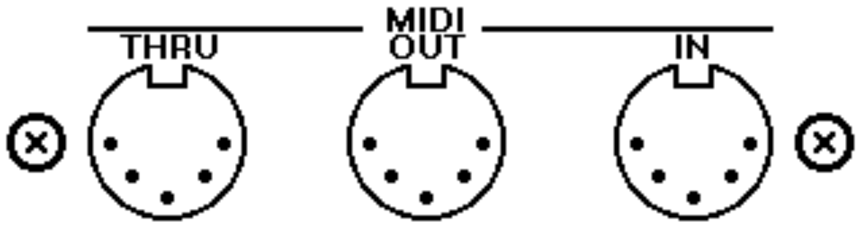
What is MIDI?



By Paul D Lehrman, PhD
Director, Music Engineering
Tufts University

Sections adapted from MIDI For The Professional by
Paul D. Lehrman & Tim Tully

What is MIDI?

By Paul D. Lehrman, PhD
Lecturer in Music and Director of Music Engineering,
Tufts University

MIDI is an acronym for "Musical Instrument Digital Interface." It is primarily a specification for connecting and controlling electronic musical instruments. The specification is detailed in a document called, not surprisingly, "MIDI 1.0 Detailed Specification", which is published and distributed by the MIDI Manufacturers Association and available free to all MIDI Association Members on the www.midi.org website.

The primary use of MIDI technology is in music performance and composition, although it is also used in many related areas, such as: audio mixing, editing, and production; electronic games; robotics; stage lighting; cell phone ring tones; and other aspects of live performance.

The MIDI command set describes a language that is designed specifically for conveying information about musical performances. It is not "music", in that a set of MIDI commands is not the same as a recording, say, of a French horn playing a tune. However, those commands can *describe* the horn performance in such a way that a device receiving them—such as a synthesizer—can *reconstruct* the horn tune with perfect accuracy.

Included in the MIDI Specification is a method for connecting MIDI devices via 5-pin DIN connectors and cables, which will be explained later in this article. But over the years other means have been developed for sending and receiving MIDI commands, such as USB-MIDI and Bluetooth-MIDI, and readers should consult the www.midi.org website for the latest information regarding enhancements to the MIDI Specification.

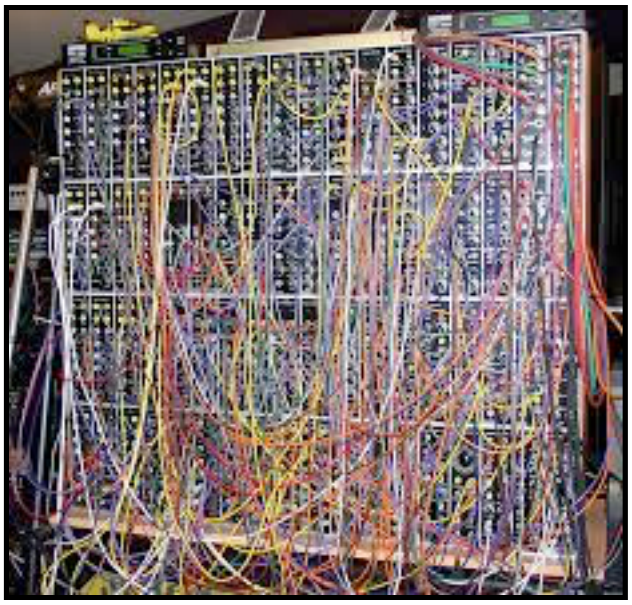
How it came about: a short history of electronic music

Electronic music has been around since the 1940s, and in some respects it's even older than that. In the early days, electronic composers built studios out of discrete components, like oscillators, filters, mixers, and frequency shifters, many of which were originally designed for use in decidedly non-musical contexts, such as radio repair, testing laboratories, audiology clinics, and telephone networks, as well as conventional recording studios. From these tools, which made up the "classical" electronic-music studio, composers could create a wide variety of interesting and (in the right hands) musical sounds.

Some inventors in those early days designed electronic gear expressly for musical purposes, and among the results were instruments that had their own distinct sounds, like the Theremin, the Trautonium, and the Ondes Martentot, and instruments that used electronics to imitate conventional instruments, like the Hammond organ and the Cordovox.

The dawn of synthesizers: patch cords and voltage control

By the 1960s, the first dedicated electronic-music systems, known as “synthesizers,” appeared. Early synthesizers consisted of modules that took over the functions of the discrete components of the classical studio, as well as new types of modules specially designed for music manipulation. These included envelope generators, which change a sound’s amplitude (volume) over time; envelope followers, which impose the amplitude envelope of one sound onto another; modulators, which combine two or more sounds in ways that create more complex sounds; and sequencers, in which discrete events or parameter values can be stored and played back over time, with the tempo determined by a timer or “clock”, which itself was often a module.



Because the modules in a synthesizer were designed to work together, interfacing them in unusual and creative ways was relatively easy: an envelope generator could be used to open and close a filter, for example, so that a sound’s tone or timbre could evolve over time, as well as (or instead of) its volume. A sequencer could be used not only to play a series of pitches, but also to change—in discrete steps—the volume or tonal characteristics of a sound over time.

Modules were connected together with short cables known as “patch cords”, and the connection of many cables to create a particular sound or sequence of sounds became known as a “patch.” This term survives today in digital and software synthesizers to describe a particular sound or “voice.” In the modular synths, the patch cable carried voltages, and the modules communicated with each other using “voltage control,” in which a module receiving a varying voltage behaved as if it had a knob that was being turned by an unseen hand.

Here’s an example: a conventional oscillator has a knob for setting its output frequency. A voltage-controlled oscillator instead has a control input that accepts varying voltages and changes the output frequency based on that voltage. A 1-volt signal may produce a tone at 100 Hz, a 2-volt signal a 200-Hz tone, a 3-volt signal a 400-Hz tone, and so on. With each 1-volt increase in the control voltage, the pitch of the oscillator rises by one octave: i.e., its frequency doubles. This is known as “1-volt-per-octave” voltage control. Voltage control can also be applied to a filter, in which it can vary the cutoff frequency or bandwidth; to a mixer, in which it can change the level of one or more inputs; or to a sequencer’s clock, in which it can control the tempo.

Voltage control became very popular among synthesizer manufacturers, but it had its problems. Since there was no standardization of voltage-control schemes, synthesizers from different manufacturers were often incompatible with each other: some used 1-volt-per-octave, while others used different ratios or curves, so that a control change that raised the pitch an octave on one synthesizer might only raise it a minor 3rd on another. Also, because the signals were analog in nature, absolute accuracy and repeatability were hard to achieve. Two sounds in a patch might be in tune one moment and out of tune a few minutes later, as the components heated up and the voltages drifted slightly.

A different type of signal was necessary for timing, where audio events produced by different modules had to be synchronized or coordinated. Instead of smoothly changing voltages, what was needed was a more immediate, binary (on/off) signal. These signals, generally pulses of a certain length, were known as “triggers.” A sequencer could fire off triggers to initiate sounds as easily as it sent voltages that controlled timbres. Unfortunately, the variations in triggering schemes among different synthesizer makers were even greater than those in voltage control.

In recent years, modular synthesizers have made a comeback among musicians who like their “hands-on” way of controlling sound. Especially popular is a modular rack format generally referred to as "Eurorack", with dozens of companies around the world offering modules that can be assembled into a system. Fortunately for fans of these instruments, their manufacturers have made their devices more compatible (although not completely) with each other, and modern electronic components are more stable, so that drift is not as much as a problem. Many manufacturers make MIDI-to-trigger and control-voltage converters, so that their instruments can be integrated into a MIDI studio or performance rig.

Digital Communications

In the late 1970s, synthesizers using digital electronics were introduced, largely solving the problems of repeatability and oscillator drift. Digital synthesizers use mathematical algorithms to produce digital “models” of waveforms, and convert them into audio signals using digital-to-analog convertors, or DACs. “Patches,” now also known as “programs,” became lists of digital parameters, describing module settings and signal routings. Programs could be stored in a synthesizer’s internal digital memory, and recalling one became a simple matter of pushing a button.

The next challenge was to figure out a way to use digital electronics to connect synthesizers to each other. As synthesizer-based rock bands became more popular, a common plea among performers who found themselves trucking around huge arsenals of electronic keyboards was for a system that would let them operate all these instruments from a single, common keyboard, and use switches or other compact devices to determine which sounds they produced when.

