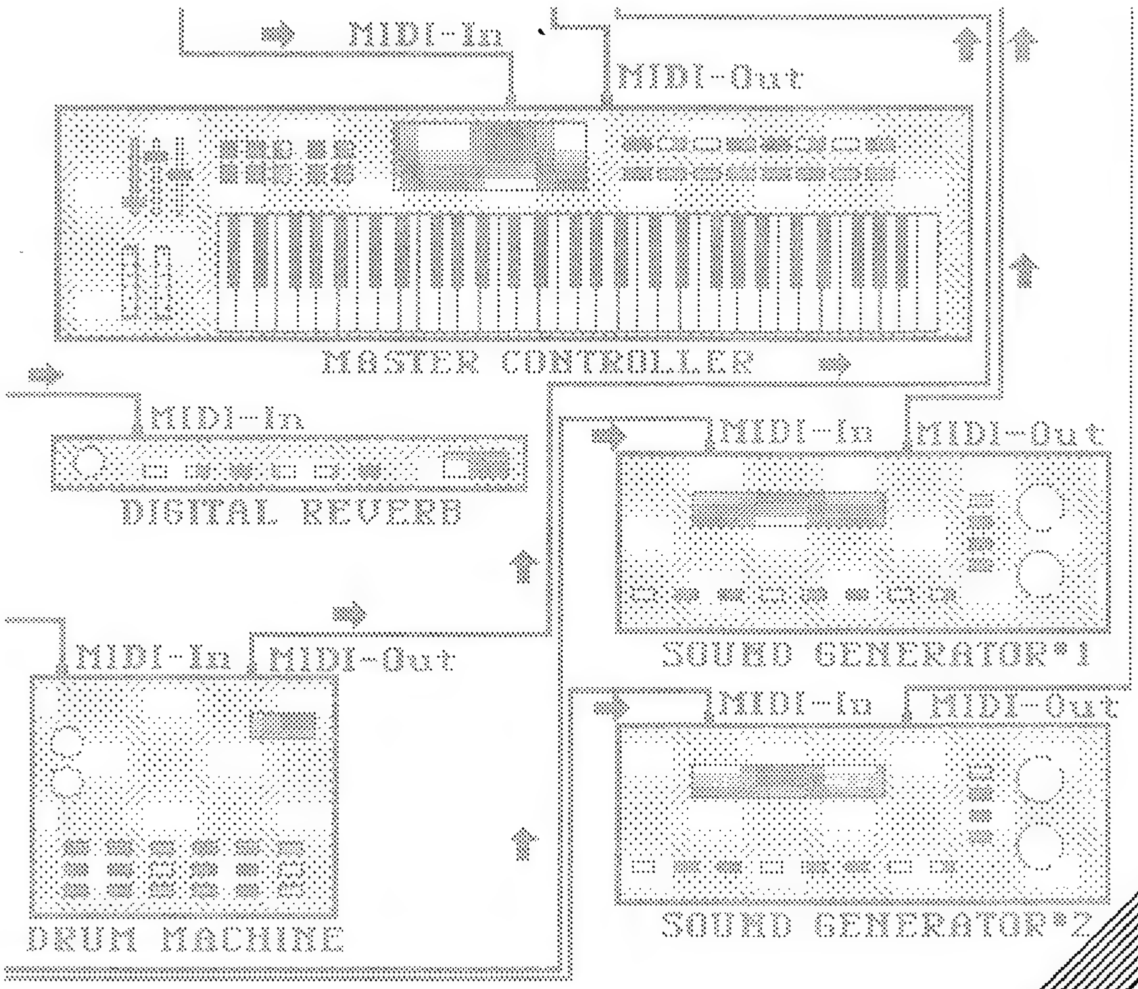
DEMYSTIFYING MIDI

Norman Weinberg



Nimbus Press

MIDI LANGUAGE JUST WHAT IS MIDI?

THE MIDI LANGUAGE

MIDI is the musical buzz word of the '80s. Very few advances in music have had such a profound impact upon the way we can create music, listen to music, and relate to music within our society. MIDI is an abbreviation for "**M**usical **I**nstrument **D**igital **I**nterface" and has changed the face of the music industry more than any single event in recent history.

But, MIDI isn't an event that happened. It's not a product that is produced by a manufacturer. **MIDI is a language**, really nothing more and nothing less. When we speak (verbally) to each other, we have agreed upon certain conventions or rules, so that our ideas can be passed on from one person to another. If I see you walking down the street, and I say, "How's things?", you have a pretty good idea what I'm trying to ask you. You understand what I'm saying because we both know the language, and have agreed that certain sounds are going to mean certain things.

If you think about it, we really have another way we communicate with our language (most languages in fact). We can communicate with each other by using spoken words (or sound waves moving through the air) or we can talk to each other by using written words. With written language, we still must agree that certain words will mean certain things, but there are a few additional problems.

When I write something down on a piece of paper, I have not yet communicated anything to anyone. My idea does not pass from my mind to yours until you read it. This book is a good example of that point. If you simply buy it, you own the words, but I haven't yet communicated with you until you read what is on the paper.

Another problem develops when we want to use the language to describe itself. How does language work? How do the letters fit into words, and then into complex ideas? In order to examine the spoken language with written language, we have to create a lot of new words. Some words, like verb, noun, and adjective are the simple ones that most people understand. Some of the words that describe the way our language works might be a little harder to understand like prepositions, participles, and conjunctions. These are the ones that might have given you trouble in school.

Spoken words, however, get the ideas across in an almost instantaneous manner. When we talk, it takes no extra effort, other than listening, to hear what is being said as long as we understand the language.

The spoken version of MIDI is what the instruments do when they are hooked up in some sort of configuration that uses MIDI. You can't really "speak" MIDI to an electronic instrument. When one instrument plays, the other instruments hears the communication, and responds in an instantaneous manner. The written MIDI language is different. It can be looked at, taken apart, labeled, and analyzed in much the same way that you studied the English language in school. This written version of MIDI is what we will be taking a look at in this book.

"Oh my God, is this guy going to take us back to high school English class?" Maybe yes ... just a little. But, there are going to be times when you're working with your MIDI system and everything just plain goes haywire. This is when knowing these names, labels, and structures will come in handy. Believe it! At the very least, you want to be able to cuss them out properly by using the correct terms. Before getting into the real meat of MIDI, let's take a brief look at some of the background of this language.

A BRIEF LOOK AT MIDI HISTORY

The roots of MIDI date back only a few short years. Electronic instruments themselves were only invented a few years ago (compared to the violin or timpani). In 1983, a group of interested people got together and worked out a system by which musical instruments (notably synthesizers) could communicate with each other. In order to see how this might be accomplished, lets take a look at what a synthesizer might do when you push down a key and try to play a note.

Inside all synthesizers is a small computer— a microprocessor — which contains the information that is required to make a sound. When you press down a key on a keyboard synthesizer, there is an electrical contact that tells the microprocessor which key you are pushing down. This electrical impulse is usually a certain amount of voltage that is read by the microprocessor.

The microprocessor has been programmed at the factory so that a certain voltage will cause it to play a certain pitch. For the purpose of this example, we can say that a particular voltage is going to play the pitch of "G#". The microprocessor doesn't really care where that voltage originated. It might have been sent from the keys of the synth. It might have been sent by some glitch in the electrical power supply, or it could have been sent by a toaster. The computer chip doesn't know, and it just doesn't care. As long as it reads that particular voltage, it's going to play that same G#.

This is how the idea of a communication language originated. If the microprocessor doesn't care where it gets its voltage from, why not send the voltage from another synthesizer's keys. This would mean that a musician could play one keyboard and hear two different sounds. One sound would come from the instrument that was actually being played, and another sound would come from the instrument that was also receiving the voltages.

In fact, this is how it was done before the advent of MIDI. There are several pre-MIDI instruments that can receive their voltage instructions from an external synth. This worked fine for a while, but there were some major problems.

An instrument from manufacturer X might use different voltage levels than an instrument made by company R. These two different voltage set-ups could also be different than those used by brand S. The end result was that many instruments were not compatible with instruments of another brand. Many musicians felt that the time for some sort of standard between electronic instruments was long overdue.

MIDI is really the brainchild of many people, but the primary force behind the standard was Dave Smith of the Sequential Circuits company. As early as 1981, he, along with Tom Oberheim of Oberheim Electronics, and I. Kakehashi of the Roland Corporation met to discuss the possibility of using a standard for synthesizer communications.

The original idea for this standard was to be called USI (Universal Synthesizer Interface). By November of the same year, the universal interface was made public at a meeting of the Audio Engineers Society. The concept of the USI had gained a lot of followers, and in January of 1982, there was a meeting of people from many facets of the music industry. Sequential Circuits, Roland, Korg, Yamaha, Kawai, E-Mu, Oberheim and others were all in attendance.

The end result was that USI became MIDI in the effort to include **all** musical instruments, not only synthesizers. Along with changing the name, the interface was expanded to include many more performance facets, such as pitch bend and control change.

