first chunk within the List would be at an offset of 0).

The dwBlockStart field specifies the byte offset of the start of the block containing the position. This offset is relative to the start of the waveform data within the 'data' or 'slnt' chunk.

The dwSampleOffset field specifies the sample offset of the cue point relative to the start of the block. In an uncompressed file, this equates to simply being the offset within the waveformData array. Unfortunately, the WAVE documentation is much too ambiguous, and doesn't define what it means by the term "sample offset". This could mean a byte offset, or it could mean counting the sample points (for example, in a 16-bit wave, every 2 bytes would be 1 sample point), or it could even mean sample frames (as the loop offsets in AIFF are specified). Who knows? The guy who conjured up the Cue chunk certainly isn't saying. I'm assuming that it's a byte offset, like the above 2 fields.

Cue Chunk

```
#define CueID 'cue '   /* chunk ID for Cue Chunk */
```

```
typedef struct {
  ID        chunkID;
  long      chunkSize;

  long      dwCuePoints;
  CuePoint  points[];
} CueChunk;
```

The ID is always **cue** . chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields.

The dwCuePoints field is the number of CuePoint structures in the Cue Chunk. If dwCuePoints is not 0, it is followed by that many CuePoint structures, one after the other. Because all fields in a CuePoint structure are an even number of bytes, the length of any CuePoint will always be even. Thus, CuePoints are packed together with no unused bytes between them. The CuePoints need not be placed in any particular order.

The Cue chunk is optional. No more than one Cue chunk can appear in a WAVE.

Playlist chunk

The Playlist (plst) chunk specifies a play order for a series of cue points. The Cue chunk contains all of the cue points, but the Playlist chunk determines how those cue points are used when playing back the waveform (ie, which cue points represent looped sections, and in what order those loops are "played"). The Playlist chunk contains one or more Segment structures, each of which identifies a looped section of the waveform (in conjunction with the CuePoint structure with which it is associated).

Segment Structure

```
typedef struct {
  long      dwIdentifier;
  long      dwLength;
  long      dwRepeats;
} Segment;
```

The dwIdentifier field contains a unique number (ie, different than the ID number of any other Segment structure). This field should correspond with the dwIndentifier field of some CuePoint stored in the Cue chunk. In other words, this Segment structure contains the looping information associated with that CuePoint structure with the same ID number.

The dwLength field specifies the length of the section in samples (ie, the length of the looped section). Note that the start position of the loop would be the dwSampleOffset of the referenced CuePoint structure in the Cue chunk. (Or, you may need to hassle with the dwChunkStart and dwBlockStart fields as well if dealing with a Wave List or compressed data).

The dwRepeats field specifies the number of times to play the loop. I assume that a value of 1 means to repeat this loop once only, but the WAVE documentation is very incomplete and omits this important information. I have no idea how you would specify an infinitely repeating loop. Certainly, the person who conjured up the Playlist chunk appears to have no idea whatsoever. Due to the ambiguities, inconsistencies, inefficiencies, and omissions of the Cue and Playlist chunks, I very much recommend that you use the Sampler chunk (described later) to replace them.

Playlist chunk

```
#define PlaylistID 'plst'  /* chunk ID for Playlist Chunk */

typedef struct {
  ID        chunkID;
```

```
long      chunkSize;

long      dwSegments;
Segment   Segments[];
} PlaylistChunk;
```

The ID is always **plst**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields.

The dwSegments field is the number of Segment structures in the Playlist Chunk. If dwSegments is not 0, it is followed by that many Segment structures, one after the other. Because all fields in a Segment structure are an even number of bytes, the length of any Segment will always be even. Thus, Segments are packed together with no unused bytes between them. The Segments need not be placed in any particular order.

Associated Data List

The Associated Data List contains text "labels" or "names" that are associated with the CuePoint structures in the Cue chunk. In other words, this list contains the text labels for those CuePoints.

Again, we're talking about another imbedded IFF List within the WAVE file. NOOOOOOOOOOOOOO!!!! What's a List? A List is simply a "master chunk" that contains several "sub-chunks". Just like with any other chunk, the "master chunk" has an ID and chunkSize, but inside of this chunk are sub-chunks, each with its own ID and chunkSize. Of course, the chunkSize for the master chunk (ie, List) includes the size of all of these sub-chunks (including their ID and chunkSize fields).

The "Type ID" for the Associated Data List is "adtl". Remember that an IFF list header has 3 fields:

```
typedef struct {
  ID      listID;      /* 'list' */
  long    chunkSize;   /* includes the Type ID below */
  ID      typeID;      /* 'adtl' */
} ListHeader;
```

There are several sub-chunks that may be found inside of the Associated Data List. The ones that are important to WAVE format have IDs of "labl", "note", or "ltxt". Ignore the rest. Here are those 3 sub-chunks and their fields:

The Associated Data List is optional. The WAVE documentation doesn't specify if more than one can be contained in a WAVE file.

Label Chunk

```
#define LabelID 'labl'  /* chunk ID for Label Chunk */

typedef struct {
  ID      chunkID;
  long    chunkSize;

  long    dwIdentifier;
  char    dwText[];
} LabelChunk;
```

The ID is always **labl**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields nor any possible pad byte needed to make the chunk an even size (ie, chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing pad byte).

The dwIdentifier field contains a unique number (ie, different than the ID number of any other Label chunk). This field should correspond with the dwIndentifier field of some CuePoint stored in the Cue chunk. In other words, this Label chunk contains the text label associated with that CuePoint structure with the same ID number.

The dwText array contains the text label. It should be a null-terminated string. (The null byte is included in the chunkSize, therefore the length of the string, including the null byte, is chunkSize - 4).

Note Chunk

```
#define NoteID 'note'  /* chunk ID for Note Chunk */

typedef struct {
  ID      chunkID;
  long    chunkSize;

  long    dwIdentifier;
  char    dwText[];
} NoteChunk;
```

The Note chunk, whose ID is **note**, is otherwise exactly the same as the Label chunk (ie, same fields). See what I mean about pointless duplication? But, in theory, a Note chunk contains a "comment" about a CuePoint, whereas the Label chunk is supposed to contain the actual CuePoint label. So, it's possible that you'll find both a Note and Label for a specific CuePoint, each containing different text.

Labeled Text Chunk

```
#define LabelTextID 'ltxt'  /* chunk ID for Labeled Text Chunk */

typedef struct {
  ID      chunkID;
  long    chunkSize;

  long    dwIdentifier;
  long    dwSampleLength;
  long    dwPurpose;
  short   wCountry;
  short   wLanguage;
  short   wDialect;
  short   wCodePage;
  char    dwText[];
} LabelTextChunk;
```

The ID is always **ltxt**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields nor any possible pad byte needed to make the chunk an even size (ie, chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing pad byte).

The dwIdentifier field is the same as the Label chunk.

The dwSampleLength field specifies the number of sample points in the segment of waveform data. In other words, a Labeled Text chunk contains a label for a **section** of the waveform data, not just a specific point, for example the looped section of a waveform.

The dwPurpose field specifies the type or purpose of the text. For example, dwPurpose can contain an ID like "scrp" for script text or "capt" for close-caption text. How is this related to waveform data? Well, it isn't really. It's just that Associated Data Lists are used in other file formats, so they contain generic fields that sometimes don't have much relevance to waveform data.

The wCountry, wLanguage, and wCodePage fields specify the country code, language/dialect, and code page for the text. An application typically queries these values from the operating system.

Sampler Chunk

The Sampler (smpl) Chunk defines basic parameters that an instrument, such as a MIDI sampler, could use to play the waveform data. Most importantly, it includes information about looping the waveform (ie, during playback, to "sustain" the waveform). Of course, as you've come to expect from the WAVE file format, it duplicates some of the information that can be found in the Cue and Playlist chunks, but fortunately, in a more sensible, consistent, better-documented way.

```
#define SamplerID 'smpl'  /* chunk ID for Sampler Chunk */

typedef struct {
  ID             chunkID;
  long           chunkSize;

  long            dwManufacturer;
  long            dwProduct;
  long            dwSamplePeriod;
  long            dwMIDIUnityNote;
  long            dwMIDIPitchFraction;
  long            dwSMPTEFormat;
  long            dwSMPTEOffset;
  long            cSampleLoops;
  long            cbSamplerData;
  struct SampleLoop Loops[];
} SamplerChunk;
```

The ID is always **smpl**. chunkSize is the number of bytes in the chunk, not counting the 8 bytes used by ID and Size fields nor any possible pad byte needed to make the chunk an even size (ie, chunkSize is the number of remaining bytes in the chunk after the chunkSize field, not counting any trailing pad byte).

The dwManufacturer field contains the MMA Manufacturer code for the intended sampler. Each manufacturer of MIDI products has his own ID assigned to him by the MIDI Manufacturer's Association. See the MIDI Specification (under System Exclusive) for a listing of current Manufacturer IDs. The high byte of dwManufacturer indicates the number of low order bytes (1 or 3) that are valid for the manufacturer code. For example, this value will be 0x01000013 for Digidesign (the MMA Manufacturer code is one byte, 0x13); whereas 0x03000041 identifies Microsoft (the MMA Manufacturer code is three bytes, 0x00 0x00 0x41). If the WAVE is not intended for a specific manufacturer, then this field should be set to 0.

The dwProduct field contains the Product code (ie, model ID) of the intended sampler for the dwManufacturer. Contact the manufacturer of the sampler to ascertain the sampler's model ID. If the WAVE is not intended for a specific manufacturer's product, then this field should be set to 0.

The dwSamplePeriod field specifies the period of one sample in nanoseconds (normally 1/nSamplesPerSec from the Format chunk. But note that this field allows finer tuning than nSamplesPerSec). For example, 44.1 KHz would be specified as 22675 (0x00005893).