Roland, Oberheim, Sequential Circuits, and Fender Rhodes were among the manufacturers who developed proprietary digital control schemes that allowed keyboard synthesizers to be slaved to each other, as well as external sequencers, capable of playing entire songs or sets, to be integrated with them. For example, a musician using a Roland synthesizer would play a piece of music on the keyboard, and data representing all of the keystrokes would be sent out of the synthesizer on a special digital control cable, connected to a Roland sequencer in its own box. The sequencer recorded the keystroke data in real time, and could play it back later with great accuracy. The sequencer could be connected—again with a digital cable—either to the same instrument from which the music originally came, or to another Roland synthesizer capable of reading the digital information.

Although some of these digital control schemes were highly evolved, the problem of incompatibility remained. Each manufacturer had its own ideas for implementing digital control, so instruments from different manufacturers still couldn’t communicate with each other.

Sequential Circuits, a California synthesizer maker founded by an inventor named Dave Smith, became the first company to propose a common digital interface for synthesizers from different manufacturers, and introduced its idea of a Universal Synthesizer Interface (USI) in October 1981 in a technical paper at a convention of the Audio Engineering Society (AES). Meetings followed over the next year among representatives of several American and Japanese electronic instrument makers, and the first MIDI synthesizer, Sequential’s Prophet 600, shipped in December 1982. The first two MIDI synthesizers from different companies were publicly hooked together a month later (a Prophet 600 and a Roland Jupiter 6) when Dave Smith from Sequential Circuits and the late Ikutaro Kakehashi from Roland Corporation introduced the technology to an astonished crowd at a meeting of the National Association of Music Merchants (NAMM).

MIDI synthesizers immediately hit the market from a number of manufacturers, and the official MIDI Specification 1.0 was published in August 1983 based on the input from and collaboration between the leading synthesizer companies at the time. Some manufacturers held onto their own control schemes for a little while, thinking theirs was still a better way to do things, but it quickly became obvious that MIDI was going to become the undisputed industry standard, and soon just about every serious electronic instrument under development included MIDI capability.

The MIDI protocol became hugely successful very quickly, and fueled by MIDI technology, the electronic music hardware and software industries exploded. Today MIDI technology is found not just in keyboards and hardware synthesizers, but also in all sorts of musical controllers, software synths, mixing and DJ consoles, audio processors, games, theatrical control systems, Websites, and billions of smartphones.

The Goals of MIDI

MIDI is based on two major principles: **universality** and **expandability**. Every manufacturer of MIDI equipment, if it uses the word “MIDI” anywhere on the case or in the documentation, is expected to implement MIDI technology correctly. This means that the device must accept and correctly act upon (or ignore, if appropriate) the data generated from any other MIDI device, and also that any MIDI data it generates must be understood (or ignored, if appropriate) by any other MIDI device, without causing any errors. It also means that every MIDI command has the same meaning to the receiver as it does to the transmitter (which is not to say that it has to be *interpreted* the same way, but that's up to the user).

Just as important as a device’s ability to know what to do when it receives MIDI is that it knows what *not* to do. The MIDI Specification is a very comprehensive set of commands, and no device on the market can respond to every command in it. This is perfectly okay, as long as the device knows enough to *ignore* the commands it doesn’t understand, and not try to interpret them. This feature is a key aspect of MIDI’s universality, and has also allowed the MIDI Specification to be expanded over the years. New commands added to the spec that aren't understood by older devices are simply ignored.

Flexibility

The MIDI command set was deliberately designed with “holes” in it: commands that are purposely left undefined. This was done so that as new uses for MIDI are developed, the command set can be expanded to accommodate them. For example, MIDI Time Code Quarter Frame Messages were not in the original Specification and the command they use was “Undefined” in the original Spec. Since devices built before 1987 were designed to ignore commands that were undefined at the time, MIDI Time Code messages should not screw them up—they will continue to ignore the messages.

Other new functions are possible using new combinations of commands. The MIDI Sample Dump Standard, added in 1986, and MIDI Show Control, added in 1991, both utilize an originally unused form of System Exclusive messages known as “Universal System Exclusive.” Devices, new or old, that don’t understand these messages will ignore them. Other MIDI commands have been redefined, or their meanings clarified, through periodic published supplements and revisions of the MIDI Specification, but they remain basically unchanged, and their compatibility with existing MIDI devices is never compromised.

Today, the MIDI Specification is still referred to as “1.0”, even though it has expanded substantially through the years. The original command set is still contained in the Specification, and will never be altered. Every MIDI device ever made should be able to talk to every other MIDI device ever made—even though some devices will do a lot more than others. This is one of the most profound aspects of the MIDI Specification: a device created today with a MIDI port on it is still capable of communicating with a device manufactured over 30 years ago.

Simplicity

The MIDI Specification is, compared to a specification like MP3 digital audio, quite simple. That's because it takes advantage of a principle called "distributed intelligence." Instead of a central controller doing all the work and sending all the musical waveform data, and the peripherals being merely passive converters of that data into analog sound, in MIDI the peripherals themselves have computer processors and memory.

MIDI assumes that the receiving device can take care of the task of defining and producing the sound when it receives an appropriate command. This means that the structure of the language can be kept small and simple.

Costs and Compromises

Another, less obvious, goal of the original designers of MIDI was to keep costs down. This was done in the interest of universality, so that as many manufacturers as possible would adopt the protocol. When it was first introduced, it was estimated that adding MIDI technology to a digital synthesizer would cost between $5 and $10 at the manufacturing level. Anything more than that, the reasoning went, would give too many manufacturers an excuse to balk at incorporating it into their products.

This decision was not universally hailed, however. There were certain trade-offs that had to be made to keep the cost of MIDI down (especially hardware cost) which many people thought would severely limit its capabilities. As it turns out, these critics were not entirely wrong, and there are a number of issues involving the speed and data capacity of the MIDI-DIN transport, but these issues have largely been resolved by transmitting MIDI data over alternate transports like USB.

How It Works

The MIDI Electrical Specification defines a serial data protocol, which means that MIDI commands are sent one "bit" at a time over the cable (although other transports may use a different approach). A "bit" is a single binary digit: on or off, 0 or 1. In the MIDI protocol, bits are grouped into "bytes" of 8 or 10 bits, depending on how you count them (which we'll explain in a moment), and the bytes are combined into commands, or messages. A MIDI command may consist of one byte, or two, or three, or many. What the MIDI Specification does, in a nutshell, is to specify how the commands—the groups of bytes—are to be structured and deciphered. It also defines the electrical characteristics of circuitry that can be used to transmit and receive the data.

Bits and Bytes

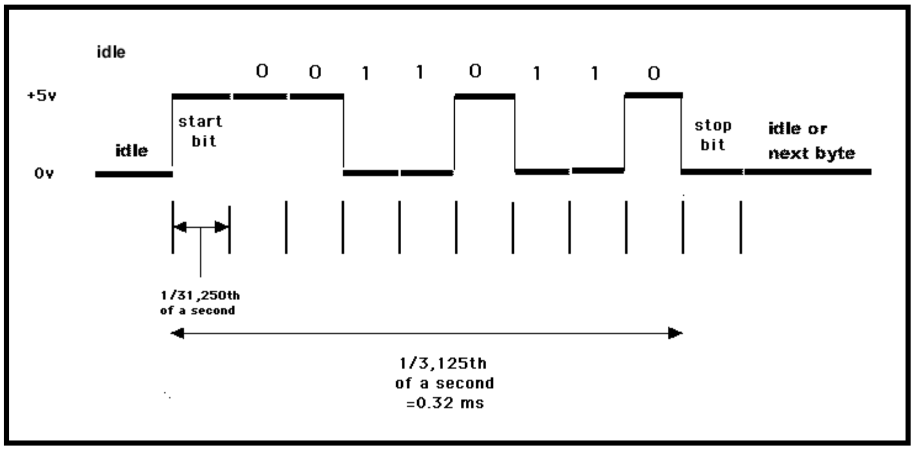
The MIDI Specification includes both hardware (the electrical spec, also known as "MIDI-DIN") and software (the message spec, also known as "MIDI Protocol") and in its original form both of these parts were necessary to transmit MIDI data.

When MIDI is sent over a MIDI-DIN cable, the signal is a pulse, like a square wave, transmitted 31,250 times per second, on a 5-volt line. The MIDI spec allows a ±1% tolerance in the speed—anything slower or faster than that will not work. This speed was chosen because 31,250 is a power-of-two factor—1/32nd—of 1,000,000. Back in the 1980s, 1,000,000 cycles per second (1 MHz) or half of that rate (500 kHz) were common clock speeds for the integrated circuits (or "CPUs", for "Central Processing Units") at the heart of personal computers, synthesizers, and other digital devices, and dividing that clock by a power of two to generate a MIDI stream is a simple thing for software and circuit designers to do. Since all microprocessors today still operate at (often very large) multiples of 1 MHz, that initial choice remains a good one.