The first synthesizer to be produced with MIDI was the *Prophet 600* which rolled off the assembly line during December of 1982. In January of 1983 at the NAMM convention (National Association of Music Merchants), the *Prophet 600* was connected to a *Roland JP-6*, and MIDI music was made.

In August of 1983, MIDI 1.0 specification was defined, and while certain aspects of the interface continue to change and to grow, **MIDI is here to stay!**

MESSAGE TRANSMISSION

THE LETTERS OF THE LANGUAGE

MIDI communication data is sent from one instrument to another in a steady stream of bits. You may already know that a bit is a single on or off command. Because the MIDI data is really being sent to a microprocessor, and this processor is a computer, the data must be in a form that computers understand. Computers are really very stupid. They only understand two "letters": on and off. These two letters actually represent the flow of an electrical current which is either off or on.

They receive these on and off messages as the numbers 0 (current on) and 1 (current off). What makes computers seem like they're smart is the fact that they can send, receive, and manipulate these number-letters at an incredibly fast rate. The word "bit" actually stands for **bi**nary digi**ts**. These binary digits of 0 and 1 are the two "letters" that are hooked together to form the "words" of MIDI. The entire MIDI language only uses these two letters.

THE WORDS OF THE LANGUAGE

If the MIDI language was like our verbal language, then a word could be made up of any combination of letters. Some words might use only one letter while other words could use as many as twelve or more. Computers are not quite like humans. They need to know (in advance) how many letters are going to be used in each word. Not only that, but every word must have the same number of letters; no more and no less.

If the MIDI language used words that were made up of two bits, then there would only be four different words (2^2) in the vocabulary. These words would be the result of all the different combinations that are possible using those two bits. The different words would be 00, 01, 10, and 11. If MIDI words contained three bits, then the language would only have eight (2^3) words: 000, 001, 010, 011, 100, 101, 110, and 111. These groupings of bits, that together form a complete word are called a "byte".

In MIDI, a byte is defined as a group of eight bits. These eight bits can form the numbers between 0000-0000 and 1111-1111 (the numbers here are separated for ease in reading). Using the same formula as we did above, you can discover that there are 256 possible words (2^8) in the language of MIDI. The Byte Chart on Page 122 lists all of the possible MIDI words using two-bit to eight-bit bytes.

If you were going to create a language to describe music in terms of numbers, you would most likely want to have more than 256 different words at your disposal. The number 256 is fairly large if you're talking about different ways to cook hamburger or people that your next door neighbor invited over for a few beers. But, it's not a large number in terms of words that are going to be used to express your musical ideas. Let's face it, we would use up 88 of those words just by playing all the different keys on a piano keyboard.

A solution to this problem was found that would increase the number of possible commands in a very dramatic way. While there can only be 256 distinct words, we can use more than one word to describe a complete musical event. In fact, with MIDI, complete events (or complete instructions) are anywhere from one to three or even more bytes in length. In order to pull this off, the first binary digit (bit) in the series has been designated as an identification bit. This bit is used to determine whether the MIDI word is a **Status** or **Data** byte.

STATUS AND DATA BYTES

As I just said, complete MIDI events are the result of one or more separate bytes (words). Just how many bytes will be required for the complete event is something that has been figured out in advance and programmed into the instrument's microprocessor.

Status Bytes — Status bytes are MIDI words that begin with the number "1". It is always the first part of a complete MIDI event. A status byte is a byte that is going to tell the microprocessor which parameter is going to be affected. In other words, the status byte tells the receiving synthesizer how to interpret the other MIDI words that will follow. Depending on which status byte is being used, it also lets the receiving instrument know how many more bytes to expect before the MIDI event is complete. Some examples of status bytes are Note-On and Note-Off commands, Program Change commands, and Mode Message commands. All of these will be explained in detail in the next part of this book.

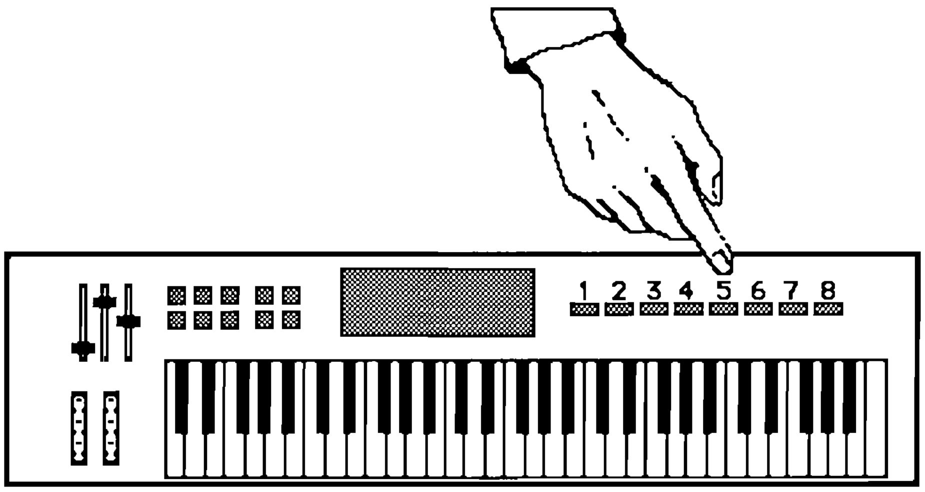
Data Bytes — Data Bytes are MIDI words that begin with the number "0". These bytes tell the microprocessor to what degree or level a certain parameter is going to be set. The specific parameter has already been sent to the instrument in the form of the status byte. Now the data bytes that follow will tell the instrument the rest of the information to complete the MIDI event.

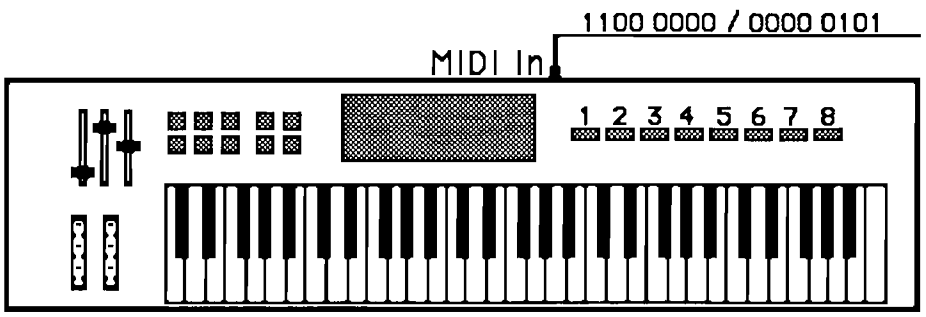
Let's take a look at how this might be done. For this example (See Example Number 1), let's say that you have a synthesizer which selects its different sounds by hitting different buttons on the front panel. If you want to tell the microprocessor of the synth to play a flute sound (button five), you simply punch the fifth button and the synthesizer will play that sound. If you want to tell the synth to play that same sound through MIDI, the action is a little different.

First, you need a command that tells the microprocessor that you want to select the parameter of "Program Change" (in MIDI terms, the different sounds that synthesizers make are called programs). Then, you need to tell the microprocessor that you want number five. The command that tells the synth to change the program is a status byte, because it is selecting a certain parameter. The command that tells the synth to select the sound in program number five is a data byte, because it is selecting a level or a degree of that particular parameter. Because this combination of status and data bytes are

Example No. 1

Without MIDI, you simply push button number five on the front panel.





Using MIDI, commands come in through the MIDI cable. First the command for program change, then the command for number five.

required to perform that particular action, the complete MIDI event consisted of two distinct words: one status byte and one data byte.

How does all of this status and data stuff increase the possible number words that are available with an eight-bit byte? It doesn't! In fact, since the first bit of our word has been taken up by the designation of the status or data classification, we have fewer words. If the first number determines that it is a data byte, then the remaining seven numbers only allow for 128 distinctly different words (now only 2^7). But, these words are data bytes for 128 different status commands.

Let's look at this in a different way. If we have a 128 different data byte possibilities and 128 different status possibilities, then we can determine up to 128 different settings of 128 different things...128 different notes, 128 different program sounds to call up at any one time, 128 different levels of volume (velocity), 128 different levels of an LFO (Low Frequency Oscillator), and on and on. If you do a little math, you will find that there now are 16, 384 distinct MIDI words. This is a vast increase over the original 256 MIDI words!

The trick is that even though we have fewer words, complete MIDI events are made up of a *combination* of words. Let's look at it in still another way. Let's say our language has 128 words for animals: bird, dog, cat, horse, mosquito, etc. and 128 descriptive words: large, short, blue, funny, etc. We can become much more descriptive by combining the words in different ways. I think that by now you should be getting the picture.

In reality, it doesn't work quite this way. Even if you stayed up all night long, you most likely couldn't think of 128 parameters that you wanted to control. After all, those are a lot of parameters!! The people who put together the MIDI specification couldn't think of 128 either, so they used some of the bits in a status byte to designate other information.

HOW MANY ARE THERE — 127 OR 128?

