

learn.sparkfun.com

MIDI Tutorial - learn.sparkfun.com

56-71 minuti

[Favorited Favorite 23](#)

Introduction

Perhaps you've seen the plug on the back of something. In today's world of micro-USB and thunderbolt connections, it's a rather large circular connector, about 1/2" in diameter, with five electrical connections. There are often two or three of these plugs in a row.



This is the **Musical Instrument Digital Interface (MIDI)** plug. Musical instruments use these ports to communicate performance data, but the protocol has also been extended to related devices, such as stage lighting and recording studio equipment.

MIDI itself is a relatively simple serial communication standard, but it can be daunting because there's a lot of terminology. In the following sections, we'll do our best to explain the terminology, while exploring the finer technical details.

Background

MIDI is built atop some concepts we've explored in more detail in other tutorials.

- MIDI transmits data using [serial ports](#).
- To make good use of the transmitted data, it's helpful to know how to convert to and from [hexadecimal](#), and use [binary operators](#).

History

To lend some context to MIDI, it helps to look at what came before, and understand the problem it was intended to solve.

Proprietary Analog Systems

The earliest commercial synthesizers were large analog systems, made up of individual modules that got plugged together to configure the sound. One classic example of this sort of synth is the Moog Modular.



The Moog System 35 Modular Synthesizer (Photo Courtesy [Moog Music](#))

On these systems, the configuration of the patch cables, knobs, and switches determined what sort of sound was created. Even today, the configuration for a sound on a synthesizer is called a **patch**, though it might not involve any cables or plugs.

The signals between the modules were all analog, using voltages to represent musical parameters. One voltage might control the pitch, another the timbre, and a third the amplitude. Within the system, the signaling between the modules was standardized, so all of the modules were intercompatible.

The voltage compatibility didn't always translate to other vendors. Each manufacturer implemented their own variant of the analog interface, tailored to their specific needs. There was little agreement as to how the interface worked. While instruments could be connected, there was no guarantee that they would respond the same way. There were simply too many permutations to allow easy communication.

In the late 1970's, Herbie Hancock was exploring the possibilities of synthesizers, and had a live rig that incorporated instruments from many different manufacturers. He wanted to be able to play any of them from a single custom keyboard.





The Back Cover of Herbie Hancock's 1978 record Sunlight

Bryan Bell, Herbie's sound technician, designed and built a [system to interface](#) those synths, as well as manage storing and loading patches. It was [expensive and primitive](#), but it proved the value of the underlying concepts.

Introducing Digital Systems

By the end of the 70's, many synthesizers contained a digital microprocessor, controlling the analog circuitry, and implementing features such as automatic tuning, pattern generation (arpeggiation and sequencing), and sound storage. Roland introduced the short-lived Digital Control Bus (DCB) to allow interconnection of their synthesizers, and DIN-SYNC to synchronize drum machines, but neither was widely adopted.

In the early 1980's, following on the concepts of DCB and DIN-SYNC, Roland joined with manufacturers, including Sequential Circuits, and Oberheim, to co-develop a new standard for connecting synthesizers. MIDI was introduced at the 1983 NAMM trade show, where a Sequential Prophet 600 interfaced with a Roland Juno 106.

MIDI gained popularity as the personal computer caught on. It was state of the art in 1984, when the Apple II and Commodore 64 were the height of home computing. As technology raced on, digital audio has become commonplace, and MIDI has been adapted and extended several times. While most modern computers lack an actual MIDI port, USB is easy to convert to MIDI with an adapter.

MIDI Devices

Before we get into the details of the protocol, let's look at some of the devices that incorporate MIDI and introduce the terminology surrounding them.

Keyboard Synthesizer

The most obvious MIDI device is the ordinary keyboard synthesizer. For the purposes of this tutorial, we'll be calling anything that can generate sound a **synthesizer** or **tone generator**. From a general perspective, the specific method used to generate the sound is unimportant, and there are many subtle distinctions that differentiate between instruments.





Kurzweil K2000 Synthesizer (Image courtesy [wikimedia commons](#))

The keyboard synthesizer typically has a piano keyboard (called the **controller**), and an internal tone generator, which could be analog, digital, or even mechanical. The player presses a key, and the synth produces a sound in response.

Depending on the underlying technology, a synthesizer can be:

- **Monophonic** - capable of playing a single note at a time, like woodwind or brass instruments.
- **Polyphonic** - capable of playing multiple notes simultaneously, like a guitar, piano or organ. The number of simultaneous sounds is often limited, specified as the number of **voices** it can generate.
- **Multitimbral** - a polyphonic synthesizer that is capable of playing different sounds simultaneously. Multitimbral instruments often have different modes for using these sounds:
 - **Layering** simply lets the performer play the multiple sounds at the same time, in unison.
 - A **keyboard split** assigns different voices to different regions of the keyboard. One example is a string bass on the left side of the keyboard, and a piano on the right side.
 - A **velocity split** changes sounds depending on how hard the keyboard is struck. A soft touch could result in a [glass harmonica](#) sound, while a harder hit might elicit a [pyrophone](#).

The controller and sound generator functions don't need to be bundled in the same unit, though. They could also be separate components.

Standalone Sound Generators

Standalone MIDI sound generators, also known as **sound modules**, are instruments that generate sound, but have no on-board provision for playing. They can take many forms, the two most common might be the tabletop module





Moog Minitaur Module(*image courtesy [Moog Music](#)*)

and the rack-mount unit.



Roland D550 Rackmount Synthesizer (*Image courtesy [wikimedia commons](#)*)

One other type of synthesizer we'll include in this category is the **virtual** or **software synthesizer**, a piece of software running on a computer. There are many virtual synthesizers available today, including emulations of classic, vintage instruments, and complex types of sound generation that might be impractical as dedicated standalone devices.

While the shapes differ, they are functionally equivalent. They receive commands on their MIDI input, and respond by generating sounds. They are usually used with a computer sequencer or an external MIDI controller.

Standalone Controllers

The flip-side of the standalone sound generator is the standalone MIDI controller. The most common is probably the *MIDI keyboard controller*.



This offers a set of piano keys, sometimes grouped with extra knobs, wheels or levers. It doesn't generate any sound on its own, instead sending MIDI messages to an external tone generator.

MIDI controllers have evolved significantly in recent years, particularly as virtual instruments have proliferated. A MIDI controller allows the player to access settings using physical controls. They can turn a physical knob instead of reaching for the mouse. A wide variety of controllers are now available,

reflecting many different musical applications. These include electronic drumsets, saxophones, mixing consoles, and DJ turntables.