The MIDI-DIN signal is "asynchronous", which means there is no underlying clock pulse going all the time unlike, for example, digital audio and SMPTE timecode, so a transmitter can start sending data at any moment, and the receiver is expected to understand it as soon as it arrives.

Since it is a digital signal, the pulse has two states: Logical 0 and Logical 1. These are actually “upside-down”: a Logical 0 is the current-*on* pulse, and Logical 1 is current-*off*. Since an idle MIDI line is a line with no current flowing, the first bit in any MIDI byte is always a 0 (On), and is known as the “start bit”. Following the start bit are eight data bits, which can either be 1s or 0s, followed by a “stop bit”, which is always 1 (Off). Therefore, a MIDI byte has 10 bits, but only eight of them actually carry information.

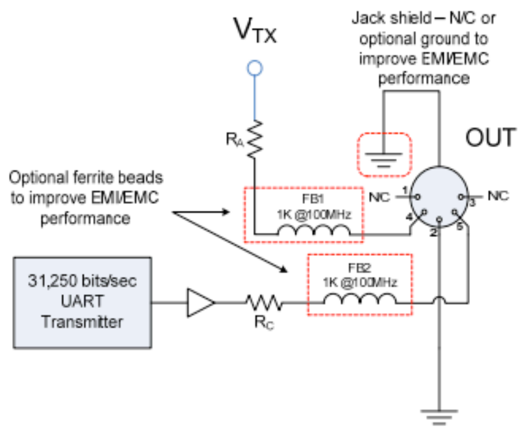
The fact that the last bit is Off simply means that there has to be a minimum amount of Off time between one MIDI byte and the next: a “resting” interval of 1/31,250 second. If that tiny “rest” is not there, and the next start bit (On) comes too early, the result is called a “framing error”, and the data will not be received correctly.



Since the MIDI-DIN bit rate is 31,250 bits per second, and there are 10 bits in a byte, the MIDI *byte* rate is 3,125 bytes per second. Most complete MIDI messages require two or three bytes, so the transmission rate over MIDI of musical data is approximately 1000-1500 “events” per second. This is pretty fast, but not infinitely so. It means, for example, that there is no such thing as two absolutely simultaneously-occurring events in MIDI—any two events must be at least 0.6 milliseconds apart. While for the most part this is an acceptable situation, under some circumstances it can be a limiting factor in the performance of a MIDI system.

Jacks and Cables

MIDI data is sent in a "simplex" fashion, meaning it is sent in one direction only, from a transmitter to a receiver. With MIDI-DIN, MIDI data is sent from one physical device to another, emerging from the “MIDI OUT” jack on the sending device, using a "MIDI cable". A MIDI cable consists of a shielded two-conductor twisted pair, the same as balanced audio cable. The MIDI OUT jack is a 5-pin circular DIN-style connector. DIN connectors were at one time common in European hi-fi equipment, which is where the name came from: it’s an acronym for “Deutsche Industrie Norm,” which means nothing more exotic than “German Industrial Standard.”


VTX
Jack shield – N/C or optional ground to improve EMI/EMC performance
OUT
RA
Optional ferrite beads to improve EMI/EMC performance
FB1 1K @100MHz
NC
NC
1
3
4
5
2
FB2 1K @100MHz
31,250 bits/sec UART Transmitter
RC

Pin 2 must be tied to ground on the MIDI transmitter only.

The buffer between the UART transmitter and R_C is optional and system-dependent.

The UART is configured with 8 data bits, no parity, and 1 stop bit, or 8-N-1.

The resistor values depend on the transmission signaling voltage, V_{TX}, as detailed below.

The optional ferrite beads are 1k-ohm at 100MHz such as MMZ1608Y102BT or similar.

V_{TX}	+5V ± 10%	+3.3V ± 5%
R_A	220Ω 5% 0.25W	33Ω 5% 0.5W
R_C	220Ω 5% 0.25W	10Ω 5% 0.25W

Figure 1 – MIDI OUT Circuit

The MIDI signal is on Pin 5 of the connector, and the voltage to drive the circuit (see Figure 1) is on Pin 4. Pin 2 is connected to the cable shield, and Pins 1 and 3 are not normally connected.

The other end of the MIDI cable is then plugged into a "MIDI IN" jack on the receiving device, which uses the same pin configurations, except that Pin 2 of the jack, the shield, is *not* connected. This is done to avoid the possibility of ground loops, in which different ground potentials of various devices connected together cause hum. In a studio containing dozens of discrete components, ground loops from audio and power cables are always a problem, and the designers of MIDI didn't want their cables to contribute to the problem. In addition, the MIDI Specification states that the actual shield connections on the MIDI jacks should never be connected to any chassis or electrical grounds.

In fact, to insure further against grounding and other possible electrical problems, MIDI IN jacks are not actually hard-wired to the synthesizer or other device on which they're mounted. Instead, all MIDI IN jacks contain an "optoisolator", an electronic device consisting of a tiny light-emitting diode (LED) and a photocell. When the jack receives a bit, the LED lights up, and the photocell responds by sending current into the rest of the receiving device. Therefore, between the MIDI encoding/decoding circuitry on two different pieces of equipment there is never any *direct* electrical connection.

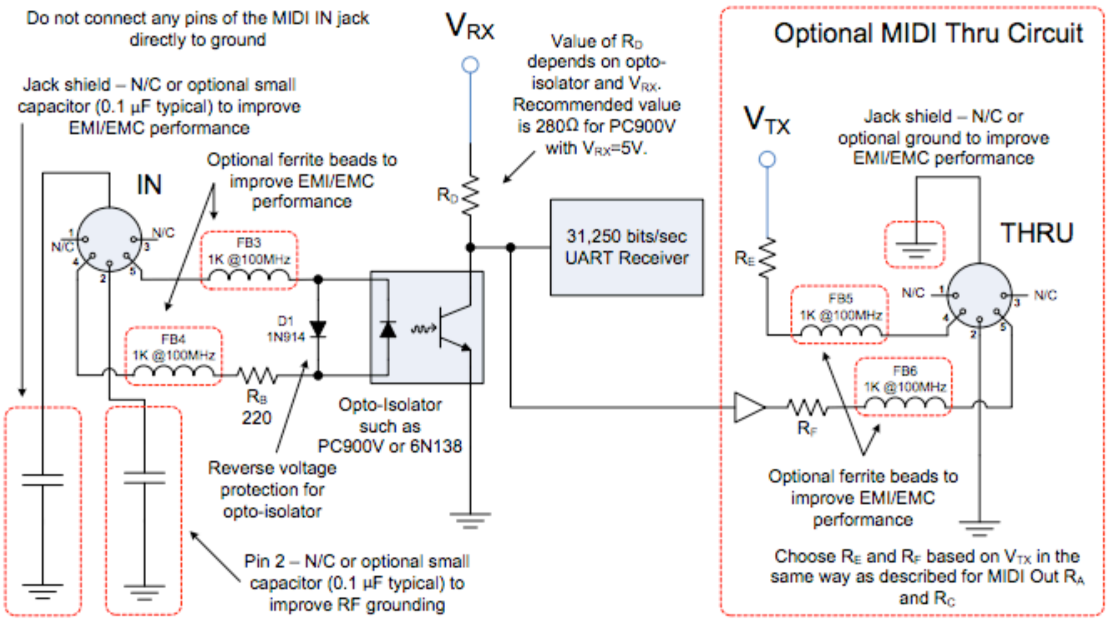
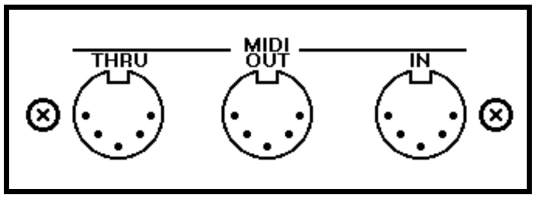

Do not connect any pins of the MIDI IN jack directly to ground
Jack shield – N/C or optional small capacitor (0.1 µF typical) to improve EMI/EMC performance
IN
N/C
N/C
Optional ferrite beads to improve EMI/EMC performance
FB3 1K @100MHz
FB4 1K @100MHz
D1 1N914
RB 220
Reverse voltage protection for opto-isolator
Pin 2 – N/C or optional small capacitor (0.1 µF typical) to improve RF grounding
VRX
RD
Value of RD depends on opto-isolator and VRX. Recommended value is 280Ω for PC900V with VRX=5V.
31,250 bits/sec UART Receiver
Opto-Isolator such as PC900V or 6N138
Optional MIDI Thru Circuit
VTX
RE
Jack shield – N/C or optional ground to improve EMI/EMC performance
THRU
N/C
N/C
FB5 1K @100MHz
FB6 1K @100MHz
RF
Optional ferrite beads to improve EMI/EMC performance
Choose RE and RF based on VTX in the same way as described for MIDI Out RA and RC

Figure 2 – MIDI IN and THRU circuit

The optoisolator is commonly connected to a chip known as a "UART," for "Universal Asynchronous Receiver/Transmitter." The UART translates the MIDI pulses into a form that the receiving device can understand. In a MIDI transmitter it works the other way around: the UART takes data from the synthesizer and makes MIDI out of it.