Why is there a difference between what I hear from different sources? Some people say that there are 128 different programs or patches to MIDI, but the highest number available is 127.

It sounds like you went to school at about the same time that I did. When I was learning how to count, the first number was "1". If you start to count a group of apples, you begin with "1"! Computers, however, are different, in that their first number is "0". Remember, to a computer, "0" is the first legal number. They only work with the two numbers of "0" and "1". In base two, the first apple you count is really the number "0", then comes number "1".

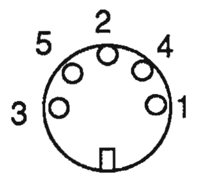
Sometimes you will find different people talking about MIDI and hear both of the numbers of 127 and 128. Keep in mind that there are 128 different data numbers. So there can be 128 different patches or program changes, velocity levels, or anything else. But they are numbered from 0 to 127. See Page 118 at the back of this book for a conversion chart for the various types of numbering systems used.

MESSAGE MECHANICS

THE MIDI CABLE

The connection between the two microprocessors in different synths is made with a cable. The cable specified for MIDI is a five pin **DIN** plug similar to the type of connection you may have on the back of a foreign made tape deck or small personal computer. The DIN connector is relatively inexpensive to produce and to buy, especially when you compare it to another popular type of connection, the XLR or Canon plug. DIN cables are much more reliable than the familiar guitar plugs with a one-quarter inch phone jack on each end. Those wouldn't be of much use anyway, as MIDI requires at least three conductors. By using a five pin DIN connector, there are two unused, spare pins that might be put into use for some now unkown, future developments of MIDI. The MIDI specification calls for all instruments to be equipped with female DIN plugs. This means that a standard MIDI cable has two male ends (the ones with the pins), so that either end can be plugged into any MIDI instrument.

Example No. 2



The five pin **DIN** plug. Pins four and five carry the MIDI signal while pin two is the ground.

SERIAL COMMUNICATION

One of the important things about MIDI, is the style or type of communication that is used. MIDI transmits and receives in a communication protocol called "serial" transmission. Serial communication can be defined as a type of communication in which the data stream moves down the cable one bit at a time. Yes, I said *one bit!* Because these bits are in a steady stream, a couple of additional bits are required.

Each MIDI word also contains a **Start Bit** and a **Stop Bit**. These serve to separate one MIDI word from another. These bits tell the synthesizer something like: "here comes a MIDI byte" and "that was all of it". If a word in the MIDI language is eight bits long, combined with its start and stop bit, then a complete MIDI command is ten bits in length.

The other popular type of communication (not used in MIDI) is called Parallel. In Parallel communications, the eight bits that make up the MIDI command would move down eight separate (yet, connected) cables at the same time. Parallel cables are sometimes called "ribbon" cables. I'm sure that some of you have seen this kind of cable before. Some computers use parallel cables to connect with their printers. While this type of communication might be faster, parallel cables cost more money to manufacture, and would naturally cost more for the end user to buy.

To make matters worse, these MIDI standards are not any type of law. Manufacturers are free to do whatever they want. But, the result may be an instrument which is incompatible with the rest of the MIDI world.

The only true way to learn what any MIDI device can or can't do is to read the MIDI implementation chart that should be included in the owner's manual of any MIDI instrument. Later in this book, you will learn how to read and interpret an implementation chart. But, right now, let's go look at channel voice messages.

As mentioned earlier, channel voice messages are one of the two types of MIDI messages that are sent over specific channels (the other type of channel messages are called channel **mode** messages which will be discussed in just a little bit). Channel messages are those messages that contain a four-bit channel designation in their status byte. These four bits specify which of the sixteen MIDI channels will carry the command. Any instrument in a MIDI system that is listening to that specific MIDI channel will respond to this message.

Example No. 5

Bit Assignments for Channel Commands

1 101 0001

This bit designates that the command is a status byte.

These bits designate the particular channel command (in this example, channel pressure).

These four bits designate the particular MIDI channel (in this example, channel 2).

Channel Bits	Channel Number	Channel Bits	Channel Number	Channel Bits	Channel Number	Channel Bits	Channel Number
0000	1	0100	5	1000	9	1100	13
0001	2	0101	6	1001	10	1101	14
0010	3	0110	7	1010	11	1110	15
0011	4	0111	8	1011	12	1111	16

Example Number 5 shows just how these channels are specified in the status byte. The first number on the left of the example is the bit "1". This is the bit that determines the byte's function as a status byte. The next three numbers (101) indicates the particular status command. These last four bits are the ones that determine the channel. The chart at the bottom of Example Number 5 shows the various combinations of bits that indicate all sixteen channels.

Because there are now only three bits that can be used to determine the different commands (one bit required for the status designation itself and four additional bits for the channel designation), there are only eight possible combinations. Seven of these are used for channel voice messages, and they include the command for note on, note off, polyphonic key pressure, channel pressure, program change, control change, and pitch wheel change.

NOTE ON — 1001 nnnn

The note on command is really one of the most important MIDI commands. All MIDI instruments that perform with pitches and musical sounds, must be able to respond to some sort of note on command. The "n" letters in the Status Byte here refer to the four-bit combination that determines the channel (1-16) that carries this command.

A note on command takes three pieces of information in order to form a complete MIDI action. The first piece of information that is required is the status byte that says, "note on". But the synthesizer also needs a couple of additional data bytes that are going to tell it which note to turn on and how loud to turn it on.

The first data byte that is transmitted after the status byte is a MIDI number that determines the pitch. Note numbers in MIDI run from 0-127, with 0 being the lowest and 127 being the highest. Using this numbering system, the pitch "middle C" is at MIDI note number 60. The distance between one number and another is a measurement of half-steps. See Appendix Page 117 for a full listing of all the MIDI note numbers.

After the pitch has been designated by MIDI, the synth requires an additional piece of information for velocity (See the sidebar for the differences between velocity and volume). Velocities also run from 0-127, with 0 being no velocity and 127 being the maximum velocity. A velocity data level of "0" can have an additional meaning as well. See the sidebar called "Running Status" for an explanation of how this works.

VELOCITY AND VOLUME

Most often, when people talk about keyboards or other electronic instruments being "velocity sensitive" or "sending velocity", they are talking about their ability to play dynamics (different volumes). Instruments that are not velocity sensitive only play at one dynamic no matter how hard or soft the keys are played. MIDI instruments which do not respond to changes in velocity usually send data byte 64 as the velocity value.

Is there a difference between volume and velocity? The answer is that sometimes there is a difference and sometimes there isn't. Synthesizers that can read velocity really *do* measure velocity, or the speed at which a key moves from not being pressed (key-up position), to being pressed (key-down position). If this rate of velocity has been programmed to control the VCA (voltage controlled amplifier), then the faster the key moves down, the louder that note can play. In this case, the difference between velocity and volume is minimal. Most inexpensive synths, if they respond to velocity at all, work this way. On this type of instrument, you can get a very loud sound just by "flicking" the key with your finger — almost like a finger snap. What the keyboard is actually measuring is not how hard the key has been pressed, but how fast it went down.

In another case, however, the amount of velocity might be programmed to affect another musical parameter instead of amplitude or volume. What if the different velocity rates were fed into the part of the synthesizer that determined whether the sound was going to come out of the left or right speaker? If the programmer set up something like, "all notes with a velocity of 88 and below should come out of the left side, and all notes with a velocity of 89 and above should come out of the right side", then the velocity level is going to determine the sound's placement within the stereo field. Now there is a *big* difference between velocity and volume.

Some instruments can assign the velocity level to many different parameters, or even to more than one parameter at a time. For instance, the level of velocity could control dynamics, stereo placement, tonal quality (using different levels of filters), and pitch (higher velocities cause higher pitches than lower velocities) all at the same time. Of course, there are those trade-offs again. Instruments that will let you control all of these different parameters with velocity are going to be more expensive than those instruments that only let the velocity control the volume.

NOTE OFF — 1000 nnnn

The note off command is the other most important MIDI command. The note off message does pretty much what you think it does. When you let the key back up, the synth sends a note off command. A complete note off command, just like the note on, requires three bytes. The status byte that tells the synth that it is going to turn a note off, and two data bytes.

The first date byte that follows a note off status command is the MIDI note number of the key that is going to be turned off. The second date byte following the status byte is the note off velocity. Not all synthesizers respond to a note off velocity, but this piece of data is just the reverse of the note on velocity. It is a reading of how fast you move the key from the key down position back to the key up position. This data byte can allow your style of playing to affect the duration of a note's release (slow or fast decay).