A keyboard controller and a tabletop drum-pad controller

Drum Machine

A drum machine contains two major elements: a sound generator, and a sequencer.



The Roland TR-808 drum machine (Image courtesy [wikimedia commons](#))

The sound generator portion is similar to those described above, a synthesizer tailored to making drum and percussion sounds.

The **sequencer** is used to capture and play back drum beats, triggering the associated sound generator. Some sequencers are read-only, only playing back prerecorded rhythms -- remember the "cha-cha", "tango" and "merengue" buttons on your grandparents' parlor organ? Most drum machines are programmable, allowing the user to create their own rhythms, although the specific programming interface varies widely.

Computers

MIDI has grown up alongside the personal computer, and PCs frequently feature MIDI interfaces. MIDI was hidden in the [15-pin joystick interface](#) on older PC soundcards, broken out to 5-pin DIN connectors with a pigtail adapter. Of course, the 15-pin joystick plug has disappeared, and joysticks now connect via USB. Conveniently enough, USB also defines a [MIDI device class](#), so MIDI is available on modern computers.

We mentioned virtual instruments [above](#), but there are many other types of MIDI application for the PC. Whether it's connected with an actual MIDI port, or via USB, the personal computer is capable of running programs to serve a number of different MIDI functions.

- Like the drum machine, a **sequencer** program records MIDI messages and plays them back, possibly also allowing the user to edit and rearrange the MIDI data. Sequencers take many forms, from the very simple to the extremely sophisticated.
- **Notation programs** allow the user to compose music by writing in musical notation on the staff, then print the resulting scores.
- A **patch editor** serves as an onscreen substitute control panel for a synthesizer. Editors are especially popular for instruments with only a small LCD for accessing the internal parameters, as the PC GUI can display the sonic parameters much more clearly.
- The **librarian** is used to load, save, and organize the sounds on an instrument.

These features may all be combined in a single application. A modern sequencer might also host virtual instruments, be able to edit and organize patches, as well as record and edit MIDI and digital audio.

Other Applications

MIDI sometimes pops up on devices that aren't strictly musical instruments. One common application is controlling stage lighting, although large light systems are more commonly controlled via the [DMX-512](#) serial protocol.

Musicians are also a creative bunch of people. There are [many examples](#) of people building unique instruments, enabled by the ubiquity of MIDI.

The MIDI Standard

The MIDI standard is maintained by the MIDI Manufacturers Association (MMA), who have an online presence at [MIDI.org](#). The MIDI specification itself is a printed document, available [from the MMA](#). This is a substantial document, which provides in-depth descriptions of many aspects of the protocol.





If you're casually experimenting with MIDI, there are plenty of [reference materials](#) available online. However, if you're serious about making MIDI instruments, this document is worth having. Joining the MMA as a manufacturer also grants you a System Exclusive ID code.