The dwMIDIUnityNote field is the MIDI note number at which the instrument plays back the waveform data without pitch modification (ie, at the same sample rate that was used when the waveform was created). This value ranges 0 through 127, inclusive. Middle C is 60.

The dwMIDIPitchFraction field specifies the fraction of a semitone up from the specified dwMIDIUnityNote. A value of 0x80000000 is 1/2 semitone (50 cents); a value of 0x00000000 represents no fine tuning between semitones.

The dwSMPTEFormat field specifies the SMPTE time format used in the dwSMPTEOffset field. Possible values are:

```
0  = no SMPTE offset (dwSMPTEOffset should also be 0)
24 = 24 frames per second
25 = 25 frames per second
29 = 30 frames per second with frame dropping ('30 drop')
30 = 30 frames per second
```

The dwSMPTEOffset field specifies a time offset for the sample if it is to be syncronized or calibrated according to a start time other than 0. The format of this value is 0xhhmmssff. hh is a signed Hours value [-23..23]. mm is an unsigned Minutes value [0..59]. ss is unsigned Seconds value [0..59]. ff is an unsigned value [0..(- 1)].

The cSampleLoops field is the number (count) of SampleLoop structures that are appended to this chunk. These structures immediately follow the cbSamplerData field. This field will be 0 if there are no SampleLoop structures.

The cbSamplerData field specifies the size (in bytes) of any optional fields that an application wishes to append to this chunk. An application which needed to save additional information (ie, beyond the above fields) may append additional fields to the end of this chunk, after all of the SampleLoop structures. These additional fields are also reflected in the ChunkSize, and remember that the chunk should be padded out to an even number of bytes. The cbSamplerData field will be 0 if no additional information is appended to the chunk.

What follows the above fields are any SampleLoop structures. Each SampleLoop structure defines one loop (ie, the start and end points of the loop, and how many times it plays). What follows any SampleLoop structures are any additional, proprietary sampler information that an application chooses to store.

SampleLoop Structure

```
typedef struct {
  long  dwIdentifier;
  long  dwType;
  long  dwStart;
  long  dwEnd;
  long  dwFraction;
  long  dwPlayCount;
} SampleLoop;
```

The dwIdentifier field contains a unique number (ie, different than the ID number of any other SampleLoop structure). This field may correspond with the dwIdentifier field of some CuePoint stored in the Cue chunk. In other words, the CuePoint structure which has the same ID number would be considered to be describing the same loop as this SampleLoop structure. Furthermore, this field corresponds to the dwIndentifier field of any label stored in the Associated Data List. In other words, the text string (within some chunk in the Associated Data List) which has the same ID number would be considered to be this loop's "name" or "label".

The dwType field is the loop type (ie, how the loop plays back) as so:

0	Loop forward (normal)
1	Alternating loop (forward/backward)
2	Loop backward
3 - 31	reserved for future standard types
32 - ?	sampler specific types (manufacturer defined)

The dwStart field specifies the startpoint of the loop. In other words, it's the byte offset from the start of waveformData[], where an offset of 0 would be at the start of the waveformData[] array (ie, the loop start is at the very first sample point).

The dwEnd field specifies the endpoint of the loop (ie, a byte offset).

The dwFraction field allows fine-tuning for loop fractional areas between samples. Values range from 0x00000000 to 0xFFFFFFFF. A value of 0x80000000 represents 1/2 of a sample length.

The dwPlayCount field is the number of times to play the loop. A value of 0 specifies an infinite sustain loop (ie, the wave keeps looping until some external force interrupts playback, such as the musician releasing the key that triggered that wave's playback).

The Sampler Chunk is optional. I don't know as if there is any limit of one per WAVE file. I don't see why there should be such a limit, since after all, an application may need to deal with several MIDI samplers.

Instrument chunk

The Instrument Chunk contains some of the same type of information as the Sampler chunk. So what else is new?

```
#define InstrumentID 'inst'  /* chunkID for Instruments Chunk */

typedef struct {
  ID      chunkID;
  long    chunkSize;

  unsigned char UnshiftedNote;
  char          FineTune;
  char          Gain;
  unsigned char LowNote;
  unsigned char HighNote;
  unsigned char LowVelocity;
  unsigned char HighVelocity;
} InstrumentChunk;
```

The ID is always **inst**. chunkSize should always be 7 since there are no fields of variable length.

The UnshiftedNote field is the same as the Sampler chunk's dwMIDIUnityNote field.

The FineTune field determines how much the instrument should alter the pitch of the sound when it is played back. Units are in cents (1/100 of a semitone) and range from -50 to +50. Negative numbers mean that the pitch of the sound should be lowered, while positive numbers mean that it should be raised. While not the same measurement is used, this field serves the same purpose as the Sampler chunk's dwFraction field.

The Gain field is the amount by which to change the gain of the sound when it is played. Units are decibels. For example, 0db means no change, 6db means double the value of each sample point (ie, every additional 6db doubles the gain), while -6db means halve the value of each sample point.

The LowNote and HighNote fields specify the suggested MIDI note range on a keyboard for playback of the waveform data. The waveform data should be played if the instrument is requested to play a note between the low and high note numbers, inclusive. The UnshiftedNote does not have to be within this range.

The LowVelocity and HighVelocity fields specify the suggested range of MIDI velocities for playback of the waveform data. The waveform data should be played if the note-on velocity is between low and high velocity, inclusive. The range is 1 (lowest velocity) through 127 (highest velocity), inclusive.

The Instrument Chunk is optional. No more than 1 Instrument Chunk can appear in one WAVE.

I have often been asked if there is any software to do digital effects upon MIDI output (ie, distortion, chorus, delay, etc). Some people have the idea that they would like to use software that implements the same effects upon their sequencer's MIDI tracks as is offered upon a hardware effects unit.

Remember that MIDI is not audio data. MIDI is control data. That is, the MIDI stream is electrical digital signals that convey "hardware instructions", not "audio output". (For more details about MIDI, read the article What's MIDI?). So, you can't regard MIDI as an audio signal, and treat it as so.

If you were thinking about some sort of software that somehow intercepts the MIDI data that a sequencer program is sending out of your computer's MIDI interface to external MIDI gear, and "adds effects" such as "distortion" to that MIDI stream, then you definitely have a misunderstanding of what MIDI is and what it does, and/or how certain effects are implemented.

First of all, you definitely don't want to alter the MIDI signal itself because not only aren't those "hardware instructions" audio signals, altering them will achieve absolutely no purpose except to make it impossible for your gear to communicate over MIDI. It would mutate that MIDI "language" into a "foreign language" that your external MIDI gear would no longer understand. So, for example, you can't plug a MIDI cable into a distortion box or delay device and expect to be "adding effects" as if you were doing so to a musical, audio signal. You wouldn't be.

You can add more instructions to the MIDI stream that may result in causing a sound module to simulate the effects that an effects box may have. But, the sound module itself must be capable of simulating those effects. And that's the gist of it right there. You're not adding the actual effects to the MIDI stream. Rather, you're adding more MIDI instructions that tell your sound module to do something with its hardware which simulates some effect like distortion, delay, reverb, etc. Hopefully, the sound module's hardware can be manipulated to simulate the effect. Of course, by nature, this all must be done in real-time, because the effect can't be achieved until the sound module actually is "sounding", which of course it does in real-time.

Let's consider the easiest effect to simulate via MIDI messages; delay. A delay is simply a discrete repeat of a sound. A digital delay will usually just "buffer" the sound coming into it, and then "play it back" a few fractions of a second later. The human ear hears this as a delayed repeat. One way to make a sound module produce a delay effect is to have all of the MIDI messages that are sent to it resent after a few fractions of a second have passed. You can easily do that with a sequencer track as so:

1. Record a musical track and copy it to a second track (ie, duplicate all of the same MIDI data onto two tracks).
2. Move the second track's MIDI events forward in time (ie, add a small amount of time, usually measured in "clock pulses", to each MIDI event on the second track). Most sequencers have a feature whereby you can specify an amount of time to be added to all events in a track.
3. Play both tracks on one MIDI module. (If your module automatically cuts off overlapping

note-ons, you'll need to reassign the second track to a different MIDI channel than the first track. In this case, you'll need a multitimbral module to hear both tracks).

The effect will be a delay as the module plays both tracks in real-time; the original track as well as the "delayed" track. You can make the delay longer by adding more time to the second track's events (or make it shorter by subtracting time). You can adjust the volume of the delay by adjusting all of the note velocities in the second track. Most sequencers have a feature whereby you can specify an amount to be added or subtracted from all note events' velocities.

You can even stick MIDI Pan Controller messages, in various places in your second track, to come up with a "panning echo". But you'll probably need to add lots of Pan events, by hand, unless you use CakeWalk's CAL language to automate the procedure, or you've got some programmable slider on your module which can generate Pan Controller messages which you record with the sequencer.

An important thing to consider is that your sound module is using some of its voices to play the second track too. So this effect cuts down on the polyphony you have left for your other musical parts.

Nevertheless, a delay effect is achievable on virtually any MIDI module via MIDI (ie, most all MIDI modules play polyphonically, and either allow overlapping note-ons or are multitimbral. So, the above technique will work with most modules).

There are even software programs that can wedge themselves inbetween a MIDI sequencer and a MIDI interface's driver, and duplicate the MIDI stream with a delay. So, you could get the effect without needing to duplicate a sequencer's track. There are also programs that read the MIDI IN of an interface and duplicate a delayed stream to the MIDI OUT, so you could use the computer to add a delay to a part that you're playing on the module. But ultimately, your module is using more of its voices to play those delayed notes.