Example No. 6

CHANNEL VOICE MESSAGES

Status Byte	Data Byte	Data Byte
Note Off 1000 nnnn	Note Number	Velocity Value
Note On 1001 nnnn	Note Number	Velocity Value
Poly Key Pressure 1010 nnnn	Note Number	Pressure Value
Control Change 1011 nnnn	Control Number	MSB/LSB Value
Program Change 1100 nnnn	Program Number	None Sent
Mono Pressure 1101 nnnn	Pressure Value	None Sent
Pitch Wheel Change 1110 nnnn	LSB Value	MSB Value

POLYPHONIC KEY PRESSURE — 1010 nnnn

Polyphonic key pressure is one of the forms of "aftertouch". Aftertouch is a word that describes the synthesizer's ability to control a sound *after* the key has been pressed down and *before* the key is released (keep in mind that the word key does not just mean a piano-type key). Just like note on and note off commands, polyphonic key pressure requires two additional data bytes. The first will signify the proper note that is going to receive the key pressure, and the second data byte will signify the level or amount of pressure (0 for none up to 127 for maximum).

To understand aftertouch, let's think for a minute about a piano and how it makes its sound. When you push down a piano's key, a hammer strikes the string making it vibrate, and the sound is made.

Example No. 9

CHANNEL MODE MESSAGES

Status Byte	**Data Byte**	**Data Byte**
Control Change 1011 nnnn	Local Control 0111 1010	On/Off Value
Control Change 1011 nnnn	All Notes Off 0111 1011	Dummy Byte 0000 0000
Control Change 1011 nnnn	Omni Off 0111 1100	Dummy Byte 0000 0000
Control Change 1011 nnnn	Omni On 0111 1101	Dummy Byte 0000 0000
Control Change 1011 nnnn	Mono On 0111 1110	Number of Mono Channels
Control Change 1011 nnnn	Poly On 0111 1111	Dummy Byte 0000 0000

ALL NOTES OFF — 0111 1011

The all notes off command is designated by the first data byte of 123. The second data byte is always the number 0, and really doesn't give the synth any more information. It is required because all controller commands must have three bytes. This just makes things more consistent.

The all notes off command tells all instruments listening to that channel to turn off all its notes. This command is used quite often by sequencers. If you are playing a song with your sequencer, and you stop right in the middle of a note, the synth may not receive a corresponding note off command, and you might end up with a stuck note. Most sequencers will send an all notes off command when you tell the sequencer to stop.

The following four mode messages will cause a synth to perform an all notes off. This allows the instrument to begin in a new mode without the worry of stuck notes.

MIDI MODE SELECTIONS

There are four different MIDI modes that can be accessed by a master controller as well as from the front panel of various MIDI devices. (See pages 13-15 for a review of these four modes). The first data byte after the status byte is going to select one of the following mode commands listed below. These mode messages are received over the basic channel of the MIDI device.

It is possible for a single MIDI instrument to have more than one basic channel. As an example, an eight voice poly-timbral synth may be programmed to act as if it is split into two separate four-voice instruments. When this is the case, each four-voice instrument may be instructed to listen to its own basic channel.

Omni Off — 0111 1100

This is selected by the data byte of 124. The second data byte is always 0. Again, the zero value just serves the purpose of having the status byte followed by two data bytes. When this command is given to a synthesizer, it is asked to turn off its omni setting. While the omni mode is turned off, the receiving synthesizer will only listen to and act upon commands that are directed toward a single MIDI channel. This is referred to as the "basic channel". Instruments that receive this command also respond by turning off any notes that it is currently playing.

Omni On — 0111 1101

This is selected by the data byte of 125. The second data byte is 0. When instruments receive this command, they respond by turning on the omni setting. In this mode, synthesizers can listen to and act upon MIDI data coming in on all 16 MIDI channels at one time. This command also asks the receiving synth to cancel all its notes.

Mono On — 0111 1110

This is selected by the data byte of 126. This command also asks the receiving device to cancel its notes. Instruments that receive this command respond by changing into a mono type of instrument. In other words, that instrument can only sound one note at a time while it is in a mono on setting. The second data byte for mono on does send additional information. It takes the form of 0000 nnnn. In this case the "n" numbers refer to how many channels the device will listen to. This is a little tricky, so let's take a closer look at it.

The MIDI specification allows for a single MIDI instrument to listen to several different channels while it is in this mode. Remember that each channel will still only be in a monophonic setting. How does the receiving instrument know how many channels to listen to? The number of channels that is designated in the second data byte tells it!

If the basic channel is set to MIDI channel 2, and the second data byte instructs the synth to listen to four channels, then the synth will be listening to channels 2, 3, 4, and 5. If the basic channel is MIDI channel 7, and the second data byte says to listen to six channels, the instrument will be listening to channels 7, 8, 9, 10, 11, and 12 If the value of the second data byte is 0, then this command allows the receiving instrument to listen to all MIDI channels (with one voice per channel), from its basic channel on up to channel 16.

Poly On — 0111 1111

This is selected by the data byte of 127. Again, the second data byte is 0. MIDI devices that receive this command are asked to turn on their poly receiving mode. Poly and mono commands are mutually exclusive. This means that selecting poly on will deselect mono on, and vise versa. In poly mode, instruments can respond in a polyphonic manner (the ability to play more than one note at a time). This command also asks the receiving instrument to cancel all its notes that are currently sounding.

SYSTEM COMMON MESSAGES

System messages differ from channel messages in one major aspect. System common messages *are not* channel specific. Because they are not intended for any one channel, they carry no channel information. All system messages have a structure of 1111 nnnn.

You might remember that channel messages used three bits to indicate their function. The three bits could be combined to produce eight different words, but, channel commands use only seven of them. This eighth word (111) is the indication for a system message. In this particular case, the "n" numbers following 1111 specify the particular system message.

System common messages are either one, two, or three bytes long, depending on just what the command happens to be. These commands apply to all channels, and all instruments in the system will react to them in the same manner. Again, all MIDI devices will not send or respond to every one of these messages. As your MIDI system becomes more complex, these system common messages become more and more important.

Example No. 10

SYSTEM COMMON MESSAGES

Status Byte	Data Byte	Data Byte
Quarter Frame 1111 0001	Message Type/ Time Frame	None Sent
Song Position Pointer 1111 0010	LSB Value	MSB Value
Song Select 1111 0011	Song Number	None Sent
Tune Request 1111 0110	None Sent	None Sent
EOX 1111 0111	None Sent	None Sent

QUARTER FRAME — 1111 0001

In MIDI Specification 1.0, this status byte (1111 0001) was an undefined system common command. With the latest MIDI Specification update, this byte has a new meaning.

The quarter frame message is used in conjunction with the MIDI Time Code. For this reason, we will wait to discuss the quarter frame message. Please look at the section concerning MIDI Time Code on page 65 for a full description of this command.

SONG POSITION POINTER 1111 0010

The song position pointer status command is followed by two data bytes. The first data byte is the LSB, and the second is the MSB. These two bytes are combined to form a high resolution value.

Song Position Pointer performs a fairly simple task. It can command a slave drum machine or sequencer to begin playing (or continue playing) from a specific point in time. This is really a wonderful feature that can save a lot of time.

Imagine that you want to change something that happens

at least two samples of a sound in order to represent its wave. This seems to make sense, as it would require, at the very least, a positive and a negative reading. The highest note on a piano is 4186 Hz (cycles per second). In order to represent this pitch, there would have to be a minimum of 8362 samples per second. That would give us the sound, but what about any overtones that might be present? If the upper range of human hearing is about 20,000 Hz, then the minimum sample rate should be 40,000 samples per second. As a reference, a compact disk has been sampled with a 16-bit A/D converter (65,536 numbers) at 44,100 samples per second (up to 22,050 Hz).

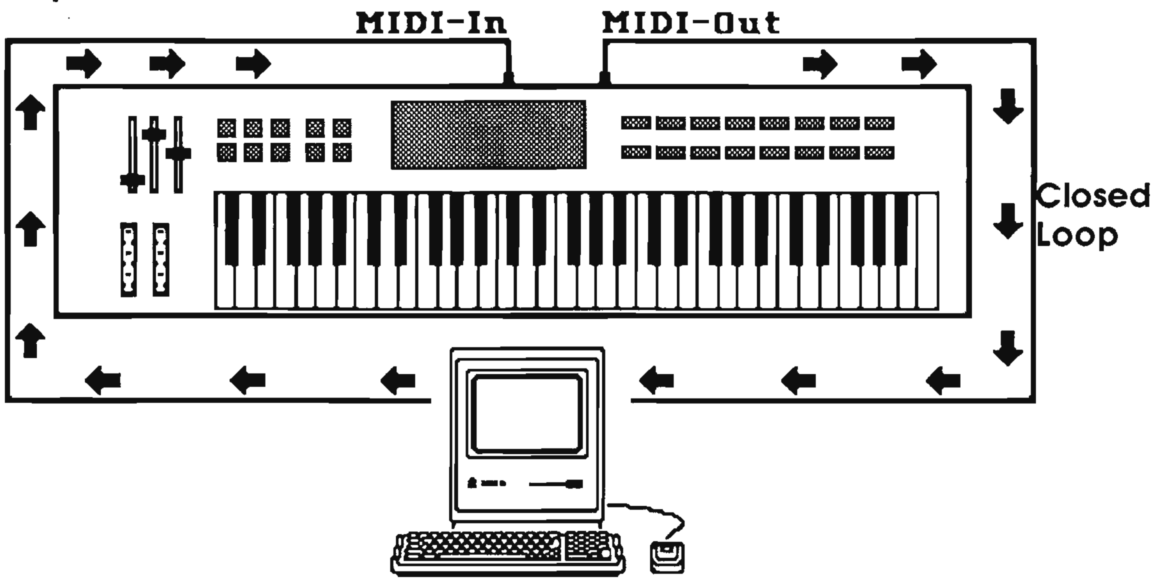
The big question is "Is it really the original analog sound?" The real answer is "No, of course not". But, if you get a high quality sampler, the sound is as close to the original as the recorded sound of a compact disk.