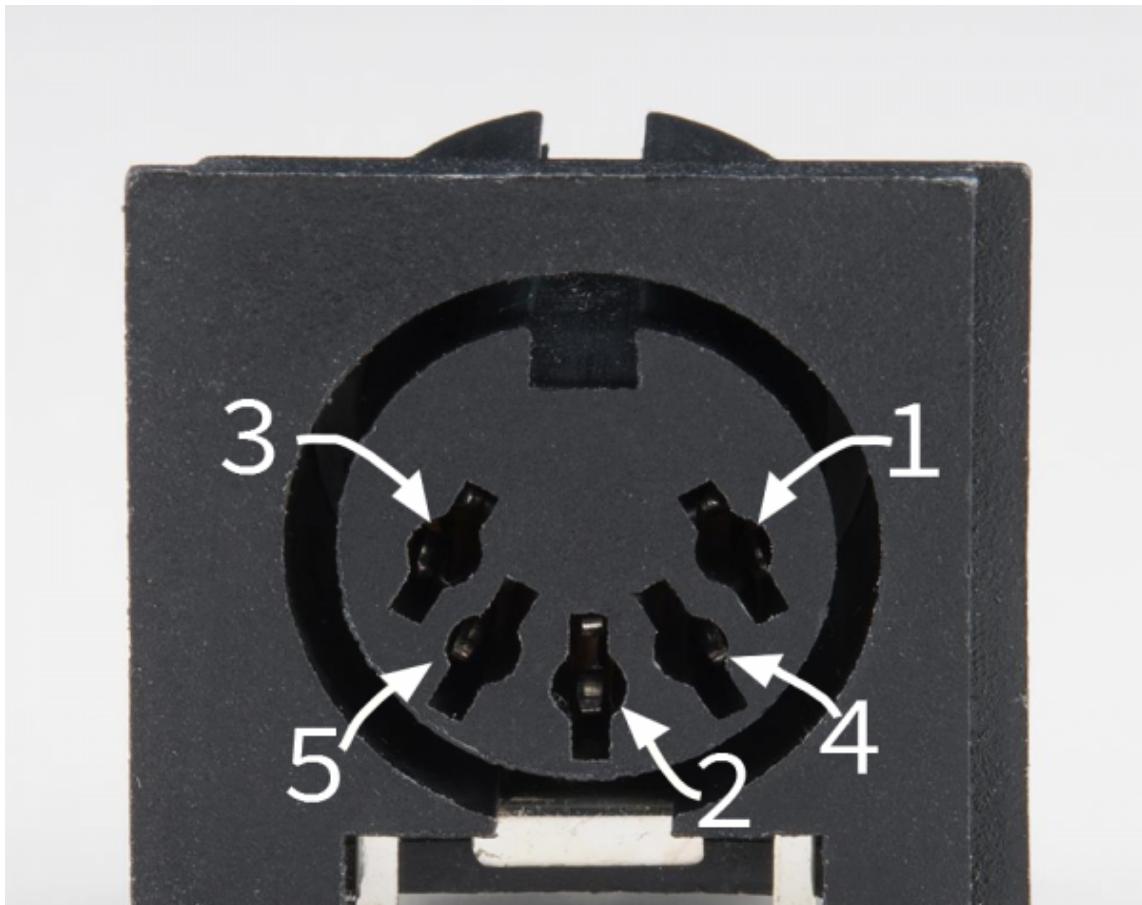
Hardware & Electronic Implementation

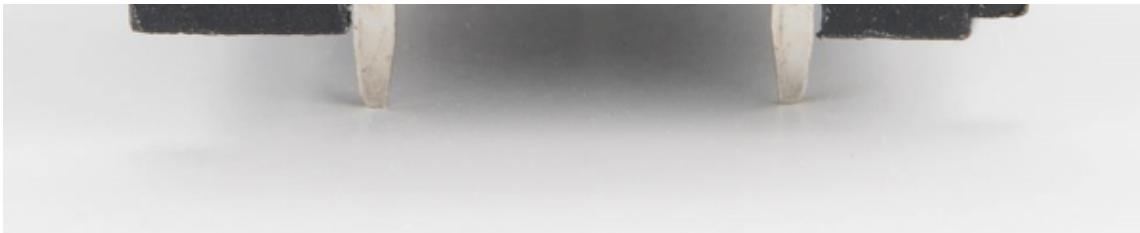
MIDI Hardware

One of the design goals of MIDI was that it needed to be relatively inexpensive. When MIDI was created, most electronic instruments were already built around a microprocessor system, so MIDI was defined as a micro-to-micro digital communication buss. One goal was that the hardware for the interface would only increase the bill of materials cost by about \$5, which, by comparison to the rest of the parts, was an insignificant cost.

Physical Implementation

Since electronic musical instruments already feature a number of plugs and connections, MIDI specifies one that wasn't commonly in use at the time: the circular 5-pin DIN connector. A unique connector was selected so things couldn't be misconnected.





Note that the pins on the connector are numbered out of order -- it's as if two more pins were added between the pins of a 3-pin connector. To help keep it straight, the numbers are frequently embossed in the plastic of the connector.

To plug into that connector, you need a MIDI cable.



While the connectors have five pins, only three of them are used. If you want to wire your own MIDI cable, you'll need two male, 5-pin DIN connectors and a length of shielded, twisted-pair (STP) cable.

A MIDI cable is connected as follows:

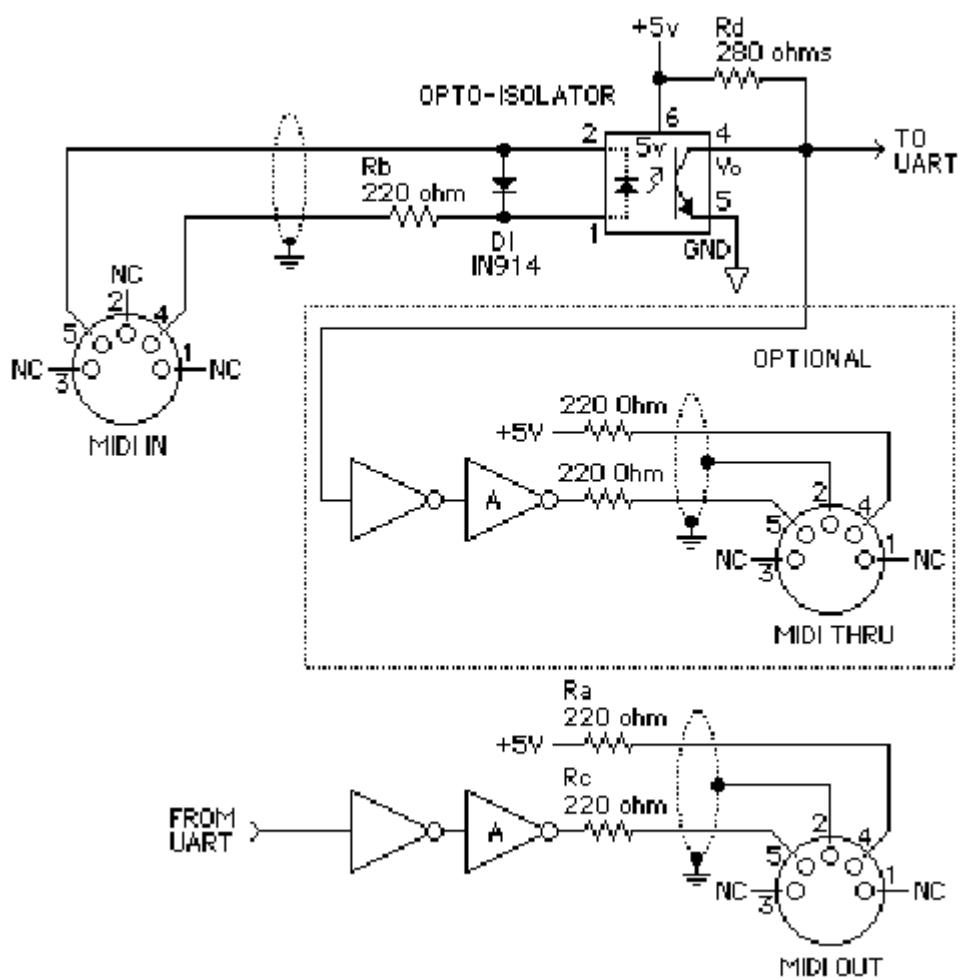
MIDI Cable Wiring		
First Connector	Cable	Second Connector
Pin 1	No Connection	Pin 1
Pin 2	Shield	Pin 2
Pin 3	No Connection	Pin 3
Pin 4	Voltage Reference Line	Pin 4
Pin 5	Data Line	Pin 5

The spec defines a maximum cable length of 50 feet (15 meters).

Note: If you're looking for a 5-pin DIN cable with all five pins connected, you don't want a regular MIDI cable, as there's no guarantee that all of the pins are connected. You need a 5-Pin DIN cable. In a pinch, you can substitute a 5-pin DIN cable for a MIDI cable, but not the other way around.

Electronic Implementation

The MIDI spec contains a recommended circuit for the ports, illustrated below.



Copyright 1985 MIDI Manufacturers Association

MIDI Standard Hardware (Courtesy [MIDI.org](https://www.midi.org))