Let's consider reverb. Reverb is nothing more than many, many, many, many delays (ie, repeats) of a sound, one right after the other. The time inbetween each delay is so short that our ear hears this long, long, long series of delays as a continuous sound (rather than a whole bunch of delays) that we refer to as "reverb". I deliberately used the phrases "many, many, many" and "long, long, long" because I need to point out that we're typically talking about thousands of repeats in order to get a typical reverb sound. You'd run out sequencer tracks to copy delayed MIDI events, as well as voices on your MIDI gear, before you'd ever get enough delays to achieve even a decent reverb effect. So, forget about it. You'll need dedicated hardware (ie, some sort of DSP setup for a reverb effect) for this effect.

Let's consider distortion. Distortion involves clipping of audio signals. You have to feed a Distortion box (or algorithm) audio data. In the case of a Distortion box, it typically expects an analog audio

signal (ie, such as what comes out of the AUDIO OUT jack on your MIDI gear). That's definitely not a MIDI stream. A software algorithm typically operates upon digital audio data (such as what is contained in a WAVE file, or what comes from a sound card's Analog to Digital Converter). That's definitely also not a MIDI stream. The only way to produce distortion via MIDI is maybe to boost the note velocities upon all note-on events, turn up the volume of the Patch, and hope that your module's audio output clips. Most MIDI gear is designed so that even if you "turn everything up to maximum", the output won't clip. There's enough "headroom" built in. So, it's doubtful that this technique will work. (Besides, if you use volume to overdrive the output stage, then you'll need an external volume control for the overall volume of the part). Again, distortion is something that you'll need dedicated hardware to achieve.

To simulate a chorus effect, your MIDI module would have to allow lots of control over the module's Voltage Controlled Filter (ie, VCF) via MIDI messages. Very few modules offer that. Again, it'll probably be easier and more effective to use dedicated hardware for such.

A leslie effect involves continuously panning a sound (which you can do on virtually every module via MIDI Pan Controller messages) as well as varying filter cutoff (which again, is probably not implemented via MIDI).

In conclusion, about the only effect that is "do-able" via MIDI (without any dedicated effects hardware beyond what is in nearly every module manufactured today) is delay.

It should be pointed out that many hardware effects devices nowadays allow their parameters to be controlled via MIDI messages, even in real-time. So, you could use computer software to control the effects hardware, but you can't just replace the hardware with a software program.

The nice thing about computer sound cards that do have some effects hardware built-in (ie, maybe a generic DSP chip, or dedicated delay/chous/reverb/etc units) is that the cards are typically designed so that computer software can directly access, control, and maybe even somehow transform the card's hardware effects (ie, maybe by loading new algorithms into the DSP).

As a side note, it's possible to use computer software to modify digital audio data, and then send this modified data to a Digital to Analog Converter (ie, DAC) for playback. But of course, digital audio is something entirely different than MIDI. Software that deals with digital audio waveforms often offers the ability to perform effects algorithms such as delay, reverb, distortion, flanging, etc. After all, such software is directly manipulating the audio data, not MIDI control information, and so algorithms can be applied directly to that audio data. With MIDI, you're always controlling some sound source which is producing the audio data in real-time. If the sound source has no way of doing what you're asking it to do (via MIDI instructions), then you can't manipulate the audio data as desired.

So, it may be advantageous to wait to apply your effects until after you record the MIDI playback into some digital audio program such as Cool Edit. Play/record each MIDI track separately, as a distinct audio track in Cool Edit. And then apply the effect to that audio track.

Active Sense is a type of MIDI message (ie, that travels over your MIDI cables just like note data or controller data). It's used to implement a "safety feature". It can provide an automatic "MIDI panic" control that will cause stuck notes and other undesirable effects to be resolved whenever the MIDI connection inbetween modules is broken or impeded in some way.

Here's how Active Sense works.

Assume that you have two sound modules daisy-chained to the MIDI OUT of a keyboard controller. Let's also say that all three units implement Active Sense. If you aren't playing the controller at any given moment (and therefore the controller isn't sending out MIDI data), then after a little while of sitting there idle, the controller will automatically send out an Active Sense MIDI message that tells the attached sound modules (connected to the controller's MIDI OUT) "Hey, I'm still connected to you. Don't worry that you haven't seen any MIDI note data, or controller messages, or Pitch Wheel messages, etc, from me in awhile. I'm just sitting here idle because my owner isn't playing me at the moment". When the sound modules hear this "reassurance" from the controller, then they expect the controller to repeat this same Active Sense message every so often as long as it's sitting idle. That's how the sound modules know that everything is still OK with their MIDI connections. If not for that steady stream of Active Sense messages, then there would be no other MIDI data sent to the sound modules while the controller was left idle, and the sound modules would have no way of knowing whether they were still connected to the controller.

Now, if you walk up to the controller and start playing it, then it will stop sending these "reassurance" messages because now it will be sending other MIDI data instead. And then later, if you again leave the controller sitting idle, it will resume sending Active Sense messages out of its MIDI OUT jack.

By constantly "talking" to the sound modules (sending Active Sense messages whenever it has no other MIDI data to send), the controller is able to let the modules know that they're still connected to the controller.

So how is a "safety feature" derived from this?

Let's say that you press and hold down a key on the controller. This sends a MIDI Note On message to the sound modules telling them to start sounding a note. OK, while still holding that note down, you disconnect the MIDI cable at the controller's MIDI OUT (ie, disconnect the daisy-chained modules). Now, the modules are left sounding that "stuck note". Even if you now release the key on the controller, there's no way for the controller to tell the modules to stop sounding the note (because you disabled the MIDI connection). OK, so the modules are holding that stuck note, and they're waiting for some more MIDI messages from the controller. After awhile, they start to get worried, because they know that if the controller isn't sending them MIDI note data, or controllers, or Pitch Wheel messages, etc, then it should at least be sending them Active Sense messages. And yet, they're not getting even that reassurance. So, what happens? Well, after each module waits for a suitably long time (ie, 300 milliseconds, which is a long time for MIDI devices, but a fraction of a second to humans) without receiving an Active Sense message, then the module automatically concludes that the MIDI connection is broken, and it automatically turns off any sounding notes, all by itself. Hence, the stuck note is turned off.

Active Sense allows <u>any</u> module to detect when the MIDI connection to the preceding module is broken. For example, let's say that you disconnected the MIDI cable between the two sound modules. (ie, The first module is still attached to the controller's MIDI OUT, but now the second module is no longer daisy-chained to the first module's MIDI OUT). In this case, the first module will still be getting the Active Sense messages from the controller, but these can't be passed on to the second module (because it's disconnected from the MIDI chain). Therefore, the second module will quickly realize that its MIDI connection has been broken.

Active Sense can also be useful to MIDI software, particularly sequencers. For example, let's say we have some MIDI player software. Whenever this software is not playing a sequence, it sends out Active Sense messages. Now let's say the software "locks up" or crashes in the middle of playing back a MIDI file, leaving some "stuck notes" still playing on some external module. What happens? Well, obviously, the software will be unable to send any more MIDI messages to the external module. After awhile, the module will think *"Hmmmm, I know that someone has been sending me Active Sense messages whenever there aren't any other MIDI messages. But I've been waiting for 300 milliseconds now, and I've gotten no MIDI messages at all -- not even Active Sense. Something is wrong. I had better kill any playing notes".* The net result is that, within a fraction of a second after the software player locks up, the module automatically cuts off any stuck notes (and stops doing anything else it was doing). As you can see, Active Sense may be a very useful safety feature for live performance.

So, Active Sense is a "safety" feature. It allows a MIDI module to know when its connection to some other unit's MIDI OUT jack has been broken. In such a case, the module can take automatic safety precautions such as turning off any sounding notes, or resetting itself to a default state, in order to avoid being left in a "stuck" or undesirable state.

Note that many (in fact, most) MIDI units do <u>not</u> implement Active Sense, and therefore aren't setup for this "safety" feature. (Typically, Roland gear <u>does</u> implement Active Sense. Roland is one of the few companies that adhere to the MIDI spec to this degree). If you connect a controller with Active Sense to a module that doesn't implement Active Sense, then the controller's reassurance messages are ignored, and the module won't be able to realize when a connection has been broken.

So how do you know if your unit implements Active Sense? Well, look at the **MIDI Implementation Chart** in your users manual. For sound modules, you'll look down the Recognized column until you get to Active Sense (it's listed in the **AUX Messages** section almost at the bottom of the chart). Make sure that there is a circle instead of an X shown beside Active Sense. For a controller, you'll look down the Transmitted column.

Roland System Exclusive Implementation

Introduction

About the time that Roland released the S-10 sampler (mid 1980's as I recall), the company had devised a method of transfering any and all parameters (ie, settings) of a MIDI device by using System Exclusive messages over MIDI. So, any setting that you could adjust from the front panel of the device, you could also adjust over MIDI with a System Exclusive message.

Note: From here on, I will refer to a System Exclusive message as "SysEx".

It's true that many other manufacturers also allowed you to "dump" settings over MIDI. But the real difference was how Roland implemented this feature. What Roland did was devise a way that you could adjust one, and only one, specific setting (without affecting any other settings) using a single, small SysEx message. Other manufacturers typically did not do this. They required you to change many settings simultaneously using one, relatively large SysEx message. You couldn't easily adjust a single parameter without affecting the others, and certainly not using a small SysEx message.

What were the ramifications of this?

Well, let's say that you have a patch editor program on your computer, and it has graphical buttons, sliders, etc, that you manipulate with the mouse in order to change settings over MIDI. Let's say that you click on a button to change one particular setting. With Roland's method, the patch software can now send a very small, SysEx message to the module to change only that one setting. This takes merely a fraction of a second (because it's a very small SysEx message), so when you click the button, it seems as if the change is made instantaneously on the module. You can immediately play the module and hear how the new setting sounds.