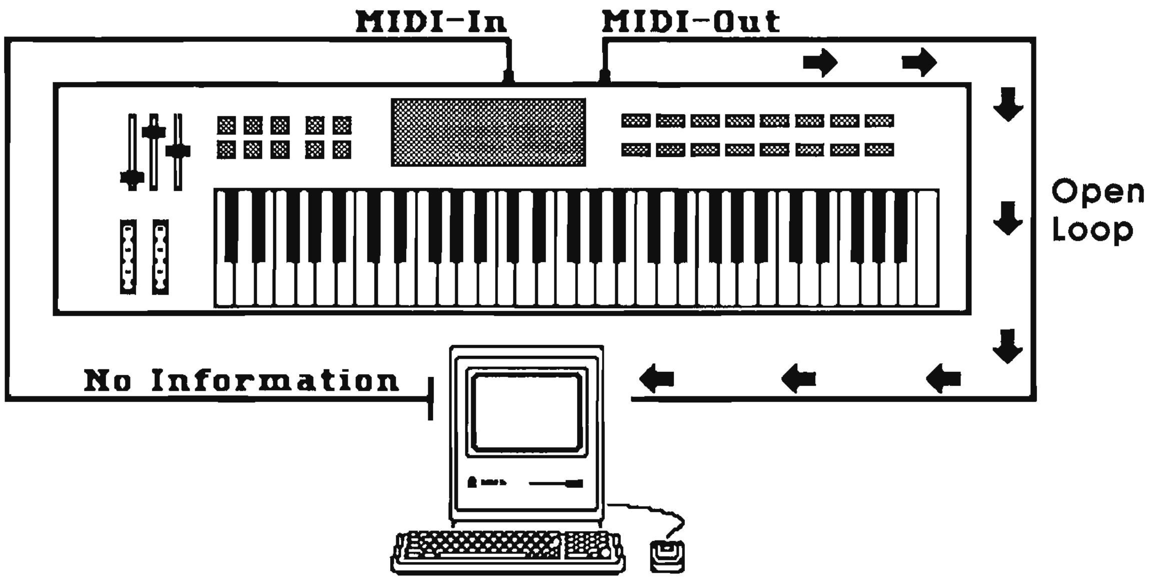
other sampler. In theory, this could be done even if the samplers were different brands of instruments. Different brands can't talk to each other with normal system exclusive commands because of the manufacturer's I.D. number. This protocol of sending and receiving sample information is called the "Sample Dump Standard". Just like MIDI itself, these commands are simply more bytes that have been defined and approved. To put it another way, we are adding more words to the MIDI language by using more combinations of bytes.

Now that we've got that detail taken care of, let's move on to the commands. Universal non-real-time system exclusive messages can work in two different ways. They can pass information either in a closed loop, or an open loop. Example Number 13 shows the difference between the two loop types. In an open loop, the sending unit (often called the "source") sends the message and gets no response from the receiving unit (often called the "destination"). Communications in a closed loop system, also called "handshaking", go in both directions. The destination unit sends its own information back to the source. In other words, they are constantly talking to one another; a little conversation that goes on between the two microprocessors. In Example Number 13, the destination device is a computer.

There is one more term that we should define before we continue with the explanation of the Sample Dump Standard. There is a period of time in which nothing happens in the source device. This is sort of a waiting period during which the source device waits for the destination device's answer. This time period is called "time out". If the source device hears an answer from the destination device, it will process the response and do whatever the destination device asks. If the source device doesn't hear anything during this time out, then it assumes that there is an open loop configuration, and will continue with the sample dump procedure. Some MIDI devices will require the closed loop in order to operate, others don't. As an example, the E-Mu Systems *SP-12* drum machine will respond with a "No Disk

Example No. 13





Closed Loop — Information moves in both directions: from the synth to the computer and from the computer back to the synth.

Open Loop — Information moves in only one direction: from the synth to the computer. The computer sends no data back to the synth.

Connected" message if it finds an open loop and will go no further.

The Sample Dump Standard consists of an exact series of messages (see Example Number 14). These messages, of course, begin with the system exclusive command of 1111 0000, followed by the I.D. number for universal non-real-time commands (126 or 7E in hex format). Next follows a byte which designates the device channel. This is not to be confused with the normal MIDI *voice* channels. There are 16 MIDI *voice* channels, but there are 128 different *device* channels. Using this type of system, a single command unit (most likely a computer) could handle all the non-real-time messages for 127 different MIDI devices. If the number in this position is 127 (7FH), then the message is intended for all devices in the system.

Following the device channel byte, there is a "Sub-I.D." byte which specifies the exact message type. Currently, there are only eight defined Sub-ID numbers, and we will get into those in just a moment. Next follows the data. There can be any number of data bytes contained in this section depending upon the message type. Last, but certainly not least, is the EOX command of F7H.

Before we take a look at how two devices might communicate with each other using non-real time commands, let's talk about the eight different Sub-I.D. numbers that have already been defined.

Example No. 14

HEX	**BINARY**	**DEFINITION**
F0	1111 0000	System Exclusive Status Byte
7E	0111 1110	I.D. For Non-Real Time
00-7F	0??? ????	Device Channel Number
00-7F	0??? ????	Sub-I.D.
00-7F	0??? ????	Series Of Data Bytes
F7	1111 0111	EOX (End of Exclusive) Status Byte

SAMPLE DUMP HEADER — 0000 0001

The sample dump header (See Example Number 15) is sent one time at the beginning of the sample dump. This header serves one main function. It tells the receiving device everything it needs to know about the data that is going to follow. By using this header at the start of the sample dump, the receiving device can determine if it has enough memory to accept the dump, and just how to interpret all of the information about the sample. Let's break this thing apart, analyze it, and see what makes it tick.

FO, 7E — By now, you know that these two commands are the system exclusive status byte and the I.D. for universal non-real-time system exclusive commands.

00-7F — Again, this is the device channel number, and is required for all non-real-time commands. It can specify any of 127 different devices that might be hooked up to the system. Look back at Example Number 14, and you will see that this data byte always follows the 7E data byte. If the number 127 is designated for this byte, then these messages are sent to all devices in the system.

01 — This is the Sub-I.D. number which specifies the sample dump header.

— These two bytes are the indication of the sample number. Most samplers allow you to sample more than one sound, and this command specifies which one of those samples you are going to work with. Because this is a two-byte command, it has high resolution, and can handle up to 16,384 different samples.

bb — This stands for "significant" bits, and can be any number from 08H to 1CH. Sampling machines use different "resolution" rates to measure the sounds that they are going to play back. This measurement of how detailed the sound is going to be, is very important to the quality of sound. A greater number of values means that the digital sound can be a more accurate representation of the original analog sound.

Some machines use an eight-bit format, and can measure sounds by assigning any certain point in time, any value between 0 and 255. Twelve-bit machines can assign values from 0 to 4095, and sixteen-bit machines can work with numbers values from 0 to 65,535. If the machine was working with an eight-bit resolution, then the number at this position would be 08H (0000 1000). A sample that has sixteen-bit resolution would have 10H in this position (0001 0000). The value of 1CH would be used for a sample that had 28-bit resolution.

pp — This is an abbreviation for the sample period. We've just talked about the "resolution" of the sample, and this number is an indication of how fast the measurements were taken. Samplers call

Example No. 15

SAMPLE DUMP HEADER — 0000 0001

HEX	BINARY	DEFINITION
F0	1111 0000	System Exclusive Status Byte
7E	0111 1110	I.D. For Non-Real Time
00-7F	0??? ????	Device Channel Number
01	0000 0001	Sub-I.D. For Sample Dump Header
00-7F		Series Of Data Bytes
##	0??? ????	Sample Number LSB
##	0??? ????	Sample Number MSB
bb	000? ????	Significant Bits
pp	0??? ????	Sample Period LSB
pp	0??? ????	Sample Period NSB
pp	0??? ????	Sample Period MSB
ll	0??? ????	Sample Length LSB
ll	0??? ????	Sample Length NSB
ll	0??? ????	Sample Length MSB
ss	0??? ????	Sustain Loop Start Point LSB
ss	0??? ????	Sustain Loop Start Point NSB
ss	0??? ????	Sustain Loop Start Point MSB
ee	0??? ????	Sustain Loop End Point LSB
ee	0??? ????	Sustain Loop End Point NSB
ee	0??? ????	Sustain Loop End Point MSB
tt	0000 000?	Loop Type
F7	1111 0111	EOX (End of Exclusive) Status Byte

WHAT ARE ALL THOSE FUNNY LETTERS?