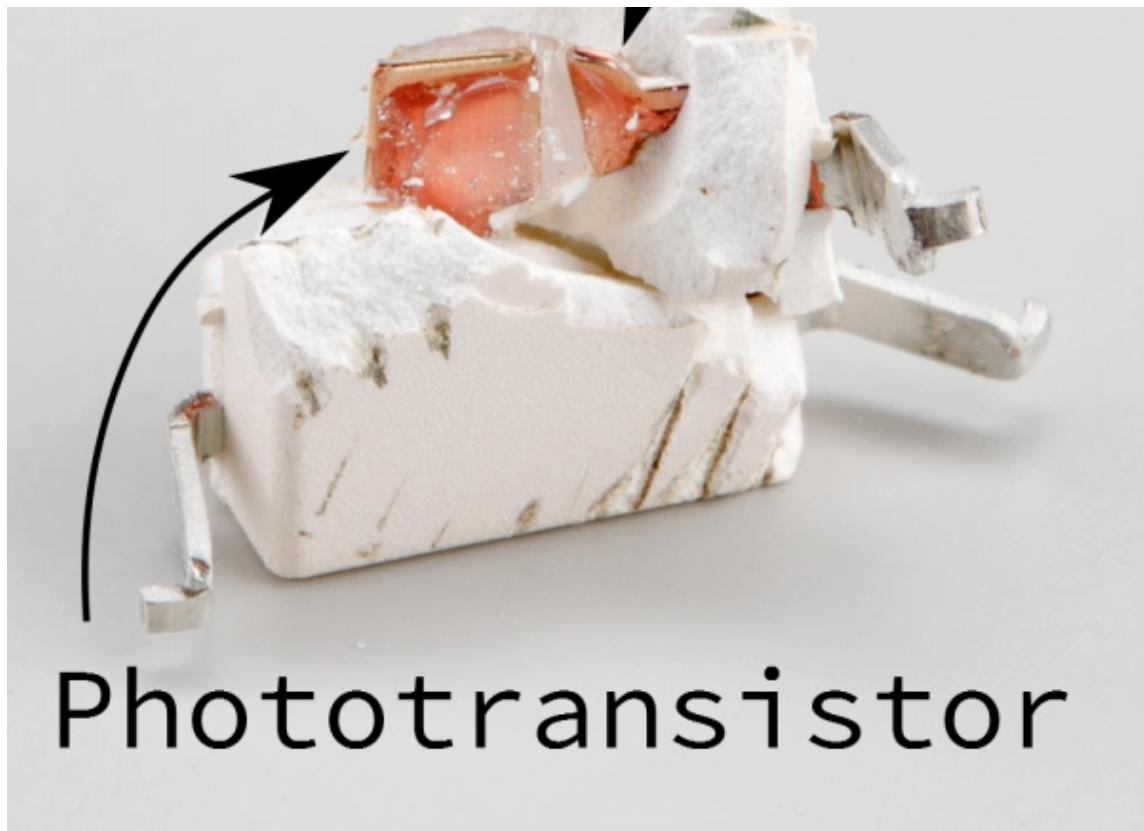
There are three sections to this circuit.

MIDI Input

The top portion of the schematic is the MIDI input. It's the most sophisticated part of the circuit, because it calls for an opto-isolator. *The reference design specifies the long obsolete Sharp PC-900; modern designs frequently use the 6N138.*

[Opto-isolators](#) are an interesting component, frequently used in interface circuitry. They allow data to be transmitted without forming an electrical connection. Internally, the incoming signal blinks an LED. The light is picked up by a phototransistor, which converts the signal back into an electronic signal. The LED and phototransistor are physically separated by a short distance. In the photo below, the gap is filled with translucent plastic.





Phototransistor

Opto-isolator internals

We'll explain more about how the optoisolator works [below](#), but first we need a MIDI output to connect to it.

MIDI Out

At the bottom of the schematic is the transmitting hardware, behind the MIDI Out jack. The hardware is relatively simple -- the UART transmit line simply runs to a pair of logic inverters. On older microcontrollers, the I/O pins were relatively weak, and couldn't source or sink enough current to light the LED in the optocoupler. The pair of inverters are an inexpensive way to increase the signal drive strength.

Modern microcontrollers, like the Atmel AVR, have much more robust pin circuitry. They are capable of lighting the LEDs directly, but the buffer is still sensible. Because the connector goes to the outside world, it's possible that it could be shorted, connected incorrectly, or experience an [ESD](#) event. The buffer is a piece of inexpensive sacrificial circuitry, akin to a fuse, which can be replaced more easily than the processor.

MIDI Thru [sic]

At the center of the schematic, within the dashed line, is the MIDI thru port. You'll notice that it's effectively a copy of the MIDI out port, but it's connected to the data line on the MIDI input -- it transmits a copy of the incoming data, allowing multiple instruments to be daisy-chained. We'll go into more detail about its usage in the [topologies](#) section.

MIDI Ports In Practice

Not every MIDI device includes all of the ports. Some devices only need to transmit or receive, only requiring a MIDI Out or In, respectively.

The MIDI thru port is also optional, sometimes omitted to reduce size or cost.

The implementation of the MIDI thru port is also subject to interpretation. While the spec calls for the hardware implementation shown above, it isn't always followed. Some instruments use a second UART to re-transmit bytes from the input. This adds latency, which can cause timing problems in long daisy chains.

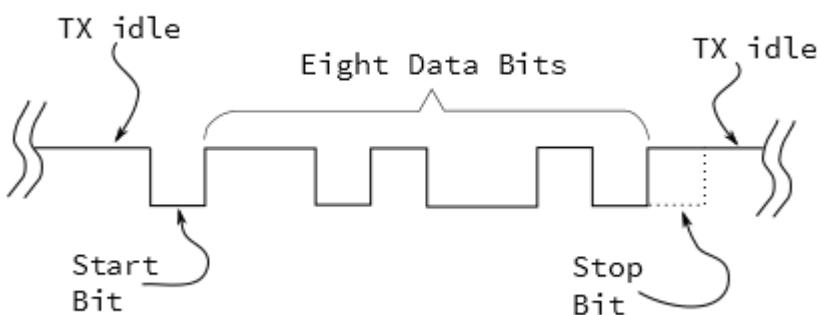
Logical Signaling

There are two signals in the schematic above that leave the page, marked "TO UART" and "FROM UART."

UART stands for "Universal Asynchronous Receiver/Transmitter". It is a piece of digital hardware that transports bytes between digital devices, commonly found as a peripheral on computer and microcontroller systems. It is the device that underlies a serial port, and it is also used by MIDI.

If you're not familiar with the basics of UARTs, you can come up to speed with our [Serial Communication](#) tutorial.

The UART signals in the schematic above are at logic levels. When it is idle, it sits at a logic high state. Each byte is prefaced with a start bit (always zero), followed by 8 data bits, then one stop bit (always high). MIDI doesn't use parity bits.