Now let's contrast that to the scheme typically used by other manufacturers. First of all, in order to change that one setting over MIDI, the patch software has to gather up all of the other settings that need to be changed simultaneously, and create one big SysEx message. Such a message could be several thousand bytes long (compared to less than a dozen bytes for the above Roland SysEx message). This takes a relatively long time to send over MIDI. You could be waiting from 15 seconds to several minutes before you're finally able to hear how the new setting sounds on the module. As you can see, this was not conducive to remotely editing those settings in realtime -- not like with a Roland module, which seems to interact almost instantaneously with the software.

There is another ramification. Because a Roland SysEx message to change one setting is so small, you could even embed that SysEx in a MIDI file. For example, let's say that you want that setting changed in the middle of a song. You can play that MIDI file without hearing any glitch as a result of sending that small SysEx message in the middle of the song.

But with the other manufacturer, if you tried to put such a large SysEx message in the middle of a song, the note data would be delayed for several seconds while the SysEx was sent, and typically you'd hear a noticeable "glitch" in the rhythmic playback.

So Roland's real accomplishment was being the first manufacturer to use a way to transfer single settings over MIDI using small, SysEx messages. (But note that Roland's method is flexible -- it also allows transfering many settings simultaneously using larger SysEx messages). This allowed patch and sequencer software to change settings in realtime.

This proved so useful that Roland decided to provide this same feature with nearly every one of their products released after the S-10. And to make matters even easier, Roland standardized much of this method so that, if you know how to change individual settings on an S-10, then you pretty much know how to change individual settings on every other Roland product made since then.

The good news is that many manufacturers learned from Roland's example, and now have similiar schemes.

Roland address map

In order to be able to transfer individual settings, Roland came up with the idea of an *address map*. All this means is that each individual setting has its own unique "location" (ie, address) where it is stored (inside of a Roland module). Just like you have a different mailing address than your neighbor does, so too each individual Roland setting has its own address. A Roland address is simply a number. So for example, while your mailing address may be "666 Some street, Anytown, USA", the address of a Roland module's *master volume* could be address number 14. No other setting in the module will be at that address.

So, to query the current value of some Roland setting, you must tell a Roland module what address you would like to query. (You send a SysEx message to do that). The Roland module will then send back the setting (ie, its current value) at that address. (The Roland module will send back a SysEx message containing the current value). So for example, to find out what the master volume is currently set to, maybe you'll send a SysEx asking for the value at address 14. And the module sends back a SysEx containing the current value of master volume.

To change an individual setting, you must tell a Roland module at which address you wish to make that change, and then tell it what you want the new value to be. (You send a SysEx message to do that). The Roland module will then store the new value at that address. So for example, to set the master volume to a value of 100, maybe you send a SysEx telling the module to store the value 100 at address 14. And the module immediately changes its master volume to that value.

Now, not all Roland modules will have all the same settings. And if they do have <u>some</u> of the same settings, those settings may be at different addresses in different modules. But all Roland modules follow the above concept of having each setting at its own unique address, and you set or query a setting as described above.

So, each Roland module may have a unique address map, and the Roland user manual is where you find the details of the address map for a given module. But before we delve into a particular module's address map, there are more things that all Roland modules have in common, so let's discuss the latter first.

Roland SysEx message

As mentioned, you query or change a setting using a SysEx message. Fortunately, Roland has maintained a bit of standardization here, so that, if you understand the SysEx messages for one module, then you've got the general idea for all of them.

To change a setting, you must tell a Roland module at which address you wish to make that change, and then tell it what you want the new value to be. You do that by sending a "Data Set 1" SysEx message, which contains not only the new value, but also the address.

Note: <u>Some</u> Roland modules support "handshaking", whereby whenever you send the module a SysEx message, it will subsequently send back a special "Ack" SysEx message to acknowledge this. For such handshaking purposes, Roland also has a "Data Set 2" SysEx message which is used in lieu of the "Data Set 1". The "Data Set 2" is used only when you want handshaking. Otherwise, the "Data Set 1" is used. This tutorial will not discuss handshaking, and instead, we'll deal only with non-handshaking SysEx transfers using "Data Set 1".

To query a setting, you first must tell a Roland module what address you would like to query. You do that by sending a "Data Request 1" SysEx message, which contains the address of the setting you wish to query. The module will return a "Data Set 1" containing the current value of that setting.

Note that MIDI deals only with 7-bit bytes of data. In other words, each data byte within a SysEx message must be from 0 to 127. (ie, There is a range limit of 128 possible "steps"). But also note that some Roland settings could be beyond this range. For example, maybe master volume can be set from 0 to 16,000. What is the implication of this? Well, if the setting's value

can be beyond 128 different steps, then the value will be broken apart into more than one data byte within the "Data Set 1" message. We'll discuss this in more detail later. (Fortunately, most of the settings in a typical address map are limited to a range of 128 or less steps, and therefore can be easily expressed as just one data byte within a "Data Set 1" message).

A Roland address is 4 (or sometimes 3, for earlier models) bytes in a "Data Set 1" or "Data Request 1" message. We'll talk more about those bytes later when we examine an address map in detail.

In the typical Roland user's manual, there is a chapter entitled **MIDI Implementation**. This chapter lists the details of how the unit responds to MIDI messages sent to it, as well as what MIDI messages the unit is capable of generating (if any) and sending to some other MIDI device (such as a sequencer). It also contains the information you need to be able to transfer settings over MIDI. If you peruse this section, you'll eventually get to a sub-section named **System Exclusive**. Within this sub-section, you'll eventually get to a sub-section **Data Transmission**, and then finally **Parameter Address Map**. These last two sub-sections tell you what you need to know to transfer settings over MIDI using SysEx.

You will find the "Data Set 1" and "Data Request 1" SysEx messages under **Data Transmission**. Let's take a look at the "Data Request 1" message for the Roland JV-1010. This is the message that you send to the JV-1010, to ask it to return a "Data Set 1" message containing the setting of interest. It has 15 bytes. Here is the JV-1010 "Data Request 1" message (and note that bytes are expressed in hexadecimal unless otherwise noted):

F0	All SysEx messages begin with an F0.
41	All Roland SysEx messages have a 41 here. Other manufacturers will have a different number here. This conclusively confirms that this is a Roland SysEx message.
DEV_ID	This is the device ID. Think of it like a MIDI channel, but for SysEx messages only. A JV-1010 can be set to listen for SysEx messages that have only a certain value here. For example, if the number you put here is 10, then only those JV-1010's set to respond to a device ID of 10 will pay attention to this SysEx message. In this way, you can have more than one JV-1010 daisy-chained, set them to different device IDs, and then be able to query their settings individually. A Roland unit defaults to using device ID 10, so normally, you'll find a 10 here.
6A	This is the model ID. A SysEx specifically for the JV-1010 will have a 6A here. Other Roland models will have a different number here. This conclusively confirms that this is a Roland SysEx message for a JV-1010.
11	This is the command ID. A "Data Request 1" SysEx will have an 11 here. Other SysEx messages, such as a "Data Set 1" will have a different number here.
ADDR_1	This is the first byte (of the 3 or 4 bytes) that make up the address of the setting whose value you wish to query. What number is here depends upon which setting's value you wish returned in a "Data Set 1". (Remember that each setting has its own unique address). This first byte is often referred to as the Most Significant Byte (MSB).
ADDR_2	This is the second byte of the address.
ADDR_3	This is the third byte of the address.
ADDR_4	This is the fourth and last byte of the address. Note: For some Roland models, there are only three address bytes, and so this one is omitted.
SIZE_1	When you send a "Data Request 1" to the JV-1010, you need to tell it how many data bytes you expect to be returned in the subsequent "Data Set 1". SIZE_1 is the first byte of this count. If you're querying an individual setting that is within the range of 0 to 128, there will be only one data byte returned, and so the SIZE bytes (in this "Data Request 1") will add up to 1. For a setting whose value can be larger than this range, it will broken up into several bytes, and therefore the SIZE bytes will be 2 or greater. In other words, SIZE_1, SIZE_2, SIZE_3, and SIZE_4 taken together represent a count of how many data bytes you expect returned in a subsequent "Data Set 1". We'll later discuss this in more detail, and you'll see the useful purpose of SIZE_1, SIZE_2, SIZE_3, and SIZE_4.
SIZE_2	This is the second byte of the count.
SIZE_3	This is the third byte of the count.
SIZE_4	This is the fourth and last byte of the count. Note: For some Roland models, there are only three SIZE bytes, and so this one is omitted.

CHECKSUM	This is the checksum. What value you put here depends upon what numbers you use for the address and size bytes above. After all, the checksum is essentially just the sum of all of those preceding bytes, and it is used for error-detection (in case a SysEx message somehow gets "garbled").
F7	All SysEx messages end with an F7.

After the JV-1010 receives a "Data Request 1", it will return a "Data Set 1" containing the value of the setting you requested. (You can also send a "Data Set 1" to the JV-1010 to change a setting to a new value). Let's take a look at the JV-1010's "Data Set 1" message. It has 11 bytes, plus the data bytes for the setting's value. (And remember that, for settings that exceed the limit of 128 steps, there will need to be more than one data byte). Here is the JV-1010 "Data Set 1" message:

F0	All SysEx messages begin with an F0.
41	All Roland SysEx messages have a 41 here.
DEV_ID	This is the device ID.
6A	This is the JV-1010 model ID.
12	This is the command ID. A "Data Set 1" SysEx will have a 12 here.
ADDR_1	This is the first byte (of the 3 or 4 bytes) that make up the address of the setting whose value is in this "Data Set 1". What number is here depends upon which setting's value is in this "Data Set 1". This first byte is often referred to as the Most Significant Byte (MSB).
ADDR_2	This is the second byte of the address.
ADDR_3	This is the third byte of the address.
ADDR_4	This is the fourth and last byte of the address. Note: For some Roland models, there are only three address bytes, and so this one is omitted.
DATA_1	This is the first byte of the settings value. For a setting that is within the range of 0 to 128, there will be only this one byte. For a setting whose value can be larger than this range, it will broken up into several bytes, and what would follow is a **DATA_2**, and possibly **DATA_3**, etc. We'll later discuss an example of a setting that requires more than one data byte.
CHECKSUM	This is the checksum. What value you put here depends upon what numbers you use for the address and data bytes above. After all, the checksum is essentially just the sum of all of those preceding bytes.
F7	All SysEx messages end with an F7.