Sometimes, when I'm reading information about MIDI (especially when I'm trying to get through some company's system exclusive stuff), I see all these funny numbers that don't really seem to mean anything. What gives? I'm very confused!!

Whenever you see things that look like "##", "yy", "pp", "xx", or any other type of weird letters, don't get scared! These are abbreviations, and are just another way of saying that the information included at this point is a variable. I can't tell you the numbers that go into these slots because it varies. The people that designed the machines can't tell you either. Just keep in mind that somewhere in the information, there must be an explanation of just what those weird letters mean.

Often, these letters may be mnemonic to help you remember what they are. The sample start point might look like "ss" and "pp" may stand for program number. Other companies sometimes go straight down the letters of the alphabet like: "jj", "kk", "ll", "mm", and so on.

this the "sample rate". A sampler with a rate of 10,000 samples per second, will take 10,000 sound measurements in a single second. The Sample Dump Standard needs three bytes to convey this information. If you remember back to high resolution data, you know that two data bytes can be combined to create a fourteen-bit word (don't forget that we lost two bits of resolution because each of the data bytes must begin with the number "0".) A fourteen-bit word can only express values between 0 and 16,383. What do we do when there is a sample rate of 44,100 samples per second? This number can't be expressed by using a fourteen-bit byte. By using a three-byte word, we end up with 21-bit resolution. In Example Number 15, you can see that the LSB is sent first, the "NSB" stands for "next significant byte", and the MSB is sent last. Pulling up the ol' pocket calculator, twenty-one bits can express any value between 0 and 2,097,151. Wow, that's a fast sample rate!

ll — The three data bytes in this position are the sample length in words. "In words" means that if the sample resolution was twelve-bit, then one word would be twelve bits long. A sampler that works at 32,000 samples per second would send 32,000 "words" of information for each second of sound. A sample that was three seconds in length would contain 96,000 words. Again, in order to express larger numbers, three bytes are combined.

ss — These three data bytes are combined to express the sustain loop's starting position. This is the sample number, *in words*, where the sample loop point begins.

ee — These three data bytes indicate the sustain loop's end point. Again, the end point is expressed in words.

tt — Some samplers can play their loops in different ways. This is an indication of the loop type. If the number 00H is in this position (0000 0000), then the loop is forward. If this number is 01H (0000 0001), then the loop is backward. More loop types will probably be defined in the future.

F7 — EOX status byte.

SAMPLE DUMP DATA PACKET — 0000 0010

The sample dump data packet (See Example Number 16) contains the actual sample information. After receiving the sample dump header, the destination device now knows how to interpret this flow of information. Each packet consists of 127 bytes. This way, some devices that don't have a lot of memory can receive the data packet, process it, and then ask for more data.

F0, 7E, 00-7F — These are the same three bytes that are used for all non-real-time commands. They are the system exclusive status byte, the non-real-time Sub I.D., and the device channel number.

02 — This is the Sub-I.D. number which specifies the sample dump data packet.

cc — This number can be anything between 0 and 127. It is a running count of the data packet number. As an example, data packet number one would have 0000 0001 in this position, while data packet twenty-seven would have 0001 1011. If there are more than 128 different data packets, then this running count number starts over again with 0.

dd — This is the data itself. Each data packet consists of 120 bytes of information that can work together to form either 30, 40, or 60 different "words" (see page 54 for the definition of "words"). The exact number of words is determined by the sample format. If the sample uses eight-bit to fourteen-bit resolution, then two data bytes form a single word, and there will be 60 of these in the data packet. A sample with fifteen-bit to 21-bit resolution will require three bytes for each word (40 words), and samples that use 22-bit to 28-bit resolution will need four bytes for each word (30 words). Information contained in these bytes is left-justified, and any unused bits are filled with a "0".

xx — This is a checksum. Checksums are a form of error detection used by computers. This number is a result of the previous 124 bytes of information. When the source device sends this checksum, the receiving device figures out its own checksum. If the numbers match, then everything is OK. If the numbers don't match, then the receiving device will know that an error has occurred, and will ask the source device to send the data packet again.

F7 — This is the EOX status byte that must end every system exclusive message.

Example No. 16

SAMPLE DUMP DATA PACKET — 0000 0010

HEX	**BINARY**	**DEFINITION**
F0	1111 0000	System Exclusive Status Byte
7E	0111 1110	I.D. For Non-Real Time
00-7F	0??? ????	Device Channel Number
02	0000 0010	Sub-I.D. For Sample Dump Packet
00-7F		Series Of Data Bytes
cc	0??? ????	Packet Count
dd	0??? ????	Sample Data (120 bytes)
xx	0??? ????	Checksum of previous 124 bytes
F7	1111 0111	EOX (End of Exclusive) Status Byte

Example No. 17

SAMPLE DUMP REQUEST — 0000 0011

HEX	BINARY	DEFINITION
F0	1111 0000	System Exclusive Status Byte
7E	0111 1110	I.D. For Non-Real Time
00-7F	0??? ????	Device Channel Number
03	0000 0010	Sub-I.D. For Sample Dump Request
00-7F		Series Of Data Bytes
##	0??? ????	Sample Number LSB
##	0??? ????	Sample Number MSB
F7	1111 0111	EOX (End of Exclusive) Status Byte

SAMPLE DUMP REQUEST — 0000 0011

The sample dump request simply asks the receiving device to dump the sample. Please take a look at Example Number 17.

FO, 7E, 00-7F — These are the system exclusive status byte, the non-real-time Sub-I.D., and the device channel number.

03 — The Sub-I.D. number for a sample dump request is 03H.

— This is the sample number that is being requested. After receiving this command, the sampler will check to see if this sample number is valid. If it is, it will begin the sample dump. If it isn't, then the message is ignored. Because this is a high resolution number, it can specify any of 16,384 different samples.

F7 — EOX status byte.

SET-UP — 0000 0100

Even though the set-up command is a type of universal non-real-time system exclusive message, it actually comes into play when dealing with the MIDI Time Code. Since we're discussing the Sample Dump Standard right now, we won't get into this until page 68.

IS THE SAMPLE DUMP STANDARD READY FOR THE FUTURE?

The answer is YES! A compact disk, which is currently considered "state-of-the-art", uses a sixteen-bit resolution and a sample rate of 44,100 samples per second. The highest sample rates that I know of are those used in scientific and medical circles, and they only sample at 100,000 samples per second.

The Sample Dump Standard can handle devices that use a 28-bit resolution (268,435,456 values as compared to the 65,536 values used with the compact disk) and a sample rate of 2,097,152 samples per second. I'd say that MIDI is ready whenever the technology is!

The receiving unit gets both of these MIDI data messages and combines them into 0010 1011 (2B in hex format). By translating this binary number into decimal form (I'll do it for you), you will find that we are forty-three seconds into the time. Remember that the hours, minutes, and frames will be determined by other combinations of data bytes.

The frame nibbles are combined to form 000? ???? with the frame numbers of 0-29. The second nibbles and the minute nibbles are combined to form 00?? ????, and indicate an amount between 0-59. When hour nibbles are put together, they form 0tth hhhh. As before, the "tt" determines the SMPTE type (frame rate) using the same codes as in the full message (see page 64), while "h hhhh" indicates an hour between 0 and 23.

You might be asking yourself a very good question at this point. If it takes eight quarter-frame messages to convey the full time code, by the time the receiving device can assemble all of the various nibbles and bytes, isn't the information already two frames old? The answer is yes. But, all machines that can respond to these timing commands will keep an internal offset of "plus two frames". In other words, if a device assembles all of the information for frame number seven, it knows internally, that it should now be at frame number nine instead.

MIDI TIME CODE USER BIT MESSAGE

In our discussion of SMPTE, I mentioned that there were 80 bits included in each frame. Along with the indication of the time, there are three groups of eight-bits each that are called user bits. These can be used to form four different letters, eight different numbers, or a combination of the two. Most often, these label and identify the tape in some way, such as the time of the recording or the artist's name. These user bits do not usually change during the course of the tape, but there is a way to send this information to different devices using MIDI time code.

Example Number 23 shows the format used for sending these user bit messages through MIDI.

F0, 7F, 00-7F, 01 — These are: the system exclusive status byte, the I.D. for real-time messages, the device channel number, and the Sub I.D. for the long form time code.

01 — This is the Sub I.D. for user bit messages.

Data Bytes #1-#8 — These eight data bytes use only the four LSB digits. They are combined (much in the same way as quarter frame messages) to make four eight-bit bytes. They can be decoded in the following way: aaaa bbbb (data bytes #1 and #2), cccc dddd (data bytes #3 and #4), eeee ffff (data bytes #5 and #6), and gggg hhhh (data bytes #7 and #8).

after the release of the software. As a general rule, these universal librarians do not have the flexibility that a specific instrument's librarian would offer.