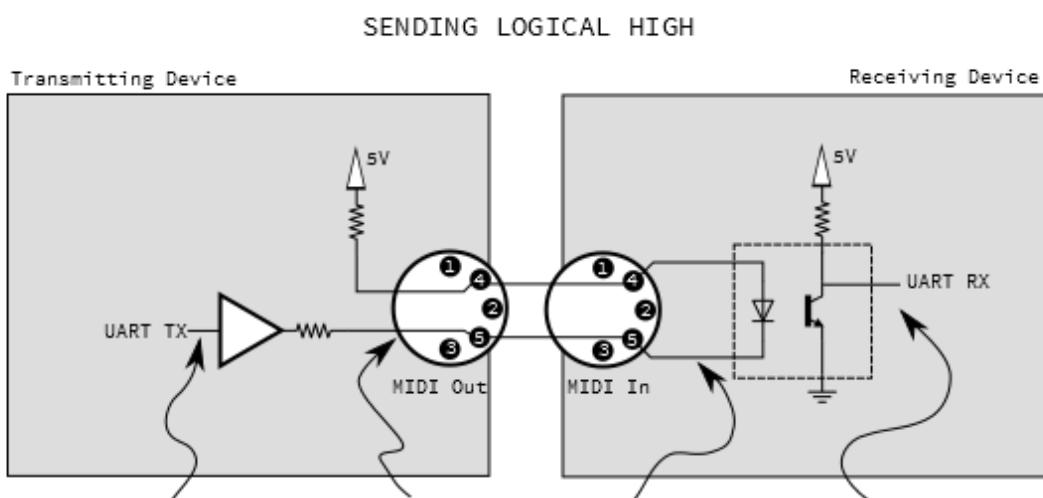
MIDI uses a clock rate of 31,250 bits per second. To send an 8-bit byte, it needs to be bookended with start and stop bits, making ten bits total. That means a byte takes about 320 microseconds to send, and the maximum throughput on a MIDI connection is 3,125 bytes per second. The average MIDI message is three bytes long, taking roughly one millisecond to transmit.

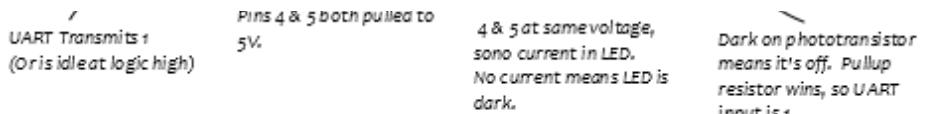
When MIDI was new, most synthesizers used discrete, external UART chips, such as the [16550](#) or the 8250. UARTs have since moved into the microcontroller, and they are a very common peripheral on AVR, PIC and ARM chips. UARTs can also be implemented in firmware, such as the [Software Serial](#) library for Arduino.

We'll go into much more detail about what the bytes mean and how to interpret them in the [next section](#).

From Output to Input

Before we move on to the messaging portion of the protocol, let's analyze the circuit formed when an output is plugged into an input. Below we see a simplified diagram, showing an output port connected to its corresponding input.

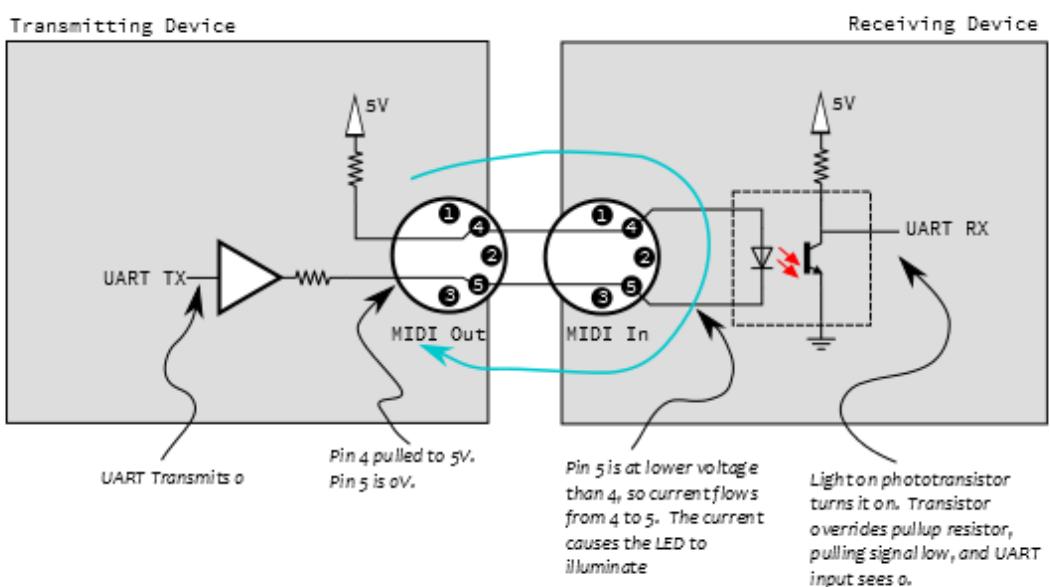




At the output, pin 4 is pulled high through a small resistor, and pin 5 is the buffered UART transmit line. On the input, these lines are tied to the anode and cathode of the LED in the opto-isolator. Since the UART output is high when not transmitting, both pins 4 and 5 will be at the same voltage, no current flows through the LED, thus it is not illuminated. When the LED is dark, the phototransistor is off, and the UART receives the voltage through the pullup, resulting in a logic one.

When the UART starts transmitting a byte, the start bit will pull pin 5 low, and current flows through the LED, illuminating it. When the LED is on, the phototransistor turns on, swamping the pullup, pulling the UART input to ground, and signaling a zero.

SENDING LOGICAL LOW



We should note here that the LED is electrically a part of the transmitting circuit. Current flows out of the transmitter, through the LED, and back to the transmitter, forming a *current loop* (illustrated in blue, above). There is no actual electrical connection between the transmitter and receiver, only the optical path inside the optoisolator. This is useful in helping to avoid [ground loops](#).

Messages

As stated in earlier sections, the overarching goal of MIDI was to allow different devices to communicate, speaking a well-defined protocol. The protocol revolves around a stream of small messages. Most messages are between one and four bytes long, although some can be longer, and some information requires groups of messages.

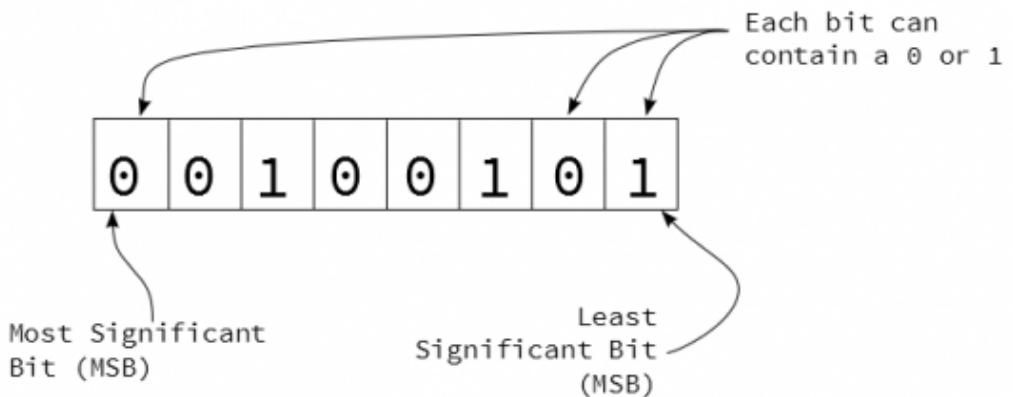
MIDI messages fall into several categories, such as performance messages ("the performer pressed the middle C key"), information about the sounds being played ("change from the piano sound to the organ sound"), and other musical data, such as metronome or real-time clocks.