If you look in the JV-1010 manual, you'll notice that there is another form of the "Data Set 1" message. It is almost the same as the above message, except for the model ID, and the fact that it has only 3 address bytes instead of 4. The model ID is 42 (which is a generic Roland model number), and this is a generic "Data Set 1" format. Here's the deal. There are some settings on the JV-1010 which are exactly the same on dozens of other Roland models. For those settings, Roland decided to instead use a second, generic "Data Set 1" format. In this way, all of the models can respond to this same generic "Data Set 1" for those identical settings. But only certain settings can be transfered with this generic "Data Set 1" -- the settings that Roland has made uniform across all of its models. Those settings (often referred to as "General Standard" or "GS" settings) will be noted in the address map, so we'll discuss an example later. For other JV-1010 specific settings, you must use the "Data Set 1" containing the JV-1010 model ID shown above.

JV-1010 Address Map

In order to understand the particulars of how to work with a Roland address map, and the "Data Request 1" and "Data Set 1" messages, we'll need to study a particular example. I'll choose the Roland JV-1010. This is a typical Roland module, and among the items that you can edit, are 2 "Rhythm Sets", 128 Patches, and 32 "Performances" (containing "Parts" that reference those 128 User patches, as well as any of the preset patches). If you're unfamiliar with these elements of Roland's architecture, you may wish to read The architecture of Roland sound modules. Although the examples I'll be showing you here are specific to the JV-1010, you'll employ the same concepts with other Roland modules.

The section of the manual that details the JV-1010 address map begins at the label **Parameter Address Map**. This section consists of a series of charts (and some footnotes). The very first chart is an *overview* of where all of the large items are to be found. By large items, I mean things such as "Performances", "Rhythm Sets", "Global settings", and "Patches". If you're familiar with a programming language such as C, think of one of these large items as a "structure". Of course, a large item can be comprised of smaller items. For example, a Performance may contain several "Parts" (structures) inside of it. So later on, there will be a chart that shows you the details of what is inside a Performance. Let's look at the overview of the JV-1010 address map. The chart is as so:

Start Address	Description	
00 00 00 00	System	1-1
01 00 00 00	Temporary Performance	1-2
02 00 00 00	Performance Mode Temporary Patch(part 1)	1-3
02 01 00 00	Performance Mode Temporary Patch(part 2)	
:		
02 08 00 00	Performance Mode Temporary Patch(part 9)	
02 09 00 00	Temporary Rhythm Setup	1-4
02 0A 00 00	Performance Mode Temporary Patch(part 11)	1-3
:		
02 0F 00 00	Performance Mode Temporary Patch(part 16)	
03 00 00 00	Patch Mode Temporary Patch	1-3
10 00 00 00	User Performance USER:01	1-2
10 01 00 00	User Performance USER:02	
:		
10 1F 00 00	User Performance USER:32	
10 40 00 00	User Rhythm Setup USER:1	1-4
10 41 00 00	User Rhythm Setup USER:2	
11 00 00 00	User Patch USER:001	1-3
11 01 00 00	User Patch USER:002	
:		
11 7F 00 00	User Patch USER:128	

What this above chart is showing you is that there is a structure called a **System** and it is located (starting) at address 00 00 00 00. You don't yet know what settings (ie, fields) are contained inside of this System structure. There will be another chart later that will show you the actual contents of this System structure. Right now, all you know is that it starts at address 00 00 00 00. (ie, The very first setting inside of the System structure will be at address 00 00 00 00. The remaining settings inside of it will be at address 00 00 00 01 and above).

The chart also shows you that there is a structure called a **Temporary Performance** and it is located (starting) at address 01 00 00 00. Again, you don't yet know what settings are contained inside of this Temporary Performance structure. There will be another chart later that will show you the actual contents of it. Right now, all you know is that it starts at address 01 00 00 00. (ie, The very first setting inside of the Temporary Performance structure will be at address 01 00 00 00. The remaining settings will be at address 01 00 00 01 and above).

If you look down the chart, you will see numerous other structures shown. For example, the JV-1010 has 32 User Performances (ie, Performances whose settings can be edited). The settings for each performance are stored in a **Performance** structure. Again, you don't yet know the details of what is inside of it. But if you look at the chart, you will see that there are 32 of them. (Actually, only 3 of the 32 are shown in the chart. The dots in the chart, where an address would normally be shown, indicate that some items have been omitted due to repetition. You're supposed to mentally fill in the blanks). You can see that the **Performance** structure for the first User Performance (ie, 1) starts at address 10 00 00 00. The **Performance** structure for the second User Performance starts at address 10 01 00 00. Although they aren't shown, you can guess (by the repeating pattern) that the **Performance** structure for the third User Performance will start at address 10 02 00 00, and the **Performance** structure for fourth User Performance will start at address 10 03 00 00. The **Performance** structure for User Performance 32 (ie, the last one) starts at address 10 1F 00 00.

The JV-1010 has two User Rhythm Sets (ie, drum kits). If you look at the chart, you will see that the settings for each Rhythm Set are stored in a **User Rhythm Setup** structure. Again, you don't yet know the details of what is inside of it. But if you look at the chart, you will see that the **User Rhythm Setup** structure for the first kit starts at address 10 40 00 00, and the **User Rhythm Setup** structure for the second kit starts at address 10 41 00 00.

Finally, the JV-1010 has 128 User Patches. You can see that they are at the bottom of the chart. A patch's settings are stored in a **User Patch** structure, and the first patch's structure starts at address 11 00 00 00. The last patch's structure starts at address 11 7F 00 00. Again, many of them have been omitted due to repetition (and to make the chart therefore smaller), and you have to mentally fill in the blanks.

So, this chart is an overview of where (ie, at which address) all of the structures (configurable settings) will be found. But you still don't know the details of what settings are inside each structure. That is what the other charts are for. If you look in the above chart, you'll notice that there are numbers on the far right. For example, beside the **System** structure, you see the designation 1-1. What this tells you is that, the details for the **System** structure can be found in another chart labeled 1-1. Sure enough, if you look at the other charts in the manual, you'll find one that is labeled 1-1 System. This shows you the contents of the **System** structure. Let's take a look at that chart:

Offset Address	Description	
00 00	System Common	1-1-1
10 00	Part 1 Scale Tune	1-1-2
11 00	Part 2 Scale Tune	
:		
1F 00	Part 16 Scale Tune	
20 00	Patch Mode Scale Tune	1-1-2

Ok, this structure starts with another structure inside of it, called a **System Common** structure. This is something that you'll need to know -- sometimes structures have other structures inside of them. The above chart isn't showing you the contents of that **System Common** structure, but you can probably guess from the designation 1-1-1 next to it that there will be another chart labeled 1-1-1 System Common, and it will show you what is inside this System Common. All you know from the above is that, the System Common is at an address offset of 00 00 within the System structure. What is meant by an address offset? Well, this offset is added to the address shown in the overview chart in order to determine the address that you'll use with the "Data Request 1" or "Data Set 1" message. If you review the overview chart, you'll recall that the System structure begins at address 00 00 00 00. So we add the offset for the System Common as so:

```
    00 00 00 00
+         00 00
---------------
    00 00 00 00
```

In other words, the System Common structure (within the System structure) starts at address 00 00 00 00.

There is another structure called **Part 1 Scale Tune** inside of the System structure. Let's consider the **Part 1 Scale Tune** structure. First of all, you'll find the details of this structure in another chart labeled with the designation 1-1-2 Scale Tune. There are 17 such structures within a System structure, with the first one at offset 10 00 (and the last at offset 20 00). So for example, if we wanted to find the start address of Part 16 Scale Tune, we add to the System's start address as so:

```
   00 00 00 00
+        1F 00
---------------
   00 00 1F 00
```

In other words, the Part 16 Scale Tune structure (within the System structure) starts at address 00 00 1F 00.

Ok, let's take a look at the **System Common** structure. You'll find it in the chart labeled 1-1-1 System Common, and it looks like this (actually, it is a rather large chart, so I am going to reproduce only the beginning of it):

Offset Address	Size	Description	Data (value)
00 00	0000 00aa	Sound Mode	0 - 2 *1
00 01	0aaa aaaa	Performance Number	0 - 127 *2
00 02	0000 00aa	Patch Group Type	0 - 2 *3
00 03	0aaa aaaa	Patch Group ID	0 - 127
# 00 04	0000 aaaa	Patch Number	0 - 255
	0000 bbbb		(001 - 256)

Now we're actually getting to some Roland settings. You'll notice that this chart lists numerous settings and their allowable "data values" (ie, the range of values that the setting is allowed to encompass). Note above that the data values for "Sound Mode" are shown as 0 - 2. This means that the Sound Mode's data value can be 0, 1, or 2. The data values for "Performance Number" are 0 - 127. This means that the data value can be any number from 0 to 127 inclusive.