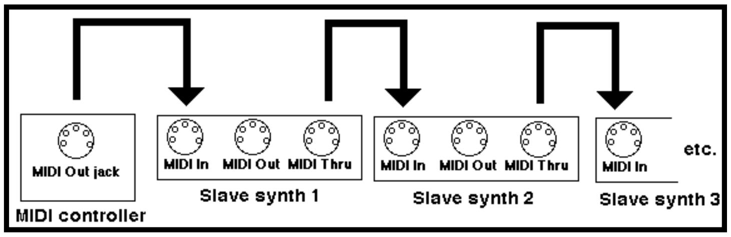
Thru Jacks, Boxes, and Mergers



Many MIDI devices have a third MIDI jack, labeled "MIDI THRU". "Thru" jacks are needed because, unlike an audio signal, you cannot send a MIDI signal to two different destinations merely by running it through a Y-connector. The optoisolator of the receiving device, so that it can screen out noise, is made to detect only signals above a certain voltage level. Dividing the

MIDI signal would cause its voltage to drop in half, and that would likely cause the optoisolator not to fire when a bit came in.

Therefore, MIDI Thru jacks are provided to “echo” the MIDI data coming into the MIDI In jack of a device, so that it can be passed on to another device. The MIDI Spec says that this echo must be perfect—no data can be changed or filtered—and it must be instantaneous: there can be no delay. The output of the optoisolator at the MIDI In jack must be directly connected to the MIDI Thru jack. In addition, the MIDI Thru jack normally does not pass MIDI data *created* by the device on which it’s mounted; that is what the MIDI Out jack is for.



In order to get the data from one MIDI transmitter to two different receivers, you must run a cable from the MIDI OUT jack of the transmitter to the MIDI IN jack of one receiver, and then another cable from the **THRU** jack of that receiver to the MIDI IN jack of the second receiver. The order of cabling—which receiver gets the direct line from the transmitter—in most cases doesn’t matter. This technique is called “daisy-chaining.” The number of receivers that can be in a daisy chain is theoretically unlimited, although there are some practical limitations.

Sometimes a MIDI transmitter, such as a sequencer, needs to send data to a number of devices, and daisy chaining is impractical. For those situations, “MIDI Thru boxes” are available, which contain a single MIDI IN jack, and four, eight, or even more MIDI THRU jacks.

A variation on the THRU jack that some devices offer is a “Thru/merge” option. With this option, MIDI data entering the MIDI IN jack is merged with data being generated by the device at the THRU jack, so that the Thru jack serves as both an OUT and a THRU.

Just as a MIDI output line cannot be electrically split, two MIDI input lines cannot be combined with a Y-connector. Besides possible voltage problems, this can cause two MIDI commands to interfere with each other, which would cause errors. “MIDI merger boxes” take care of this problem: they have two or more MIDI INs and a single MIDI OUT, and their internal electronics have sufficient intelligence to keep commands from colliding.

Getting MIDI into and out of a Computer

In the early days of MIDI, different models of computers had many different kinds of ports—serial, parallel, and expansion–which were designed for printers, modems, and other accessories, and there were "MIDI interfaces" (adapters for MIDI-DIN connections) available for most of them.

Today almost all MIDI interfaces are designed to be used with USB ports, and are compatible with almost all computers.

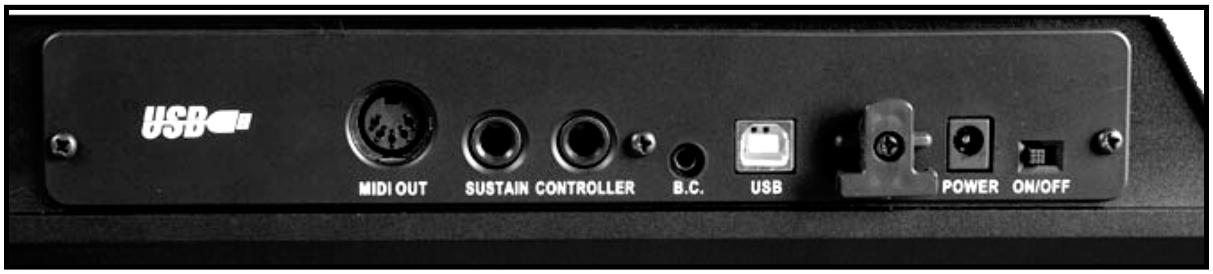
As MIDI grew in popularity, and musicians' studios became larger and more complicated, MIDI interfaces were introduced that had multiple MIDI In and MIDI Out ports. This gave musicians the ability to control a large number of MIDI instruments, and also made possible the popularization of "multitimbral" MIDI instruments, which in many cases, as we will see later, required dedicated MIDI cables.

MIDI without cables

Many of today’s MIDI tools exist only in software: controllers and sequencers interact with synthesizers *inside* the operating system of a computer, and therefore there is no need (or even way) to run cables between them. Instead, special code is installed in the operating system that allows these software devices to communicate with each other. There are a number of different methods for implementing this inter-application communication that have been developed by the makers both of these "virtual" instruments and of the computer operating systems themselves—Apple and Microsoft. They include Audio Units (AU), Virtual Studio Technology (VST), AAX, RTAS, Rewire, and DirectX. Even without cables, however, the MIDI Command Set works exactly the same way as if there were physical cables between the various components.

Alternative MIDI Connections

MIDI can also be transmitted over other types of cables, including USB, FireWire, Thunderbolt, and Ethernet. The most common of these at present is USB, and many of today's keyboards and synth modules have USB jacks. (These devices often have MIDI jacks as well to make them compatible with older systems.) All of these electrical "transports" have a much faster data rate than MIDI-DIN, and they can be bi-directional so there is no need for separate IN and OUT jacks. Again, the Command Set remains the same regardless of how the signals are delivered.



Despite these advantages, there are two potential problems when using USB and other transports for MIDI. One is that in most cases these devices cannot be connected directly to *each other*: they *have* to have a computer at the center of the system in order to communicate. So unlike with traditional MIDI cables, you cannotconnect, for example, the USB connector on a MIDI keyboard to the USB connector of a MIDI synth module: they both have to be connected to a computer, and the computer software is responsible for routing the MIDI data from the source keyboard to the destination synth.

The second is that the host computer itself needs to be able to recognize that the device plugged into is transmitting and/or receiving MIDI, and to correctly interpret the MIDI data going in or out. This is done by a piece of software in the operating system called a "driver." The major computer operating systems (Mac OS, Windows, iOS, Android, and Linux) all include "native" drivers (provided with the OS), but some manufacturers of USB MIDI equipment will design their products so that they require custom drivers. If a piece of gear works with the built-in OS drivers it is said to be “class-compliant”, and all is well. But if a manufacturer makes a piece of gear that is not class-compliant for whatever reason, it is the manufacturer's responsibility to provide proper drivers for their customers’ operating systems. These drivers are usually downloadable from the manufacturer's website, and then must be installed into the user's operating system.

In contrast, a piece of equipment that sends and receives data over conventional MIDI cables does not need any such driver: the fact that it is MIDI-compatible is enough to ensure that it works with other MIDI gear.

The Commands

When someone plays a MIDI “controller”, or transmitter, it generates one or more MIDI commands or messages. Playing a note on a keyboard, for example, sends a message corresponding to that action out the keyboard's MIDI OUT jack. The same thing happens when you hit a drum pad, press a pedal, move a pitch bend lever, move a fader, or blow into a wind controller.

A MIDI message contains the information for a complete musical action. When you press a key on a keyboard, it generates a message that consists of three bytes. The first byte describes the *kind* of action: a key has been pressed. The second byte is the *number* of the key that’s been pressed, that is, which note you’ve played. The third is the *velocity* with which the key has been pressed: the amount of time that has elapsed between the start of the key’s travel and the end. A higher velocity number means the travel time was less, which means the key was struck harder. This is usually, but not always, interpreted as higher volume.