A quick word about written numbers: In the following sections, we'll be using both decimal and hexadecimal notation to represent numbers. Decimal numbers will be written normally, and the hex will be prefixed with `0x`. If you're new to hexadecimal, you can learn more about it in [this tutorial](#).

Status or Data?

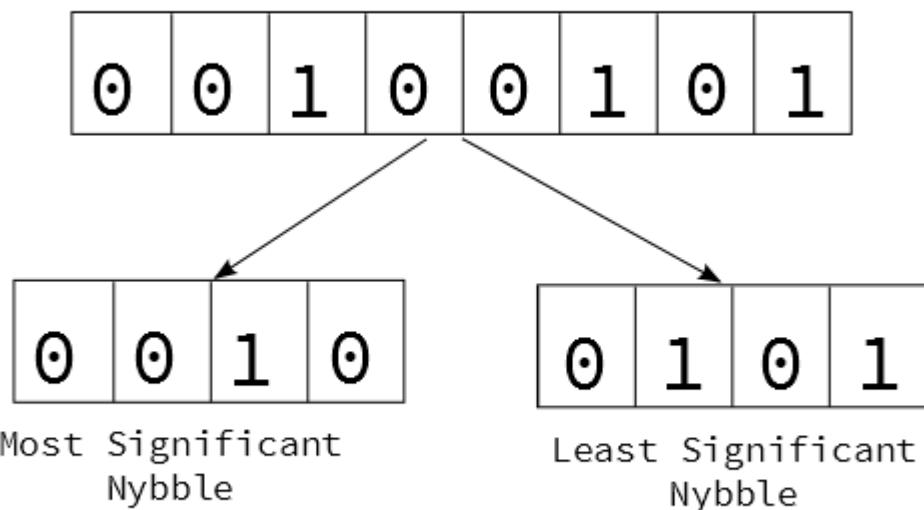
In the previous section, we discussed the UART configuration for MIDI -- 31,250 bits per second, 8 data bits per byte, and 1 stop bit. MIDI uses those 8 data bits to the fullest extent!

Within a byte, each bit may be either a 0 or 1.



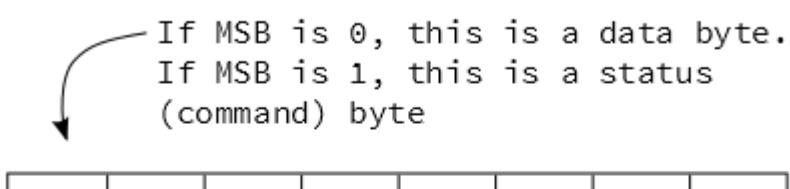
One Eight-Bit Byte

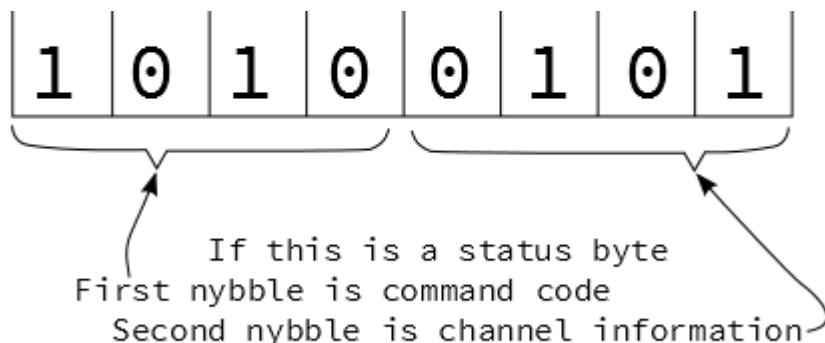
Some bytes are further divided into **nybbles**, or 4-bit chunks. Each nybble is identified by its position within the byte. When written in hexadecimal, each nybble is represented by a character. The left hand character (made up of the higher-value bits) is known as the **most-significant nybble**, and the right-hand character is known as the **least-significant nybble**.



Byte divided into nybbles

Bytes of MIDI messages are divided into 2 major categories, based on the setting of the most significant bit.





Status Byte Dissection

If the first bit is high (values between 0x80 and 0xff), it denotes a **status byte**. Status bytes are the commands of the MIDI stream.

- Status bytes are further subdivided by nybble. The first nybble specifies the command type, and the second nybble specifies which the channel the command applies to.
- There are only eight bit combinations with the MSB set (0x8 to 0xf), therefore there are eight possible status commands.
- The 4 bits in the channel nybble are all usable, giving MIDI 16 possible channels.

If the first bit is low (values between 0x00 and 0x7f), it is a **data byte**, indicating parameters that correspond to a previous status byte. Because the MSB must be zero (otherwise they'd become status bytes), the data is limited to 7-bits, or the range from 0 to 127 (0x0 to 0x7f).

Statuses & Corresponding Data

Let's look at the status bytes. We'll start with a list, then explore each in the following sections.

MIDI Status Messages					
Message Type	MS Nybble	LS Nybble	Number of Data Bytes	Data Byte 1	Data Byte 2
Note Off	0x8	Channel	2	Note Number	Velocity
Note On	0x9	Channel	2	Note Number	Velocity
Polyphonic Pressure	0xA	Channel	2	Note Number	Pressure
Control Change	0xB	Channel	2	Controller Number	Value
Program Change	0xC	Channel	1	Program Number	-none-
Channel Pressure	0xD	Channel	1	Pressure	-none-
Pitch Bend	0xE	Channel	2	Bend LSB (7-bits)	Bend MSB (7-bits)
System	0xF	further specification	variable	variable	variable

The messages with the channel number in the second nybble of the status byte are known as **channel**

messages. Channels are often used to separate individual instruments -- channel one could be a piano, two a bass, and so on. This allows a single MIDI connection to carry information for multiple destinations simultaneously. Each sound would be played by sending messages with the appropriate value in the channel nybble.