EDITORS

Just as the librarians use the system exclusive commands to send and receive bulk dumps from your MIDI device, an editor uses system exclusive commands to *program* new sounds into the device by remote control. With the aid of the computer's screen, editors can display all of the various controls of the synth at one time. This in itself is a great programming aid for the newer digital synthesizers that might use one single button to call up five or ten different programming functions. Some of the programs even use a little bit of animation to show you how the various knobs and sliders will be altered. If you have ever tried to program a sound's envelope generator from the front panel, you will quickly appreciate a computer editing program.

Again, these programs are instrument specific. An editor for one brand of instrument will not work for another brand. If you own ten different MIDI devices, you might end up owning ten different editing programs! Some editors might even contain a "random patch generator". Random generators create their own sounds from scratch by using random settings for each parameter of the synth. Sometimes, they will come up with some really great sounds. But, if you don't like the sound it created, just ask it to create another!

I think that the most amazing type of editors are the visual editors that are made for sampling keyboards. These programs let the musician actually ***see*** the waveform of any sample! Any part of the waveform can then be edited with an accuracy of under 1/40,000 of a second. Often these visual editors will include many types of digital effects, such as changing the loop points, adding different types of equalization, mixing and merging of different samples, and changing the sample's gain. Just like the envelope mentioned above, setting the sample's loop points is about ten zillion times easier when you can see it. Sure, your ears will be the ultimate judge of the loop's success, but your eyes can tell you many things that your ears might not see.

INTELLIGENT PROGRAMS

"Intelligent" programs are no smarter than other programs, but the folks who design them are brilliant! Intelligent programs use very complex mathematical formulas to help you compose music. In a way, the computer is really doing the composing, but you are setting up a certain amount of rules that it has to follow.

As an example, if you play the note "C", there are twelve other notes that can follow... another "C" or eleven others. The computer program might make up something like, " if 'C' is played, the next note will be: 'C' again - 18% of the time; 'C#' - 2% of the time; 'D' - 6% of the time" and so on. You play notes into the computer (through MIDI, of course) as the "seeds" of the composition, and the computer program makes the seeds grow into music.

As you feed your musical seeds into the computer, the program creates more music to go along with it. These programs can create additional melodies, add harmonies, invert you rhythmic material, or do any number of complex changes and additions to your original notes.

MIDI IMPLEMENTATION CHARTS

These charts are most often found in the back of the owner's manual of a MIDI device. If you know how to read them, they can tell you a great deal of information about the unit. Sometimes, information that you find or don't find in the chart can tell you whether or not the device will do what you want. Or, even if it will do what the salesman has told you it will do.

Example Number 25 shows a sample MIDI implementation chart for a simulated synthesizer called the *DAQ 750*. Please keep in mind that manufacturers really don't have any guidelines as to how this chart should look, and what it should include. Often these charts are translated into English from another language. For these reasons, just about any type of spelling, or different symbols may be used in the chart.

At the top of the chart is a header. The header often gives the name of the manufacturer, the model of the device, the version number (maybe this synth has an updated set of MIDI commands), and the date. Under the header are four columns called Function, Transmitted, Received, and Remarks. The function column describes a particular type and class of MIDI messages. The transmitted and received columns will tell you whether or not the device you're reading about will transmit or receive a particular command. The remarks column may contain additional information. Now, let's take a look at each of the functions.

Basic Channel — The default row will tell you which channel the instrument will send or respond to when it is first turned on. Instruments that can program their default channel might have "1-16" in this position with "memorized" in the remarks column. The row called "changed" will show you which MIDI channels can be assigned as the basic channel by the user.

Mode — Here is the indication of the MIDI mode when the unit is first turned on. The mode numbers are usually defined at the bottom of the chart.

Example No. 25

DAQ 750 PROGRAMMABLE HYDRO SYNTH VERSION 2.0 Date:11/87

Function		Transmitted	Received	Remarks
Basic Channel	Default	1	1	
	Changed	1-16	1-16	
Mode	Default	3	1, 2, 3, 4	Memorized
	Messges	X	O	
	Altered	XXXXXXXX	X	
Note Number		21-108	0-127	
	True Voice	XXXXXXXX	21-108	
Velocity	Note On	O v=1-127	O v=1-127	
	Note Off	X v=0	X	
Touch	Key's	O	O	
	Ch's	X	X	
Pitch Bender		O	O	7 Bit
Control Change	1	O	O	Mod Wheel
	2	O	O	Breath Control
	4	O	O	Foot Controller
	64	O	O	Sustain F. Sw.
	66	O	O	Sostenuto
	Right Wheel	O	O	Assign to 1-31
Program Change		O 0-127	O 0-127	64-127=Cartridge
	True#	XXXXXXXX	1-128	
System. Exclusive		O *1	O *1	
System Common	Song Pos	X	X	
	Song Sel	X	X	
	Tune	X	X	
System Real-Time	Clock	X	X	
	Messages	X	X	
Aux Mess.	Local On/Off	X	O	
	All Notes Off	X	O	
	Active Sense	O	O	
	Reset	X	X	
Notes	*1 = Transmit / Receive only if device number is not off.			

Mode 1: Omni On, Poly	Mode 2: Omni On, Mono	O=YES
Mode 3: Omni Off, Poly	Mode 4: Omni Off, Mono	X=NO

The "messages" row shows which of the four MIDI mode messages can be sent or received by the device. In our example, this synth can't send any MIDI mode messages, but it can receive them. The "X" is the most common indication of "No", and "O" is most common for "Yes". This code should be defined at the bottom the chart. Be careful, as some companies reverse the meaning of these symbols. The "altered" row will show you if any of the mode messages are altered inside the machine. The received column is the one to watch, as the transmitted column does not apply. Some instruments may not be able to respond to a mono on command (drum machines for instance). In this case, you might find something like "Mono On → Poly On".

Note Number — Here you will find the range of MIDI note numbers that an instrument can send and receive. In our example, you can see that the notes correspond to the range of a piano keyboard. The "true voice" row is only valid for the received column, and is only required if the true voice range is less than the received note numbers. If this is the case, notes that fall outside of the true range will be transposed up or down in order for them to sound.

Velocity — This is an indication of an instrument's ability to send and receive note on and note off velocities. In the example, note off velocities are not sent or received. The "v=0" tells you that this instrument will send a note on with a velocity of 0, as a note off command (running status).

After Touch — These rows will tell you if the instrument sends or receives polyphonic key pressure or channel pressure.

Pitch Bender — This is the indication for pitch wheel change. Sometimes the remarks column will also show you the range or the resolution of this control.

Control Change — This section of the chart lists the different control change messages that the unit can send or receive. Most often, they are listed completely by their control change number and its definition. In the example, you can see that the "right wheel" can be assigned to send or receive any control change number from 1-31.

Program Change — This column will be the most useful in determining which program change messages will call up which sounds. As you can see in the example, this instrument sends and receives all messages from 0-127. The numbers from 0 through 63 will call up the internal voices, while numbers 64 through 127 will call up the voices in the cartridge.

System Exclusive — Here is where you can find out if your unit can send or receive these messages. The remarks column or the notes at the bottom of the chart will usually give you more information about this class of messages. Notice that, in the example, the synth can send and receive these messages only if the device number is not off. The owner's manual should give you more specific information about these messages.

System Common — Because this is a synth without an on-board sequencer, it doesn't have any reason to send or respond to song position pointer or song select. Because it is a digital synth that never goes out of tune, it won't send or receive the tune request messages either.

System Real Time — If you see "O" in these rows, you know that the unit will send and receive MIDI clock messages and the real time messages of start, stop, and continue.

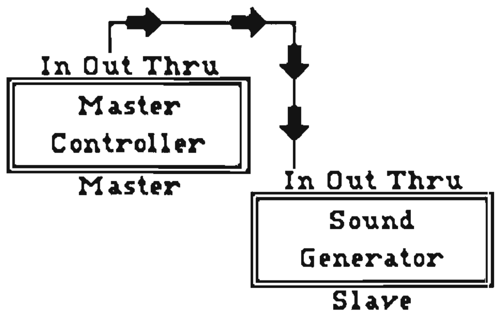
Aux Messages — These are extra messages that don't really fall into the other categories. They include local on/off, all notes off, active sensing, and system reset messages.

SIGNAL NETWORKS

Many times throughout this book, I've referred to a synth being the "master" or a synth being a "slave". A master controller an be either a dedicated input device, or any type of synthesizer that has its own input device connected (keyboard, percussion, wind, string, etc.). A slave instrument is the one that is being controlled by MIDI commands from the master instrument. The simple rule is: if you're playing two different MIDI units with one input device, then the instrument that you are physically touching is the master. The only exception to this rule is when you use a computer based sequencer to drive all the MIDI units in your system. In this case, the computer is the master device (even though you're not touching it), and all the other units are slaves.