Kind of action: a key is pressed

Command Byte = Note On

Which key is pressed?

Data Byte 1 = note number

How hard was the key played?

Data Byte 2 = velocity

The first byte is known as a “Command byte” or “Status byte” (the two terms are synonymous). Command bytes say, “Do something!” In this example, what it’s doing is called “Note On.”

The second and third bytes are the “Data” bytes. Data bytes say, “Here are the parameters for what to do regarding this particular command.”

The MIDI Specification dictates the exact meaning of all of the Status bytes (except the few that are purposely left undefined), as well as the number of Data bytes that must follow each Status byte, and what they mean. Some MIDI commands require one Data byte, some need two, some have none, and a special class of commands, “System Exclusive”, have an undefined, and often very large, number of Data bytes.

Binary, Decimal, and Hexadecimal Notation (Don’t skip this if you really want to understand how MIDI technology works!)

MIDI messages are 8-bit bytes. If we look at them as binary numbers, this means they range from 0 (all bits are zero) to 255 (all bits are 1). When we refer to these messages, we can do so in binary (e.g., 10010011) or in decimal (e.g., 147). But there is another form of notation that works even better, although it takes some getting used to. This is base-16, or hexadecimal notation, often called simply “hex”.

In hexadecimal notation, the far-right number is the ones column, but the second-from-the-right number, instead of being the 10s column as it is in decimal, is the *16s* column. So the number “14” in hex is equal to 20 in decimal: (1x16) + (4x1). “38” in hex is (3x16) + (8x1), or 56 decimal. In this document, the letter “H”, for “hex”, will always follow numbers in hexadecimal notation.

Why use hexadecimal? Because it is a very convenient way to express MIDI commands: every 8-bit MIDI value can be expressed as a two-digit hex number, and each type of MIDI command starts with a different hex digit, which can make things very tidy. For example, the MIDI message 10010011 can be broken up into two parts: 1001 and 0011, and each given a hex value: 9 and 3 in this case. So the MIDI byte 10010011 is 147 in decimal or “93” in hex. The byte isn’t really broken up into two parts when it is sent—this is just a convenience to let us look at and understand a MIDI data stream quickly.

One problem with hex notation is what to do with numbers above 9. The answer is to use letters:

A (hex) = 10 (decimal)

B = 11

C = 12

D = 13

E = 14

F = 15

So the number 47 in decimal is expressed in hex as 2F: (2x16) + F (that is, 15) x1. The possible values of a MIDI byte are 0 through 2^8-1, or 0-255 decimal, or 00-FF in hex.

Why is this an advantage over decimal notation? You'll see when we get into the details of how MIDI commands are structured.

Anatomy of a Command

Here's a typical MIDI command, in binary:	10010000	01000101	01100101
In decimal, this reads as:	144	69	101
In hex, this reads as:	90H	45H	65H

The first byte is the Command byte. The MIDI Specification says that any Command byte that starts with a 9 (in hex) is defined as a Note On. (We'll deal with the second digit a little later.) The specification then says the second byte is the note number. The lowest MIDI note (00H—we always use leading zeroes in hex notation to make sure every byte has two digits) is C, five octaves below Middle C. The number 45H in the example translates to 69 in decimal, thus we want the 69th note above the lowest C: that's the F above Middle C. The third byte, 65H, according to the Specification, is the velocity, as we discussed earlier. In decimal, this number is 101. Since the maximum velocity of a Note On is 7FH (127), this note is pretty loud.

If the first bit of a MIDI message is a "1", then the message is a Command byte. If the first bit is a "0", the message is a Data byte. This makes a receiver's initial task when it analyzes a received MIDI byte very simple: if it sees the first bit is 1, it knows the byte is a Command byte, while if it sees it is 0, it knows it is a Data byte, and it knows to relate the Data byte to the last-received Command byte.

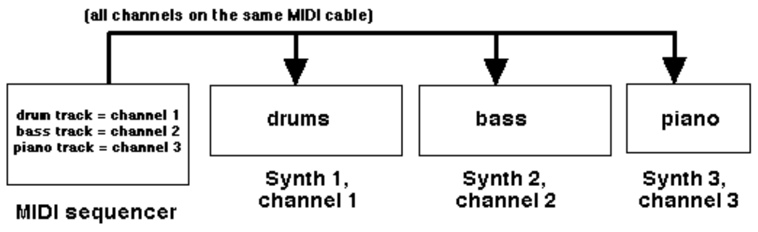
Here's where hex notation comes in handy: In hex, all Command bytes start with **8 or more**. All Data bytes start with **7 or less.** Compare that with their decimal equivalents: Command bytes are 128-255, Data bytes are 0-127. Not exactly intuitive, is it?

For a MIDI message to be interpreted properly it must consist of the correct number of bytes, as dictated by the MIDI Specification, and it may not be interrupted except in a few very special cases. If an insufficient number of Data bytes follow a Command byte before the next Command byte is received, the receiving device will assume the information is in error. Under the best of circumstances, this kind of error is simply ignored, but in some devices it can cause strange behavior or even hang up a component completely, so maintaining a clean MIDI data stream is always important.

A special case occurs when the number of Data bytes following a Command byte is too *high*. This invokes a condition called "Running Status" which we'll discuss a little later.

MIDI Channels

The MIDI Specification allows for 16 data channels. This is so that multiple MIDI devices can each receive distinct information with only a single MIDI cable (real



or virtual) connecting them. Just like the tuner in a television set, each receiving device can be set to recognize data on one and only one specific MIDI Channel, and to ignore data on all the other Channels.

Here's an example: a sequencer sends data to three synthesizers, one playing drum sounds, one bass sounds, and one piano sounds. The sequencer can send each musical part on its own MIDI Channel, and put them all on the same cable. Since each synthesizer is "tuned" to the appropriate Channel, it plays only the notes intended for it, and ignores the rest, so the drums don't play the bass part.

The Channel identity of a MIDI command is located in the *second* half of the Command byte—the second four bits, which (Here's another place where hex comes in handy!) make up the second hex digit. That digit can be 0 to F (hex), or 0 to 15 decimal, which gives us 16 MIDI Channels. In the physical world, we don't speak of "Channel 0" (nor do most people speak in hex), so MIDI Channels are normally referred to as 1 through 16. Therefore, the MIDI Channel of a Command byte whose second digit is *n* is actually *n*+1.

So the Command byte for a Note On command on MIDI Channel 8 will be 97H. A Note On command on Channel 12 will begin with the Command byte 9BH (since B in hex = 11 in decimal).

While the first MIDI synthesizers operated on only one Channel, soon synthesizers appeared that could play different sounds on different Channels at the same time. These are called "**multitimbral**" instruments, and they behave like 16 discrete synthesizers, all addressed on a single cable. Since they would take up all of the Channels on a single MIDI cable, multitimbral synths, as we mentioned earlier, were largely responsible for the growth of multiport MIDI interfaces, which allowed musicians to connect to and work with more than one multitimbral instrument in their studios.

The Command Set

Here is the Command set in a simple table in numerical order. The first half (four bits, or first hex digit) of the Command byte determines the nature of the command, and the second half the Channel.

Name	**Hex values** (channels 1-16)	**Decimal values**	**Data bytes**
Note Off	80-8F	128-143	2 (note number, velocity)
Note On	90-9F	144-159	2 (note number, velocity)
Key Pressure	A0-AF	160-175	2 (note number, pressure)
Control Change	B0-BF	176-191	2 (controller number, value)
Program Change	C0-CF	192-207	1 (program number)
Channel Pressure	D0-DF	208-223	1 (pressure)
Pitch Bend	E0-EF	224-239	2 (LSB, MSB)
System Exclusive	F0	240	variable
System Common	F1-F6	241-246	0, 1, or 2 (see below)
End of System Exclusive	F7	247	0
System Real Time	F8-FF	248-255	0 (see below)

Notes

9nH is Note On. (Remember, n+1 is the Channel number.) When it is received, a note starts playing. As we saw earlier, this command has two data bytes: note number and velocity. Since the first bit of a MIDI Data byte is always zero, this gives us a range of 00000000 to 011111111, or 00H to 7FH, or 0 to 127 decimal, for both note numbers and velocities.

As far as the note number is concerned, each increment in value is normally equal to a half-step, so this numerical range gives us a musical range of 128 half-steps, or more than 10-1/2 octaves. That's a lot bigger than your average grand piano (which has 88 keys, or 7-1/3 octaves), and in fact is greater than the range of human hearing which, at its best (about 20-20,000 Hz), is just under 10 octaves.