Channel numbering leads to some confusion. With 4 bits, there are 16 possible binary values, 0 through 15 (0x0 through 0xF). Since most people start counting at one rather than zero, MIDI devices commonly (but not always) internally add an offset of one to the binary value, resulting in the range 1 to 16. If you're having trouble getting a system to communicate, you might try adjusting channels up or down by one.

Since they're very useful, and easy to implement, we're going to start with two of the most common types of messages: Note On/Off and System Realtime.

Note Off (0x80), Note On (0x90)

The meaning of Note On and Off messages is reasonably obvious. When a key on a keyboard is pressed, it sends a Note On message, and it sends a Note Off when the key is released. On and Off messages are also sent by other types of controllers, such as drum pads, and MIDI wind instruments.

When a synthesizer receives a Note On, it starts generating sound; the Note Off instructs it to stop.

Note On and Off are each comprised of three bytes

0xnc, 0xkk, 0xvv

Where

- n is the command (note on (0x9) or off(0x8))
- c is the channel (1 to 16)
- kk is the key number (0 to 127, where middle C is key number 60)
- vv is the striking velocity (0 to 127)

Velocity is most commonly measured using a pair of switches under each key, which are slightly offset from each other. As the key is pressed, one switch closes before the other. By measuring the time between the two switch closures, it can determine how quickly the key was moving. Some instruments don't measure velocity, and instead transmit a fixed value for that byte, such as 0x40 (64).

We said these are simple on the surface, but there are a couple of tricky shortcuts that complicate the situation.

Running Status

In order to transmit fewer bytes, and free up some bandwidth on the connection, MIDI uses a form of shorthand. When the same status byte would be transmitted repeatedly, the protocol uses the notion of **Running Status**, where only the first status byte is transmitted, and successive messages simply transmit new data bytes. For example, three Note On messages would usually be:

0x90, 0x3C, 0x40
0x90, 0x3D, 0x40
0x90, 0x3E, 0x40

Using running status, the second and third status bytes are omitted:

0x90, 0x3C, 0x40, 0x3D, 0x40, 0x3E, 0x40

Although we're illustrating this in the context of Note On/Off messages, it can apply to *any* status. When a new status byte arrives, it replaces the previous running status.

Implicit Note Off

To make more effective use of running status, MIDI uses a Note On command with a velocity of zero as an alias for a Note Off. For example, starting with this on/off pair

```
0x90, 0x3C, 0x40
0x80, 0x3C, 0x40
```

We can change the off byte (0x80) into an on (0x90) with zero velocity

```
0x90, 0x3C, 0x40
0x90, 0x3C, 0x00
```

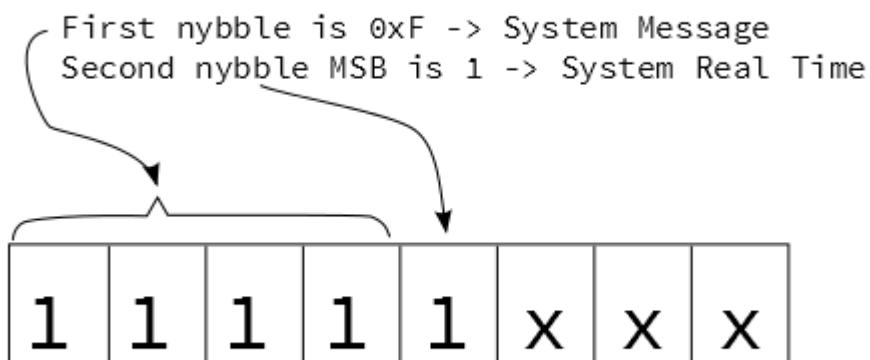
Since it is now two Note On messages in a row, we can apply running status, remove the second command, and save the transmission of one byte

```
0x90, 0x3C, 0x40, 0x3C, 0x00
```

When implicit note off is used, the implied Note Off doesn't have a velocity value. While Note Off velocity is frequently not implemented, there are times that it is an important element of a performance. In those situations, the instruments need to support it, and it might need to be explicitly enabled, often in a configuration menu.

System Real Time

Another useful category is **System Real Time** messages. These are denoted when the MSB of the second nibble of the status byte is set (values from 0xF8 to 0xFF).



These messages are all one byte long. They're mostly used to synchronize sequencers, acting as a high-resolution metronome.

System Real Time Messages		
Type	Status Byte	Usage
Timing Clock	0xF8	These messages represent metronomic timing, transmitted at a rate of 24 per quarter note. The actual rate is dependent on the tempo of the song.
Undefined	0xF9	
Start	0xFA	Start playing from the beginning
Continue	0xFB	Start playing from where last stopped

Stop	0xFC	Stop playing
Undefined	0xFD	
Active Sense	0xFE	An optional "keep alive" message that, when implemented, can be used to detect if a transmitter has been accidentally unplugged. A transmitter sends them every 300 milliseconds. If the messages stop flowing, the recipient times out, and cleans up by stopping sequences and turning off notes that would otherwise be stuck on.
System Reset	0xFF	A "panic switch" that instructs to receiver to reset to power-up condition.

Because system real time messages are one byte long, they have no data bytes, and are therefore *not* status bytes. They can be transmitted without interrupting running status.

Advanced Messages

This section of the tutorial is going to get deeper into MIDI messages. If that seems daunting, feel free to skip ahead to the [next section](#), and refer back to this page when you're ready.

Polyphonic Pressure (0xA0) and Channel Pressure (0xD0)

Some MIDI controllers include a feature known as **Aftertouch**. While a key is being held down, the player can press harder on the key. The controller measures this, and converts it into MIDI messages.

Aftertouch comes in two flavors, with two different status messages.

The first flavor is **polyphonic aftertouch**, where every key on the controller is capable of sending its own independent pressure information. The messages are of the following format:

0xnc, 0xkk, 0xpp

- n is the status (0xA)
- c is the channel nybble
- kk is the key number (0 to 127)
- pp is the pressure value (0 to 127)

Polyphonic aftertouch is an uncommon feature, usually found on premium quality instruments, because every key requires a separate pressure sensor, plus the circuitry to read them all.

Much more commonly found is **channel aftertouch**. Instead of needing a discrete sensor per key, it uses a single, larger sensor to measure pressure on all of the keys as a group. The messages omit the key number, leaving a two-byte format

0xnc, 0xpp

- n is the status (0xD)
- c is the channel number
- pp is the pressure value (0 to 127)

Pitch Bend (0xE0)

Many keyboards have a wheel or lever towards the left of the keys for pitch bend control. This control is usually spring-loaded, so it snaps back to the center of its range when released. This allows for both

upward and downward bends.