One thing that the Roland manual doesn't really tell you is what each setting does. For example, what is the "Sound Mode" setting above? Sometimes there will be a footnote beneath the chart to help clue you in. For example, on the far right in the above chart, you'll see a *1 shown beside Sound Mode's data range. This is a footnote indication. It means that, below this chart, you'll find some footnotes, and footnode 1 tells a little more about what the Sound Mode data values mean. Indeed, below the chart, you'll see footnote 1 as so:

```
*1      Performance, Patch, GM
```

This is telling you that, when Sound Mode's value is 0, this pertains to "Performance" mode. When Sound Mode's value is 1, this pertains to "Patch" mode. And when Sound Mode's value is 2, this pertains to "GM" mode. You can probably guess from this that the Sound Mode setting determines whether the JV-1010 is in Performance (multi-timbral) mode, Patch (single patch playback only) mode, or GM (automatically limited to its 128 GM patches in multi-timbral) mode.

Let's look at Performance Number. It has footnote 2 which is:

```
*2      USER:01 - USER:32, <CARD:01 - CARD:32>, PR-A:01 - PR-A:32, PR-B:01 - PR-B:32
```

This is telling you that, when Performance Number's value is 0 to 31, this is one of the 32 User Performances. When Performance Number's value is 32 to 63, this is one of the 32 Performances on a Voice Expansion card plugged into the JV-1010 (if there is such a card. The brackets indicate that this is an optional component). When Performance Number's value is 64 to 95, this is one of the 32 Preset Performances in bank A. And when Performance Number's value is 96 to 127, this is one of the 32 Preset Performances in bank B. You can guess from this that Performance Number indicates the currently

selected Performance (and the Performance selected when the JV-1010 powers up).

So the footnote tells you how to regard the data values, and therefore can be used to deduce the purpose of some setting.

As you peruse the above chart, you'll see that each setting has its own address offset. Now, remember that this System Common structure is inside of the System structure. So, to find the address of the Sound Mode setting, we need to add the Sound Mode offset to the System Common offset (within System), and then add to the address of System as it appears in the overview chart. Here is the result:

```
    00 00 00 00        (Address of System within overview)
+         00 00        (Offset of System Common within System)
+         00 00        (Offset of Sound Mode setting within System Common)
--------------
    00 00 00 00
```

In other words, Sound Mode setting is at address 00 00 00 00.

How about the Performance Number? It's address is:

```
    00 00 00 00        (Address of System within overview)
+         00 00        (Offset of System Common within System)
+         00 01        (Offset of Performance Number setting within System Common)
--------------
    00 00 00 01
```

In other words, Performance Number setting is at address 00 00 00 01.

Now let's look at the *Size* shown for each setting. This is a bit of a misnomer. What the chart is showing you isn't really the size, but rather, showing you how the value is "broken up" into data bytes within the "Data Set 1" message. Roland displays the bytes in binary notation (ie, all 8 bits of each data byte, displayed as nibbles). If you're unfamiliar with this, you may need to read up on binary notation. For example, the Size shown for Sound Mode is:

```
0000 00aa
```

The aa is where Sound Mode's data value goes. (ie, Bits 0 and 1 are the actual data, and the remaining bits are always 0). This make sense because Sound Mode can be only 0, 1, or 2. And it takes only 2 bits to express that data range. Because most Roland data values are within the range of 0 to 127, Roland typically right-justifies the bits in one data byte, and so the data value is put into the "Data Set 1" message exactly as it is specified in the chart. In other words, you'll actually use the data byte of 0, 1, or 2 verbatim in the "Data Set 1" message to set Sound Mode.

Ok, we're just about ready to create an example "Data Set 1". Let's assume that you wish to set Sound Mode to the value 2. Remember that we figured out Sound Mode's address as 00 00 00 00. So we need to plug those 4 bytes into the "Data Set 1" for ADDR_1, ADDR_2, ADDR_3, and ADDR_4 respectively. And we can see from the chart that Sound Mode requires only 1 byte (because there are only 8 bits shown for its Size). So, we'll put the value 2 as a single DATA_1 byte in the "Data Set 1". Here then is our message to set Sound Mode to the value 2 (assuming a device ID of 10):

F0	
41	
10	
6A	
12	
00	This is the first byte (of the 4 bytes) that make up the address of the Sound Mode setting.
00	This is the second byte of the address.

00	This is the third byte of the address.
00	This is the fourth and last byte of the address.
02	This is the data byte to set Sound Mode to 2.
CHECKSUM	This is the checksum. What value you put here depends upon what numbers you use for the address and data bytes above.
F7	

All that remains is to calculate the checksum, which we'll learn to do later, and you can send this message to the JV-1010 to immediately change the Sound Mode to 2 (ie, GM mode).

Let's take another example to set Performance Number to 127 (ie, 7F hexadecimal). We already saw that its address is 00 00 00 01. And from the chart, you can see that its Size is still just one byte (ie, 8 bits shown). The bits are right-justified, so we use the data value 7F verbatim. Here is the "Data Set 1" message:

F0	
41	
10	
6A	
12	
00	This is the first byte (of the 4 bytes) that make up the address of the Performance Number setting.
00	This is the second byte of the address.
00	This is the third byte of the address.
01	This is the fourth and last byte of the address.
7F	This is the data byte to set Performance Number to 127.
CHECKSUM	This is the checksum. What value you put here depends upon what numbers you use for the address and data bytes above.
F7	

What if you wanted to query the current value of Performance Number? You send a "Data Request 1" asking for it, and the Roland returns the above "Data Set 1". Here is the "Data Request 1" to query Performance Number (again assuming a device ID of 10). Note that since we expect only one data byte in the returned "Data Set 1", our SIZE_1, SIZE_2, SIZE_3, and SIZE_4 are 00 00 00 01.

F0	
41	
10	
6A	
11	
00	This is the first byte (of the 4 bytes) that make up the address of the Performance Number setting.
00	This is the second byte of the address.
00	This is the third byte of the address.
01	This is the fourth and last byte of the address.
00	This is the first byte (of the 4 bytes) that make up the count of the expected data bytes.
00	This is the second byte of the count.
00	This is the third byte of the count.
01	This is the fourth and last byte of the count.
CHECKSUM	This is the checksum. What value you put here depends upon what numbers you use for the address and size bytes above.

F7

Checksum

Now it's time to figure out what Checksum value to use. You have to individually calculate the Checksum for each SysEx message (since it will likely be different in various messages).

When calculating the Checksum for a "Data Set 1", you must add up the 4 address bytes and the data bytes. Then you divide by 128 and take the remainder. Finally, you subtract the remainder from 128. You should end up with a value of 128 or less if you did it correctly. This is the Checksum. If the Checksum is 128, then substitute a value of 0.

Let's take the following example:

F0	
41	
10	
6A	
12	
00	This is the first byte (of the 4 bytes) that make up the address of the Performance Number setting.
00	This is the second byte of the address.
00	This is the third byte of the address.
01	This is the fourth and last byte of the address.
7F	This is the data byte to set Performance Number to 127.
CHECKSUM	This is the checksum. We'll calculate it below.
F7	

The 4 address bytes are 0, 0, 0, and 1, and the data byte is 7F (ie, 127). The sum is therefore 128. If you divide 128 by 128, you get a remainder of 0. (ie, 128 goes into 128 exactly once, so there is a remainder of 0). If you then subtract 0 from 128, you end up with a Checksum of 128. So, we substitute a value of 0. Here then is the completed "Data Set 1":

F0	
41	
10	
6A	
12	
00	
00	
00	
01	
7F	
00	This is the checksum.
F7	

A typical Roland manual will have a section explaining the Checksum, and list a few examples.

The Checksum for a "Data Request 1" is calculated in the same way, except you add up the 4 address bytes and the 4 size (count) bytes.

To aid in calculating checksums, you can download my Roland Checksum Calculator for Windows. C Programmers can download the C source code as an example, or MIDI REXX programmers can download a REXX version.

Settings with multiple DATA bytes

There are some settings whose values are too large for only one data byte in a "Data Set 1". Let's take an example of such a setting. In the **System Common** structure, the Patch Number setting is such an example. Let's review the chart for the System Common:

Offset Address	Size	Description	Data (value)
00 00	0000 00aa	Sound Mode	0 - 2 *1
00 01	0aaa aaaa	Performance Number	0 - 127 *2
00 02	0000 00aa	Patch Group Type	0 - 2 *3
00 03	0aaa aaaa	Patch Group ID	0 - 127
# 00 04	0000 aaaa	Patch Number	0 - 255
	0000 bbbb		(001 - 256)

First, you'll notice that there is a # sign next to the address offset. Roland uses this to indicate a setting that requires its value to be broken up into more than one byte in a "Data Set 1" message. The next thing that you'll notice is that the allowable range for Patch Number is any value from 0 to 255 inclusive. Obviously, this is beyond the range limit of 128 steps, and that's why we require more than one byte. In fact, if you look at the Size field, you'll see that we need 2 bytes. After all, there are two groups of 8 bits shown, as so:

```
0000 aaaa
0000 bbbb
```

What you should notice from the above is that you are expected to break apart the Patch Number value into 2 bytes, by putting the high 4 bits into the first byte (right-justified), and the low 4 bits into the second byte (also right-justified). For example, let's assume that we want to set the Patch Number to the value 200. That would be hex C8. So, we take the C and put it in the first byte, and take the 8 and put it in the second byte. Of course, for the address bytes, we calculate them as so:

```
   00 00 00 00      (Address of System within overview)
+        00 00      (Offset of System Common within System)
+        00 04      (Offset of Patch Number setting within System Common)
---------------
   00 00 00 04
```

And to calculate the Checksum, we add the 4 address bytes, and the 2 data bytes (0 + 0 + 0 + 4 + 12 + 8), divide by 128 and get the remainder (24), and subtract this remainder from 128 (104, or hex 68).