As far as velocities are concerned, this gives us 128 values between ***pppp*** and ***ffff***, which is plenty. (Although for reasons we'll explain a little later, a velocity value of zero gives this command a different meaning, so there are actually only *127* values. Not much difference.)

Even inexpensive keyboards that are not velocity sensitive need to send a velocity byte with their Note On messages. The Spec requires non-velocity-sensitive keyboards to send a velocity byte of 40H (=64 decimal).

8nH is Note Off: a key has been released. The note stops playing, or its envelope goes into the Release stage, if it has one. Like Note On, Note Off needs two Data bytes: the note number, and the velocity, which in this case is the amount of time the key takes to return to its "up" position. If you send a Note On for a specific note, you can't stop the note playing just by sending *any* Note Off, you have to send a Note Off with the ***same*** note number.

The ability to respond in a meaningful way to Note Off velocity is not common on MIDI synthesizers, but it is sometimes used. Note Off velocity can be used to affect the speed of a Release stage, or the volume of an "after" sound, like the dropping of the quill back onto the string in a harpsichord.

By convention, **a Note On with a velocity of 00 is considered equivalent to a Note *Off*.** This is important in the use of Running Status, which we'll talk about in a moment. Some devices transmit only Note Ons, and use Velocity-zero Note Ons to turn notes off, while some transmit true Note Offs. The MIDI Specification states that either method is acceptable.

Program Change

CnH (remember, C=12 decimal) is **Program Change**, sometimes called "patch change". As we saw earlier, in a digital synthesizer a "program" is a memory location containing all the values of the parameters that determine an instrument's sound. One set of values—for waveforms, frequencies, envelopes, signal routings, etc.—might define a brass sound, while another defines a flute sound, and another a funky bass.

You can change programs on a synthesizer's front panel by pressing a button. The MIDI Program Change message sent from a MIDI controller to another MIDI synthesizer lets you do this remotely.

In live performance, this means you can press one button on a keyboard and one or more synths in a stack will change their sound. In a sequence, it means that a single synthesizer can have different instrumental identities at different points in a song, if you send out appropriate Program Change commands at the proper times. Program Changes are also used by effects devices to change their identity: for example, one program in a device may be for flanging and another may be for hall-type reverb; and by mixing consoles to change "scenes", or set-ups for different songs. A Program Change message transmitted from a keyboard or a sequencer allows the device to change instantly at that moment.

If a MIDI synthesizer receives a Program Change message while it is in the middle of a note, a number of things can happen, depending on how the synthesizer is designed. Some synthesizers immediately fade the note out. Some try to execute the Program Change and impose the new program's parameters on the current program, often with unpleasant results. More recent devices (usually ones that have a feature called “dynamic voice allocation”) will let the current note finish unaltered, and will change the sound only on notes that start *after* the Program Change message was received.

The Program Change message has only one Data byte: the program number. Therefore it can only address 128 different programs in a synth. Most modern synths have memory for far more programs than that, so there is another command, called Bank Select, which we'll look at shortly that expands this command's capabilities.

Continuous Controllers

BnH is **Control Change**. A Continuous Control Change message (often called just “Controller”, which should not to be confused with a controller *device*, like a keyboard) generally refers to a musical gesture that is **not** a note. Often it is something that can change the character of a sound after the note has started. Common examples of Controllers are **modulation wheel**, which normally adds vibrato or changes a filter parameter; **sustain pedal** (which in fact is usually a switch), which keeps the Sustain portion of the envelope going; or **volume pedal**, which allows level changes or fade-ins and -outs.

Note that MIDI *volume* is a very different concept than MIDI *velocity*: the former is a continuous parameter that can change while the note continues to sound, while the latter only affects the note when it starts (or stops, in the case of Note Off velocity). Also, like all Control Change messages, MIDI Volume affects **all** the notes on the Channel it is sent to, while velocity affects **individual** notes.

The MIDI Spec allows for 128 different Control Change messages. A Control Change message has two data bytes: the Controller number, and the value of the Controller at that moment. Control Change messages often occur in streams, representing the continuous movement of a wheel, pedal, or slider.

Many controllers have designated functions in the MIDI Specification: Modulation wheel is Controller 01, Volume pedal (or knob, or slider) is Controller 07, and Sustain pedal is Controller 40H (64 decimal). These designations are not *absolutely* required in a piece of MIDI gear, but they are provided to minimize confusion and make communications as smooth as possible. The number of controllers available in the MIDI spec means that up to 128 different parameters on a synthesizer—hardware or software—can be modified in real time using separate Control Change commands. On a multitimbral synth, each Channel has its own set of 128 Control Change commands.

When MIDI was being developed, there was concern that the number of values available in a Control Change command—128—was too small. For example, if a synth's volume control has 128 discrete levels, when you change the level you might hear the change in volume as an audible "step" in the sound. When a lot of those steps occur quickly, such as when a volume pedal is being moved, it could generate what is called "zipper noise." So the creators of MIDI specified that Controllers 00-19H (0-31 decimal) could be paired with controllers in the range 20-39H (32-63 decimal), to address the same function, and together these "double-precision" controllers could describe 128x128, or 16,384 different values. Controller 01 was paired with Controller 21H (33 decimal), Controller 07H was paired with Controller 27H (39 decimal), and so on. When Controllers are used this way, the lower-numbered Controllers are the Most Significant Bytes (MSB), while the higher-numbered ones are the Least Significant Bytes (LSB).

In practice, however, double-precision 14-bit Controllers turned out to be mostly unnecessary, and so their use is now rare, with manufacturers just sticking with the lower-numbered (MSB) Controllers and using the higher-numbered ones for different functions, unrelated to the lower-numbered ones.

A special meaning was given to Controller 00 a few years after the initial adoption of the MIDI Specification. As we mentioned earlier, today's synths have room for far more programs than the Program Change command can address, so Controller 00 was designated as a "Bank Select" command. Synthesizers that recognize this command can organize their programs into up to 128 different banks, each with 128 programs, or a total of 16,384 programs. The Bank Select command by itself does not change programs in a synth; it simply selects the bank from which program changes will be selected—from 0 to 127 based on the command's second Data byte—which the synth will choose from when it receives the next Program Change command.

Channel Mode Messages

Controller numbers 78H (120 decimal) and above are called “Channel Mode” messages, because they have very specific functions on the MIDI Channel they are sent. Not every device recognizes them, but they are useful in many circumstances.

Controller **78**H is “**All Sound Off.**” Any device that receives this message is supposed to immediately turn off all notes on the designated Channel, and set the volume of all voices to zero (that is, notes aren't allowed to fade). In an effects device, this message cuts off the tail of a delay or a reverb, but it doesn’t turn off the effect: new signals coming into the audio input are processed normally. The second Data byte, following the controller number, is usually 00.

Controller **79**H means “**Reset All Controllers**” (on the designated Channel): in other words, set all controller values to zero. The second Data byte is irrelevant.

Controller **7A**H is “**Local Control.**” Local Control refers to whether the keyboard on a synthesizer and the sound-generating circuitry are linked together—i.e., when you play a key, does it make a sound? When Local Control is On (the Data byte following the controller number is 7FH), it does. When Local Control is Off (the Data byte is 00), it does not. (Any other values for the last Data byte should be ignored, although some devices interpret any non-zero value as On.) While the purpose of it may not be initially obvious, Local Control is a very important function, and the ability to “decouple” a synthesizer and its keyboard, which occurs when Local Control is Off, is crucial in a multi-synth setup where a synthesizer is being used as a master keyboard.

Controller **7B**H means “**All Notes Off**” (on the designated Channel), and is useful in emergencies when notes are stuck on and you don’t know why. It lets you silence the sound quickly. Sending specific Note Off messages is always the best way to silence a synthesizer, but this command accomplishes the same thing in a hurry without having to first figure out which notes are sounding. Again, the second Data byte is ignored. It’s not quite the same as All Sounds Off, because notes are allowed to decay naturally, and it doesn’t necessarily affect the audio in effects units.

Controllers **7C**H through **7F**H are discussed under “Modes” later in this chapter.

Pressure

DnH is **Channel Pressure**, sometimes known as “mono aftertouch,” or just “aftertouch.” Channel Pressure is a measure of how hard a key is pressed *after* it is struck and is at the bottom of its travel. Special sensors below the keyboard are used to detect this pressure, which is often used to add vibrato to a voice, or to change its timbre or pitch. It works on all notes on a Channel—if pressure is applied to one key, all the notes are affected. Channel Pressure has one Data byte: the pressure value.