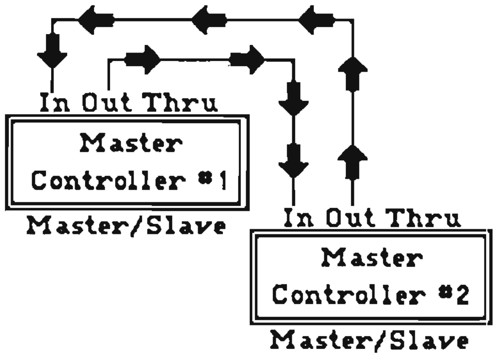
There are a variety of configurations that are possible when connecting master and slave devices. The most simple network is one which always uses the same master controller playing the same slave. Example Number 26 shows this type of configuration. This is the perfect way to connect a keyboard, wind, string, or percussion controller to a sound generator. Since the sound generator has no input device of its own, there is no reason on the planet to use it as a master controller.

Example No. 26



Another configuration is shown in Example Number 27. Here, two units (both with input devices) are connected to each other so that either can serve as master or slave. Use this set up whenever you might want both units to serve as the master controller. This is perfect if you have two different types of keyboards (weighted and non-weighted keys), and you want to use them for different types of music.

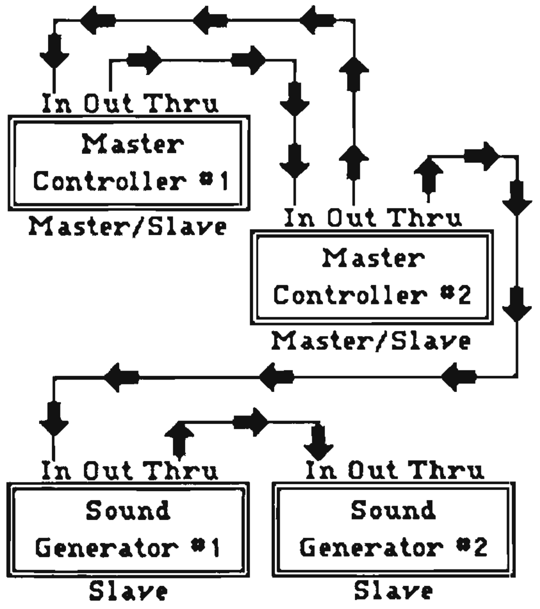
Example No. 27



In another configuration, we're using the same two master controllers, and have added two additional sound generators. Now, there are four different devices that need to be connected with each other. In Example Number 28, these are connected so that master controller #1 will control the other three devices while master controller #2 will only play the other master controller (now acting as a slave). Follow the MIDI signals down all the wires, to see what I mean. Remember that the signals coming out of the Thru ports are an exact duplicate of the signals received by the MIDI-In ports.

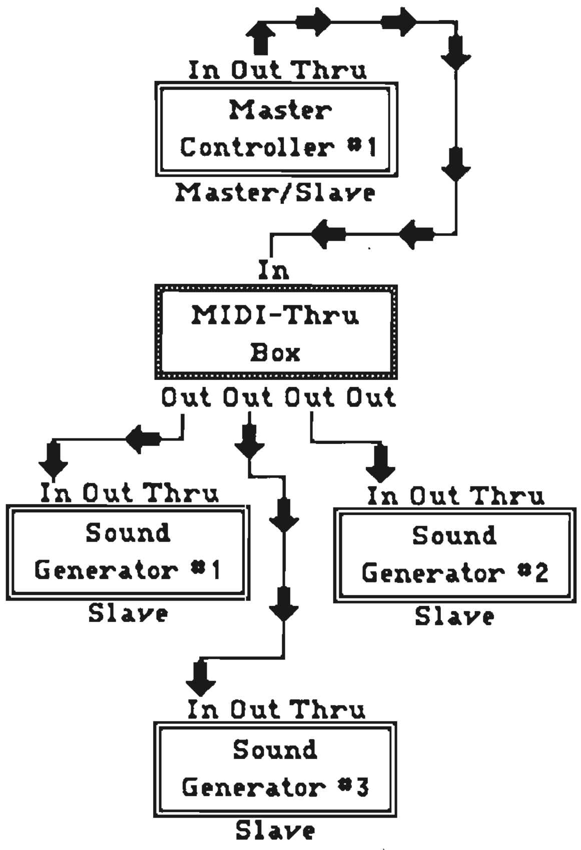
Three slave units are the maximum that can be connected in this way (sometimes called a daisy-chain). Whenever you use more than three, you may fall prey to the dreaded "MIDI delay" (see the sidebar). The solution for adding more slaves is a little unit called a "MIDI-Thru Box".

Example No. 28



In Example Number 29, we see how a MIDI-Thru box is used. The MIDI signal from the master controller is sent to the MIDI-In port of the thru box. This particular MIDI-Thru box has four outputs. Each of the outputs works just like the MIDI-Thru port of any other MIDI device. In other words, the signal coming out of each port is an exact duplicate of the MIDI-In signal. A great advantage is that you can connect more than three MIDI units without worrying about delay times. In the example, there is still room to add an additional slave such as a digital reverb unit whenever you need to expand. Some MIDI-Thru boxes can handle as many as sixteen different MIDI-In signals and route them to any or all of twenty different devices.

Example No. 29



MIDI DELAY

I've heard a lot about MIDI delay and I'm pretty confused. Just what is it?

MIDI delay can be defined as an *excessive* amount of time between sending a MIDI message and its response. Keep in mind that all MIDI synths have a certain amount of delay built into them. It may take as long as five to fifteen milliseconds from the time a key is pressed down, to the time that the synthesizer's internal circuitry plays the note. Normally, this can't be heard. Humans have a hard time distinguishing between events that happen so close together. The performer simply feels that these two events (key press and sound) happen simultaneously.

Another reason for MIDI delay might be in the way a certain MIDI system is connected. There is an internal delay of approximately three milliseconds from the time a signal is heard over the MIDI-In port and passed along to the MIDI-Thru port. If you have daisy-chained (Thru to In, Thru to In, etc.) several slaves to the master synth, then these little 3 millisecond delays can add up to the point where the delay **is** audibly noticeable. There is also a little bit of signal distortion when a message gets passed from MIDI-In to MIDI-Thru. After daisy-chaining three MIDI devices, messages may get so garbled that they can't accurately be received. The solution is to use a MIDI-Thru box for all but the most simple MIDI configurations.

From time to time, the MIDI data stream can get flooded with information. Because the MIDI data stream is sent as a series of single bits, it will take more time to send a sixteen note chord than it takes to send a single program change message. If you have a computer sequencer playing a lot of big chords, and sending a lot of other information like polyphonic aftertouch, and/or pitch wheel information to several different MIDI channels, then the data stream can simply get congested. This will also cause an audible delay between instruments.

MIDI SYSTEMS

MIDI systems can come in many different shapes and sizes. They can be simple or complex, inexpensive or expensive, include last year's models or next year's models, and come from one manufacturer or from several different companies. You can build a MIDI system just like you might build a stereo system. You can choose a self-contained system, or build it with a series of different components.

The MIDI system that you put together may not be like any other MIDI system. The devices that you choose to buy and how you wire them together will make your system unique. That is one of the things that makes MIDI so much fun. As we look at a couple of different MIDI systems, keep in mind that the master controller (shown as a keyboard in the examples), should be whatever kind that suits you the best. You may feel more comfortable using a wind, percussion, or string controller as your master input device. The sound generators can be either a dedicated unit which has no input device of its own, or the sound generator within another synthesizer.

MIDI SYSTEM #1

This system consists of a master controller and one slave. The connection is made by running a MIDI cable from the MIDI-Out of the master to the MIDI-In of the slave. That's it! What could be more simple? Now that the connection is made, let's take a look at some of the possibilities that are available to you. After you play around with these ideas, see what other neat things you can do with it!!

A. You can layer two sounds together by having the master controller play its internal sounds *and* the sounds created by the slave.

B. You can turn the layers on and off by using the local on/off MIDI message, or the volume slider on the master synth. When the volume is up, you will hear both master and slave. When you turn the volume down, you will hear only the slave.

C. If you take the time to arrange the different programs on both instruments, calling up a new patch on the master controller will call up a complementary patch on the slave.

D. If your slave device is a "polytimbral" unit (this term is used to describe those instruments that can sound more than one patch at a time), you can create different splits. As an example of a split, let's say that the lowest octave of the master controller is going to play a bass patch, the middle two

CZ-101 VOICE PARAMETERS

Voice Name________________

Voice Location________________

	LINE 1								LINE 2							
__DCO	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
_Rate																
_Level																
_Sus/End																
__DCW	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
_Rate																
_Level																
_Sus/End																
__DCA	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
_Rate																
_Level																
_Sus/End																

LINE 1	LINE 2
Wave Form 1	Wave Form 1
Wave Form 2	Wave Form 2
DCW Key Follow	DCW Key Follow
DCA Key Follow	DCA Key Follow

_LINE SELECT______	_OCTAVE______
______Line #______	______Range______
_DETUNE______	_VIBRATO______
______Plus/Minus______	______Wave Form______
______Octave______	______Delay______
______Note______	______Rate______
______Fine______	______Depth______

_MODULATION__________Ring__________Noise__________