So, here is our "Data Set 1" message:

F0	
41	
10	
6A	
12	

00	This is the first byte (of the 4 bytes) that make up the address of the Patch Number setting.
00	This is the second byte of the address.
00	This is the third byte of the address.
04	This is the fourth and last byte of the address.
0C	This is the first data byte to set Patch Number to C8.
08	This is the second data byte to set Patch Number to C8.
68	This is the checksum.
F7	

Of course, to request the currrent value of Patch Number, we need to send a "Data Request 1", and indicate that we expect back 2 data bytes.

F0	
41	
10	
6A	
11	
00	This is the first byte (of the 4 bytes) that make up the address of the Patch Number setting.
00	This is the second byte of the address.
00	This is the third byte of the address.
04	This is the fourth and last byte of the address.
00	This is the first byte (of the 4 bytes) that make up the count of the expected data bytes.
00	This is the second byte of the count.
00	This is the third byte of the count.
02	This is the fourth and last byte of the count. Note that we expect back 2 bytes
7A	This is the checksum.
F7	

Transfering an entire structure

If you wanted to dump all of the Roland JV-1010 settings to a computer, it would take a very long time to do so if you had to individually request each setting's value, and receive back a "Data Set 1" for each setting. Fortunately, Roland's implementation is more sophisticated and allows you to request (with a single "Data Request 1") an entire structure's settings. All those settings will be sent back in a single "Data Set 1" message, containing all of the data bytes for all of the settings. The first data byte(s) in the "Data Set 1" will be for the first setting within the structure. The next data byte(s) will be for the second setting, etc.

When you request an entire structure's settings, there are 2 things that you need to do with your "Data Request 1":

1. For the ADDR_1, ADDR_2, ADDR_3, and ADDR_4, you must use the address of the first setting within the structure.
2. For the SIZE_1, SIZE_2, SIZE_3, and SIZE_4, you must specify the total number of data bytes for all of the settings which you expect returned in the "Data Set 1".

Let's assume that you wish to request the entire **System Common** structure (within the System structure). Remember that the System Common structure starts at address 00 00 00 00. (ie, Its first setting, "Sound Mode", is at that address). So that's what we use for ADDR_1, ADDR_2, ADDR_3, and ADDR_4.

Now we just need to figure out how many data bytes we expect back. If you peruse the **System Common**'s chart (ie, chart

labeled 1-1-1), and look at the bottom of the chart, you'll see the designation *Total size* and it is shown as 00 00 00 62. (In other words, there are a total of 98 data bytes to be transfered (in a "Data Set 1") when the entire System Common structure's settings are transfered.

So, here is our "Data Request 1" to ask the JV-1010 to return a "Data Set 1" containing the 98 bytes that comprise the values of all settings in the **System Common** structure:

F0	
41	
10	
6A	
11	
00	This is the first byte (of the 4 bytes) that make up the address of the System Common's first setting.
00	This is the second byte of the address.
00	This is the third byte of the address.
00	This is the fourth and last byte of the address.
00	This is the first byte (of the 4 bytes) that make up the count of the expected data bytes.
00	This is the second byte of the count.
00	This is the third byte of the count.
62	This is the fourth and last byte of the count. Note that we expect back 98 bytes
1E	This is the checksum. We add up the address and size bytes (0 + 0 + 0 + 0 + 0 + 0 + 0 + 98), divide into 128 and take the remainder of 98, and then subtract from 128 to get a checksum of 30 (or 1E hex)
F7	

The JV-1010 will now send back a "Data Set 1" containing those 98 data bytes. The first byte will be Sound Mode's value. The second byte will be Performance Number's value. The third byte will be Patch Group Type's value. The fourth byte will be Patch Group ID's value. The fifth and sixth bytes will be Patch Number's value. (Remember that Patch Number's value must be broken up into two bytes). Etc.

You can likewise send a single "Data Set 1" containing 98 data bytes to set the values of all of the System Common's settings.

There is one final caveat that you need to be aware of when tranfering an entire structure. In order to avoid any problems with hardware input buffers, Roland limits a "Data Set 1" message to a maximum of 256 bytes. Because there are 11 other bytes in a JV-1010's "Data Set 1", that means that there can be only 245 data bytes in the message. So, what happens if you need to transfer a structure with a size greater than 245? Well, you have to use more than one "Data Set 1" message to transfer the settings. You'll have to send the first 245 data bytes (or so) in the first "Data Set 1", and then follow-up with a second "Data Set 1" which picks up with the remaining data bytes. The address you will use in the second message will be for the setting whose data byte starts off the second message. So, you may need to do some math to add 245 (or so) to the address that you used in the previous message.

<div align="center">**Delays**</div>

You should always give a Roland module about 40 milliseconds to process a "Data Set 1" message that you send it, before subsequently sending another MIDI message (including another "Data Set 1") to the module. If you don't, the module may report an error.

```
/* Checksum.rex
 *
 * Demonstrates how to calculate a Roland System Exclusive checksum
 * for a "Data Set 1" or "Data Request 1" message. Also demonstrates
 * how to convert the message to binary and output it to a Roland
 * module (using MIDI REXX).
 *
 * The main window has an ENTRY box labeled "Address" into which the
 * user types the 3 or 4 address bytes (each separated by a space).
 * There is also a DROP box next to this ENTRY. It contains the items
 * "Decimal", "Hex", and "Binary". The user can type the address bytes
 * in any one of these formats, and then indicate which one he is using.
 *
 * There is also an ENTRY box labeled "Data", and a COMBO box beside it.
 * This is where the user enters the data byte(s), and indicates which
 * format he is using.
 *
 * There is a "Calculate" button to click upon in order to calculate the
 * Roland checksum based upon what he typed into Address and Data. This
 * checksum is then displayed at the bottom of the window (using a TEXT
 * control).
 *
 * NOTE: If the user wishes to calculate the checksum for a "Data Request
 * 1" message, then he can type the 3 or 4 size (ie, count) bytes in the
 * Data box.
 */

ADDRESS NULL

/* Trap HALT and SYNTAX in any scripts we call which don't
 * trap HALT or SYNTAX themselves.
 */
OPTIONS 'TRAP'

/* Trap SYNTAX/ERROR/HALT, and ask for ERROR raising. Disable
 * ERROR raising for RXMSG(). We'll test for errors in our
 * loop
 */
SIGNAL ON HALT
SIGNAL ON SYNTAX
SIGNAL ON ERROR
CALL RXERR('ERROR|DISPLAY', '~!MSG')


/* =============== Create "Roland Checksum Calculator" Window =============== */
/* Default size and position. WindowID = 'RX' */
Rx = ''
```

```
/* Use a group of 2 ENTRY controls to enter the Address and Data bytes */
RxType.1 = 'ENTRY'

/* No special flags */
RxFlags.1 = ''

/* Labels for each entry, and Groupbox. Note: No groupbox */
RxLabel.1 = '&Address:|&Data:|'

/* Variable name where the text "typed into" the entries is stored */
RxVal.1 = 'TEXT'

/* RXINFO not needed */

/* ControlsPerLine, X Position, Y Position, Width */
RxPos.1 = '1 10 10 120'

/* The initial text for the 2 entries (initially blank) */
TEXT.1 = ''
TEXT.2 = ''

/* Use a TEXT Group to label the checksum */
RxType.2 = 'TEXT'

/* No border */
RxFlags.2 = 'NOBORDER'

/* No groupbox */
RxLabel.2 = 'Checksum:|'

/* ControlsPerLine (phrases per line), X Position, Y Position */
RxPos.2 = '1 10 62'

/* Use another TEXT Group to display the checksum */
RxType.3 = 'TEXT'

/* No border */
RxFlags.3 = 'NOBORDER'

/* Text line (initially blank), and no groupbox */
RxLabel.3 = '|'

/* ControlsPerLine (phrases per line), X Position, Y Position, Width, BetweenPhrases. Note
 * we deliberately set WidthOfPhrase non-0 to allow enough width to accomodate
 * a range of text digits
 */
RxPos.3 = '1 90 62 84 0'
```

```
/* Use a DROP group for "Decimal", "Hex", or "Binary" selection for address bytes */
RxType.4 = 'DROP'

/* User can't type into the ENTRY, and we want the index returned */
RxFlags.4 = 'NOBORDER|READONLY|INDEX'

/* Labels for each dropbox, and Groupbox. Note: No labels or Groupbox */
RxLabel.4 = '||'

/* Stem variable names to fetch the strings for each dropbox in the group */
RxVal.4 = 'ADDRTYPE | DATATYPE'

/* RXINFO not needed */

/* ControlsPerLine, X Position, Y Position, Width */
RxPos.4 = '1 210 4 90'

/* Items for the address type */
ADDRTYPE.1 = 'Decimal'
ADDRTYPE.2 = 'Hex'
ADDRTYPE.3 = 'Binary'
ADDRTYPE.4 = ''  /* Marks the end of the list */

/* Items for the data type */
DATATYPE.1 = 'Decimal'
DATATYPE.2 = 'Hex'
DATATYPE.3 = 'Binary'
DATATYPE.4 = ''  /* Marks the end of the list */

/* Use a PUSH group for the Calculate button */
RxType.5 = 'PUSH'

/* RESULT so that clicking it returns immediately */
RxFlags.5 = 'RESULT'

/* Label, and no groupbox */
RxLabel.5 = '&Calculate|'

/* No default value for button */
RxVal.5 = ''

/* ControlsPerLine, X Position, Y Position */
RxPos.5 = '-1 210 62'

/* Accelerators */
RxAccel = '1 1 _C_ A | 1 2 _C_ D | 5 1 _C_ C'
```

```
/* Specify NOCLOSE (since we'll do the closing ourselves), and
 * RESULT so that we can respond to the ENTER key in order to
 * perform the Checksum calculation. Also set SETVAL so that REXX
 * Dialog updates all of our RXVAL variables every time that we
 * return from RXMSG(). This is not a time critical script so we
 * don't care about the overhead. It saves us from having to do
 * RXQUERY() to grab what the user typed into Address and Data.
 */
CALL RXCREATE('Rx', 5, 'Roland Checksum Calculator', 'NOCLOSE|SETVAL|RESULT', RxAccel)

/* Initialize the Address type to "Hex" and Data type to "Decimal" */
CALL RXSET(, 'VALUE', 1, 4, 1)
CALL RXSET(, 'VALUE', 0, 4, 2)

/* Initially select the "Address" ENTRY */
CALL RXSET(, 'ACTIVE', , 1, 1)

/* Do our window message loop */
DO FOREVER

  /* Because we have only 1 window open, we don't bother to specify a WindowID
   * to RXMSG(). RXMSG() will set RXID and RXSUBID.
   *
   * Also, because we specified SETVAL, REXX Dialog will set the VAL variables
   * for all controls.
   *
   * NOTE: Our window is NOCLOSE. RXMSG() will return if the user clicks the CLOSE
   * box. But the window won't be automatically closed. We'll do the closing
   * ourselves with an explicit call to RXMSG() using an 'END' operation.
   */

  /* Wait on the window. We go to sleep while user manipulates windows,
   * until such time as the user presses ESC or ENTER, or tries to close a window
   * using its CLOSE BOX, or uses a RESULT Group or a Control within some Group
   * that has its REPORT Flag set, or an abort signal. Other things that could
   * cause RXMSG() to return are a window timeout, or if the user pressed a key
   * in a window with its KEYS Flag set, or user resizes a window with its NEWSIZE
   * flag set... but we haven't utilized those features here.
   */
  err = RXMSG()  /* NOTE: No WindowID or Operation means a PROCESS operation on any window */

  /* RXWIND now specifies which window woke us up. (The window is still
   * there because we specified NOCLOSE). RXID and RXSUBID tell us what
   * action caused RXMSG() to return. Handle this action.
   */
  IF err == '' THEN INTERPRET 'CALL' RXWIND || '_' || RXID || '_' || RXSUBID

  /* Do the next interaction */
```