AnH is **Polyphonic Key Pressure**, sometimes known as polyphonic (or just “poly”) aftertouch. It is similar to Channel Pressure, except that each note has an *independent* sensor, and can be addressed individually. It allows, for example, vibrato or a timbral change to be applied to one note in a chord but not the others. A Polyphonic Key Pressure command has two data bytes: the note number, and the amount of pressure.

Polyphonic Key Pressure is somewhat rare in the world of MIDI, because it is highly data intensive, and the amount of data it uses can clog a conventional MIDI stream and cause delays or errors. But it is becoming more common as MIDI systems that use faster transports have become more available.

Pitch Bend

EnH is **Pitch Bend**. Pitch Bend is usually generated by a wheel, lever, or joystick, with a spring-loaded center return.

Pitch Bend works both upwards and downwards, but there is no way to send negative numbers over MIDI. Therefore, a data value of 00 is considered maximum *downward* bend, and “no bend” is a value in the *middle* of the range (40H, or 64 decimal), while 7FH is maximum upward bend. Obviously, no command at all is also interpreted as “no bend.” The Reset All Controllers messages described above also resets Pitch Bend to the middle of its range (but not to zero).

The amount that a sound’s pitch will change in response to a Pitch Bend command is determined by the *receiving* synthesizer: there will usually be a parameter in that device called “pitch bend range” or some equivalent, which will determine the pitch change, in half-steps, that will be effected when a Pitch Bend command of maximum (7FH) or minimum (00) value is received. If two MIDI synthesizers are expected to bend notes in concert, it is imperative that the pitch bend range in each of the synths is set to respond in the same manner. The pitchbend range parameter can often be set by a Control Change command.

Pitch Bend takes two Data bytes, the first one being the Least Significant Byte (LSB) and the second being the Most Significant Byte (MSB). As with 14-bit controller pairs described earlier, this means that the number of possible Pitch Bend values is 16,384. It was set up this way in the Spec so that even at the highest pitch bend ranges, smooth-sounding pitch sweeps could still be accomplished.

Practically speaking, however, as with 14-bit Controllers, the LSB is almost never used, and it is normally set either to zero, or to some arbitrary constant, or to the same value as the MSB. The MSB handles the entire range: 00 is maximum downward bend, 7FH is maximum upward bend, and 40H is no bend.

Even though there aren't really any negative numbers, some sequencers and other devices display Pitch Bend messages as positive and negative numbers, with a range of -64 to +64 (or -8192 to +8192 if they take the LSB into account).

Normally when you send a Pitch Bend command, all of the notes sounding on a Channel will be affected equally. However, some devices are smart enough to apply incoming pitch bend not to all notes, but just to notes being held down on the *keyboard*, and not notes being held with a sustain pedal. Therefore, if you play a chord and “latch” it with a sustain pedal, and then play a key and hold it down while you send a pitch bend command, the chord will *not* bend, but the single note you’re holding *will*. Since this can be used to simulate the bending of a single string within a guitar chord, it is sometimes called “Guitar mode”.

Running Status Messages

Normally, a MIDI message consists of a Command or Status byte and the appropriate Data bytes, which are then followed by the next message’s Command byte and so on. However, the MIDI Specification allows for a special condition in which a single Command byte can be followed by a long string of Data bytes.

Imagine you are playing a key, and while you hold it down, you vary the Pressure to create vibrato. You start with no pressure, increase it to maximum, and then slowly release it. This action could produce a total of 254 different Channel Pressure messages—127 up and 127 down. Since the Command byte is always the same, and only the Data byte changes, if it were possible to send just the changing Data bytes, it would reduce the amount of data by 50 percent. As we have seen, the bandwidth of a MIDI cable is finite; so reducing the flow of bits can help to avoid butting up against those limits.

"Running Status" allows you to do exactly this. It can be used with any of the messages described so far, whether they take one Data byte or two. Running Status is a “condition” of the MIDI data stream. It is invoked if a Command byte is received followed by a number of Data bytes that is *higher than the normal number* of Data bytes associated with the Command byte. The extra Data bytes are assumed to be associated with the last Command byte received, and are processed just as if the Command byte had been repeated. It is for this reason that a MIDI Note On with velocity 00 is considered a "note off": when Running Status is active, a note that is on can be turned off with just a note number and velocity 00 (2 bytes) instead of a full Note Off command (3 bytes).

System Messages

The commands we've talked about so far are collectively known as "Channel Voice Messages", because they contain Channel numbers and they address one musical "voice" or instrument, at a time, on a specific MIDI Channel. The remaining commands in the MIDI Specification are "System" messages, and contain no Channel numbers. The first digit of a System message (in hex) is always "F".

System Common

Messages F1H through F7H are known as "System Common," because they are common to all receiving devices.

Name	Hex value	Decimal value	Data bytes
MIDI Time Code 1/4 Frame	F1	241	1 (timecode nibble)
Song Position Pointer	F2	242	2 (MSB, LSB)
Song Select	F3	243	1 (song number)
Tune Request	F6	246	0
End of Exclusive (see below)	F7	247	0

F1H is used to tell a sequencer or other device how fast it should be playing (in "absolute" time, not in terms of tempo), and is based on the standard frame rate of a video signal. It is used in systems where a sequencer is following a device that is generating SMPTE timecode, like multitrack audio or video players. F1 messages are sent at a constant rate of approximately 120 times per second in North and Central America and Japan, or 100 times per second in most of the rest of the world.

F2H is used in non-SMPTE based systems to tell a MIDI sequencer where to start when it receives a Continue command (below). The data bytes specify the number of 16th notes from the beginning of the sequence. There are 16,384 possible values for the data bytes, so Song Position Pointer can be used to specify a location in a sequence up to about 1000 measures long.

F3H is used with sequencers that have multiple song memories to specify which song they are supposed to play when they receive the next Start command (below).

F6H is **Tune Request**. This was originally intended for use by digitally controlled analog synthesizers. It tells them to execute whatever self-tuning routines are built into them. Although there are still plenty of these types of synths around, very few respond to this command, mostly because they are stable enough that they don't need to be retuned after they are turned on.

System Real Time Messages

Another group of System messages are referred to as "System Real Time", because they generally are used to synchronize equipment as it is running. System Real Time messages have no Channel numbers, and no Data bytes. They can be inserted into the MIDI data stream at any time, even in the middle of another message, and they can even pop up in the middle of Running Status without interrupting it (i.e., they are not interpreted as new Command bytes).

Name	Hex value	Decimal value
Timing Clock	F8	248
Start	FA	250
Continue	FB	251
Stop	FC	252
Active Sensing	FE	254
System Reset	FF	255

The first four are used, similarly to those mentioned above, to synchronize time-based MIDI devices. **F8**H is **Timing Clock**, or just “Clock”. Timing Clocks are used to lock two or more MIDI devices to the same tempo. They are generated 24 times per quarter note, and they change with the tempo (unlike MIDI Time Code Quarter Frame messages, which can change with *tape speed*, but not tempo). A “master” device sends out Timing Clocks, and a “slave” follows them.

Timing Clock is also called “MIDI Sync”, although that term (which doesn’t appear in the MIDI Specification) is more properly used in a broader sense, encompassing the Start, Stop, and Continue commands, and sometimes Song Position Pointer as well. Therefore, when we speak of a device like a sequencer or drum machine being “MIDI Sync compatible,” it means it responds to *all* of these commands.

FAH is **Start message**. This tells a device to go to the beginning of its song and start playing, at the tempo determined by the incoming Timing Clock messages that follow immediately.

FBH is **Continue message**, which is similar to Start, except that the receiving device will play from its current location, not (necessarily) from the beginning. It often follows a Song Position Pointer message.

FCH is **Stop message**. This tells a device to stop playing, and wait for a Start or Continue (*not* just a Timing Clock).

Modes

In the early days of MIDI, Modes were a very big deal, and a description of them was found at the beginning of every discussion of the MIDI Specification. The advent of polyphonic, multitimbral synthesizers and samplers has made Modes far less important than they were. There are, however, situations in which they can be useful.

There are four MIDI Modes, which are sometimes settable from a synth’s front panel, and can also be set by using two Continuous Controller “switches”: Omni On/Off (Controllers 7D and 7CH) and Poly On/Mono On (Controller 7F and 7EH). The value bytes of all of these commands (except sometimes Mono On, as we shall see in a moment) are supposed to be irrelevant, but some synths require non-zero values or even 7FH in order to act on them.

In **Omni On** mode, a receiving device will respond to data on *all* MIDI Channels. In **Omni Off mode**, it will respond on just one Channel, or in the case of multitimbral synthesizers, multiple but specific Channels.

In **Poly** mode, a device will respond to multiple incoming notes (like a chord) polyphonically.

In **Mono** mode, the device will only play one note at a time: i.e., homophonically. (“Monophonic” is actually a term from the world of audio, while “homophonic” is the correct musical term for music consisting of a single voice, but the former is increasingly used to mean the latter. So it goes.)

These four Mode messages also imply an “All Notes Off”: whenever a synthesizer receives them, it must turn off all of its voices before switching modes. They are not, however, supposed to be used *instead* of a true All Notes Off message (see above).

So the four modes, corresponding to the four possible states of the Controller switches, are:

Omni On/Poly, also known as Mode 1, which is useful in situations with one MIDI transmitter and a small number of receivers, all of which are to be played simultaneously without any distinction by Channel number;

Omni On/Mono, or Mode 2, which is never used;

Omni Off/Poly, or Mode 3, which is by far the most common mode; and

Omni Off/Mono, or Mode 4, which deserves some further discussion.

Mono (Mode 4)

Omni Off/Mono has some interesting peculiarities. First of all, a synth in Mono mode will assign "priority" to incoming MIDI notes, to determine which note will actually sound. This priority scheme may favor the last note received, the note that has been held the longest, the highest note, or the lowest.

Second, in Mono mode, many synths assume a rudimentary kind of multi-timbral identity. In some early polyphonic MIDI synthesizers, such as Sequential Circuits' Six-Trak and Casio's CZ-101, the user had a choice of having all of the available "voices" (that is, the number of simultaneous notes the device was capable of playing) respond to one MIDI Channel with a single sound or timbre (Poly mode), or to assign each voice to its *own* MIDI Channel, with its own timbre (Mono mode). The number of Channels it could play was equal to the number of voices: six in the case of the Six-Trak, four in the CZ-101. This meant that one synthesizer could function as the equivalent of four or six individual instruments, each capable of playing only *one note at a time*.

Mono-mode synthesizers have a "Basic Channel". Notes are received on this Channel, as well as on the next *n*-1 Channels, *n* being the number of voices in the synthesizer. For example, if a four-voice Mono synthesizer has its Basic Channel set to 11, it will receive notes on Channels 11, 12, 13, and 14. If a four-voice Mono synth has its Basic Channel set higher than 13, either it will run out of voice/Channel assignments, or it will "wrap" the upper voices around to Channels 1, 2, etc. Control Change, Program Change, and other voice messages can normally be sent on any of a Mono synth's active Channels, and they will affect *only* that Channel.

When a Mono On message is sent, the Data byte following it specifies how many MIDI voice/Channel assignments will be made. If the second Data byte is 0, the device is supposed to respond on the Basic Channel and as many MIDI Channels above it as it has voices for. In practice, however, Mono synths have a fixed number of voices and Channels, and so the second Data byte in the Mono mode message is usually ignored.

Although multitimbral synthesizers have eliminated the need for fixed voice-allocation schemes like these, Mono mode still has some important functions. One is in conjunction with MIDI guitar controllers. A guitar controller connected to a six-voice synthesizer in Mono mode can send information for each string on its own MIDI Channel. This is particularly important when bending a string: the Pitch Bend generated will affect only the Channel controlled by that string, and no other Channels will be affected. (The whammy bar will usually affect all six Channels simultaneously.)

Mono mode is also used when you want to simulate an old-fashioned mono synth with a polyphonic synthesizer. Normally on a polyphonic synth, when you play two notes in quick succession, the envelope of the first may ring over into the second. In Mono mode, however the envelope of the first note will be *cut off* when the second note starts.

Many synthesizers use an implementation of Mono mode that allows smooth legatos to be played from a keyboard: when the second note is played, the envelope is not re-triggered, but instead the first envelope continues, at the pitch of the second note. This is particularly useful with MIDI wind controllers, or when trying to emulate a wind instrument or human voice. Often this legato function can be turned on and off with a specific Control Change command.

System Exclusive Messages

A message that starts with **F0**H is called “System Exclusive”. System Exclusive (often called “SysEx”) messages were originally designed so messages could be sent to individual devices in a system, specified by manufacturer and model, and other devices on the line (which might even be tuned to the same MIDI Channel) would ignore them.

Here's a simple application of System Exclusive commands: a set of parameters describes a program; say a piano sound, on a certain model of synthesizer. You have another synthesizer that’s the same model, and you want that piano sound to be available in the second synthesizer too. You could look at all of the parameters on the first synthesizer, write them down, and then enter them one at a time into the second synthesizer. Or, you can tell the first synthesizer to arrange all of the parameters in a System Exclusive message in a pre-determined order, and send them over a MIDI cable to the second synthesizer. If the second synthesizer is set up to accept System Exclusive messages, it will receive the list of parameters, and store them in the proper order in one of its program registers. When you call up that register on the second synth, it will play the same piano sound. Because the messages describe a sound for a certain type of synthesizer, any other synth on the line that is not the same model will ignore the message completely.

This type of parameter-list exchange is called a “System Exclusive dump” or “Parameter dump”. A dump can contain the parameters of a single voice, or the parameters of *all* of the voices in a synth’s memory, in which case it’s often called a “Bulk dump”. Thus a System Exclusive message can be many hundreds or even thousands of bytes long. Many hardware synthesizer manufacturers offer computer editing software for their instruments; System Exclusive messages are how the computer and the instrument communicate with each other.

Immediately following the F0H header, a System Exclusive message contains a special Manufacturer’s ID number. These ID numbers are assigned to manufacturers by the governing bodies that control the MIDI Spec, the (US-based) MIDI Manufacturers Association and the (Japan-based) Association of Musical Electronics Industry. What follows the Manufacturer’s ID is whatever the manufacturer wants: usually the first thing is a model number or code, but it can be literally anything at all, and it can last as long as the manufacturer wants. Any device receiving the System Exclusive message that is not supposed to respond to it should ignore it completely.

At the end of any System Exclusive message is an **F7**H byte. This is called, not surprisingly, “**End of System Exclusive**”, and is sometimes abbreviated in documentation as “EOX.” When an F7H is received, all the devices on the system that had been ignoring the System Exclusive message “wake up,” and pay attention to whatever comes next. Until the F7H is received, a System Exclusive message should not be interrupted. The EOX command does not have any data bytes.

Universal System Exclusive

A sub-class of System Exclusive messages is called “Universal System Exclusive”. This may strike you as an oxymoron (like “jumbo shrimp” or “reasonable attorneys’ fees”), but it is real nonetheless. Most extensions to the MIDI Specification in recent years have fallen into this category, because it remains the most “open” part of the Spec. These extensions include:

• Experimental or “Non-Commercial”, for internal use by research and educational institutions, and not in any products that are allowedto be released to the public.

• Sample Dump Standard, which allows MIDI-based sampling synthesizers and computers to exchange data.

• Tuning Standard, which allows alternative tunings to be used in MIDI instruments.

• Some MIDI Time Code messages, which allows synchronization and control of MIDI devices.

• MIDI Machine Control and MIDI Show Control, which expand MIDI into the worlds of transport automation and live presentations.

Non-Commercial SysEx messages have an ID of 7DH, so such messages will always start with

F0 7DH. Sample Dump Standard and Tuning Standard messages fall under the classification of “Non-Real Time” Universal System Exclusive, which have an ID of 7EH, while MIDI Time Code, MIDI Machine Control, and MIDI Show Control messages are classed as “Real Time,” and have an ID of 7FH. As with all SysEx messages, at the end of any of these messages there must be an EOX byte, F7H.

MIDI Yesterday, Today, and Tomorrow

MIDI has been around for some 35 years, which is far longer than most other digital communications protocols. The people who created it were smart: they made sure there were ways that the language could be expanded and made more useful as new music and audio technologies evolved. They also designed it so it would stay relevant as new and faster methods of sending data electronically were developed.

No one could have imagined when witnessing two expensive synthesizers being connected together at a NAMM show that there would someday be ***billions*** of MIDI-enabled devices on the market. Yet there are: every smartphone, tablet, and computer has built-in MIDI capabilities that far outstrip those of the instruments the creators of MIDI had in mind, when they developed the brilliant idea of getting two synthesizers from different manufacturers in different countries to talk to each other. And now in the 21st Century, MIDI technology continues to empower musicians around the world as they seek to connect, learn, and create music.

Already the MIDI community is getting ready for the next generation of musical tools, and additions to the MIDI Specification are or will soon be in place to handle MIDI in Web applications, instruments with multiple dimensions of expression, and high-speed, high-resolution data transports. MIDI created the electronic music world we know today, and will continue to be a crucial part of it tomorrow.