END

```
/* ============= ESC key for "Roland Checksum Calculator" window ============== */
RX__ESC:

  /* Drop down into handling for CLOSE BOX */

/* ============ CLOSE BOX for "Roland Checksum Calculator" window ============= */
RX__END:

  /* Close this window */
  CALL RXMSG(RXWIND, 'END')

  /* Exit our script. We never return to the main loop */
  EXIT



/* ============= ENTER key for "Roland Checksum Calculator" window ============== */
RX_0_ENTER:

  /* ENTER key sets the VAL variables of all Groups, so drop down into
   * the handling of the "Calculate" button.
   */

/* ========= Group 1, Control 1 for "Roland Checksum Calculator" window ========= */
RX_5_1:

  /* This is the "Calculate" button. Note that because we used SETVAL in our
   * window, this RESULT button causes RXMSG() to return with all of our VAL
   * variables set. For example, TEXT.1 contains the address bytes typed into
   * the "Address" box, and TEXT.2 contains the data bytes typed into the "Data"
```

```
 * box. ADDRTYPE.0 contains 0, 1, or 2 depending upon whether the user selected
 * "Decimal", "Hex", or "Binary" for the DROP box for the address bytes. And
 * DATATYPE.0 contains the same for the data bytes DROP box.
 */

/* Start with a checksum of 0 */
Checksum = 0

/* Now add up all of the address bytes. We must chop off each byte from our
 * VAL variable for Address. If the bytes are expressed in hex or binary, we must
 * first convert to decimal.
 */
next = "0"
string = TEXT.1
DO WHILE next \= ""

   /* Break off the next byte */
   PARSE VAR string next string

   /* Was there another byte? */
   IF next \= "" THEN DO

      /* If it was expressed in hex or binary, we must convert to decimal */

      /* "Hex"? */
      IF ADDRTYPE.0 == 1 THEN next = X2D(next)

      /* Binary? */
      ELSE IF ADDRTYPE == 2 THEN next = X2D(B2X(next))

      /* Add to sum */
      Checksum = Checksum + next

   END

END

/* Now add up all of the data bytes. We deal with them the same way as the
 * address bytes above, and add to the previous sum.
 */
next = "0"
string = TEXT.2
DO WHILE next \= ""

   PARSE VAR string next string

   IF next \= "" THEN DO
```

```
      IF DATATYPE.0 == 1 THEN next = X2D(next)

      ELSE IF DATATYPE == 2 THEN next = X2D(B2X(next))

      Checksum = Checksum + next

    END

  END

  /* Now finish our checksum calculation by dividing by 128 and getting the
   * remainder, and then subtracting from 128, and accounting for 128 = 0
   */
  Checksum = 128 - (Checksum // 128)
  IF Checksum = 128 THEN Checksum = 0

  /* Print out the checksum value in hex */
  CALL RXSET(RXWIND, 'VALUE', D2X(Checksum) || " (hex)", 3, 1)

  /* ==================================================
   * The following code is commented out, but put here
   * for the sake of showing you how you could stuff
   * those above address and data bytes, and the checksum
   * into a variable and send it off to a MIDI Out port
   * to a Roland module.
   * ================================================== */
  TransmitSysex()

  /* Return to main loop */
  RETURN
```

```
/* ====================== TransmitSysex() ===================== */
/* This just demonstrates how you can use the address bytes,
 * data bytes, and checksum in a "Data Set 1" message sent to
 * a Roland module. I'll choose the JV-1010 here, and format
 * the message with the JV-1010 model ID. I'll also use
 * MIDIRXIO to open a MIDI Out port and send the message.
 *
 * NOTE: TEXT.1 must contain the address bytes, and TEXT.2 contains the
```

```
* data bytes (or size bytes if a Data Request 1). ADDRTYPE.0 is 0, 1, or
* 2 to indicate what format the address bytes are in. DATATYPE.0 is the
* same for the data bytes. And the Checksum has already been calculated
* and stored in Checksum, and assumed to be in decimal format.
*/

TransmitSysex:

  /* Our DataSet1 variable is going to be used to create the actual "Data Set
   * 1" message we'll send to MIDI Out usung MIDIRXIO's MidiIoOutLong().
   * Note that each byte must be expressed in decimal. A Sysex begins
   * with F0 hex, so we can specify F0 and put it in a format suitable for
   * MidiIoOutLong using X2D(). (Or we could just express F0 in decimal, namely
   * 240). Next we have the Roland ID number of 41 hex. Next
   * we have the device ID. We'll use the default of 10 hex. Next we have the
   * model number which will be 6A hex for a JV-1010. (You'll substitute the
   * appropriate model ID for your Roland unit, if not a JV-1010). Finally,
   * since this is a Data Set 1 message, we format a command ID of 12 hex. If
   * we were making a Data Request 1 instead, then we would substitute 11 hex.
   * We concatenate these 5 bytes as so: (Either line would be correct, depending
   * upon whether you prefer to work with hex or decimal).
   */
/*   DataSet1 = X2D('F0') || " " || X2D('41') || " " || X2D('10') || " " || X2D('6A') || " " || X2D('12') */
  DataSet1 = '240 65 16 106 18'

  /* Convert each address byte to a format suitable for MidiIoOutLong(), and
   * append it to our DataSet1 variable. If the bytes are expressed in decimal,
   * then we're all set to go. If hex, we use X2D(). And if binary, then we use
   * then B2D().
   */
  IF ADDRTYPE.0 == 0 THEN DataSet1 = DataSet1 || " " || TEXT.1
  ELSE DO
    next = "0"
    string = TEXT.1
    DO WHILE next \= ""

      /* Break off the next byte */
      PARSE VAR string next string

      /* Was there another byte? */
      IF next \= "" THEN DO

        /* Convert it to a format suitable for MidiIoOutLong(), and append to DataSet1 */
        IF ADDRTYPE.0 == 1 THEN DataSet1 = DataSet1 || " " || X2D(next)
        IF ADDRTYPE.0 == 2 THEN DataSet1 = DataSet1 || " " || B2D(next)

      END
    END
```

```
  END

/* Do the same for the data bytes */
IF DATATYPE.0 == 0 THEN DataSet1 = DataSet1 || " " || TEXT.2
ELSE DO
   next = "0"
   string = TEXT.2
   DO WHILE next \= ""
     PARSE VAR string next string
     IF next \= "" THEN DO
       IF DATATYPE.0 == 1 THEN DataSet1 = DataSet1 || " " || X2D(next)
       IF DATATYPE.0 == 2 THEN DataSet1 = DataSet1 || " " || B2D(next)
     END
   END
END

/* Finally, we append the checksum to DataSet1 along with the final F7 */
DataSet1 = DataSet1 || " " || Checksum || " 247"

/* Now send DataSet1 to MIDI Out */

/* Open MIDI Mapper */
error = MidiIoOpenPort()
IF error == "" THEN DO

   /* Send the message */
   error = MidiIoOutLong('DataSet1')
   IF error \== "" THEN RXSAY(error, 'MIDI Out Error')

   /* Close the port */
   DO UNTIL error == ""
     error = MidiIoClosePort()
   END

END
ELSE RXSAY(error, 'MIDI Out Error')

RETURN
```

```
/* ==================== Error Handling ==================== */
SYNTAX:
  /* Display error box */
  CALL CONDITION('M')

HALT:
ERROR:
  /* NOTE: CONDITION('D') fetches error message. CONDITION('E') fetches the
   * error number. SIGL is the line number where the error occurred.
   * Rexx Dialog has already displayed a message since we specified DISPLAY
   * option.
   */
  CALL RXMSG(,'END')
  EXIT
```