The wheel sends pitch bend messages, of the format

0xnc, 0xLL, 0xMM

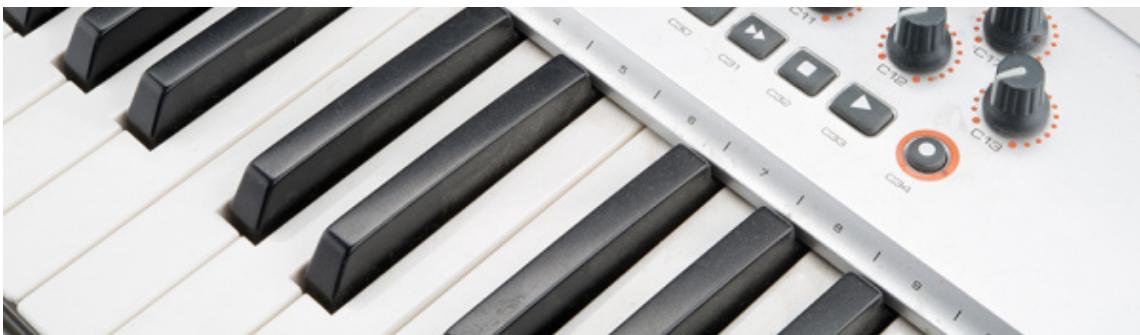
- n is the status (0xE)
- c is the channel number
- LL is the 7 least-significant bits of the value
- MM is the 7 most-significant bits of the value

You'll notice that the bender data is actually 14 bits long, transmitted as two 7-bit data bytes. This means that the recipient needs to reassemble those bytes using binary manipulation. 14 bits results in an overall range of 2^{14} , or 0 to 16,383. Because it defaults to the center of the range, the default value for the bender is halfway through that range, at 8192 (0x2000).

Control Change (0xB0)

In addition to pitch bend, MIDI has provisions for a wider range of expressive controls, sometimes known as **continuous controllers**, often abbreviated **CC**. These are transmitted by the remaining knobs and sliders on the keyboard controller shown below.





These controls send the following message format:

0xnc, 0xcc, 0xvv

- n is the status (0xB)
- c is the MIDI channel
- cc is the controller number (0-127)
- vv is the controller value (0-127)

Each of the controllers in the picture above is configured to send a different controller number.

Typically, the wheel next to the bender sends controller number one, assigned to modulation (or vibrato) depth. It is implemented by most instruments.

The remaining controller number assignments are another point of confusion. The MIDI specification was revised in version 2.0 to assign uses for many of the controllers, as shown on pages 2 and 3 of [this document](#). However, this implementation is not universal, and there are ranges of unassigned controllers.

On many modern MIDI devices, the controllers are assignable. On the controller keyboard shown in the photos, the various controls can be configured to transmit different controller numbers. The flip-side is also often true -- controller numbers can be mapped to particular parameters. Virtual synthesizers frequently allow the user to assign CCs to the on-screen controls. This is very flexible, but it might require configuration on both ends of the link and completely bypasses the assignments in the standard.

Program Change (0xC0)

Most synthesizers have patch storage memory, and can be told to change patches using the following command:

0xnc, 0xpp

- n is the status (0xc)
- c is the channel
- pp is the patch number (0-127)

This allows for 128 sounds to be selected, but modern instruments contain many more than 128 patches. Controller #0 is used as an additional layer of addressing, interpreted as a "bank select" command. Selecting a sound on such an instrument might involve two messages: a *bank select* controller message, then a *program change*.

System Messages (0xF0)

The final status nibble is a "catch all" for data that doesn't fit the other statuses. They all use the most significant nibble of 0xF, with the least significant nibble indicating the specific category.

We covered [system realtime](#) messages in the previous section. The messages are denoted when the MSB of the second nybble is 1. When that bit is a 0, the messages fall into two other subcategories.

System Common

If the MSB of the second second nybble is not set, this indicates a **System Common** message. Most of these are messages that include some additional data bytes.

System Common Messages			
Type	Status Byte	Number of Data Bytes	Usage
Time Code Quarter Frame	0xF1	1	Indicates timing using absolute time code, primarily for synchronization with video playback systems. A single location requires eight messages to send the location in an encoded <i>hours:minutes:seconds:frames</i> format*.
Song Position	0xF2	2	Instructs a sequencer to jump to a new position in the song. The data bytes form a 14-bit value that expresses the location as the number of sixteenth notes from the start of the song.
Song Select	0xF3	1	Instructs a sequencer to select a new song. The data byte indicates the song.
Undefined	0xF4	0	
Undefined	0xF5	0	
Tune Request	0xF6	0	Requests that the receiver retune itself**.

*MIDI Time Code (MTC) is significantly complex, and beyond the scope of this tutorial. If you need to know the finer points of MTC, you should probably have your own copy of the [MIDI Specification!](#)

**While modern digital instruments are good at staying in tune, older analog synthesizers were prone to tuning drift. Some analog synthesizers had an automatic tuning operation that could be initiated with this command.

System Exclusive

If you've been keeping track, you'll notice there are two status bytes not yet defined: 0xf0 and 0xf7. These are used by the **System Exclusive** message, often abbreviated as **SysEx**. SysEx provides a path to send arbitrary data over a MIDI connection. There is a group of predefined messages for complex data, like fine grained control of MIDI Time code machinery. SysEx is also used to send manufacturer defined data, such as patches, or even firmware updates.

System Exclusive messages are longer than other MIDI messages, and can be any length. The messages are of the following format:

0xF0, 0xID, 0xdd, ...

... 0xF7

The message is bookended with distinct bytes.

- It opens with the **Start Of Exclusive** (SOX) data byte, 0xF0.
- The next one to three bytes after the start are an identifier.
- Values from 0x01 to 0x7C are one-byte vendor IDs, assigned to manufacturers who were involved with MIDI

at the beginning.

- If the ID is 0x00, it's a three-byte vendor ID - the next two bytes of the message are the value.
- ID 0x7D is a placeholder for non-commercial entities.
- ID 0x7E indicates a predefined Non-realtime SysEx message.
- ID 0x7F indicates a predefined Realtime SysEx message.
- After the ID is the data payload, sent as a stream of bytes.
- The transfer concludes with the **End of Exclusive** (EOX) byte, 0xF7.

The payload data must follow the guidelines for MIDI data bytes -- the MSB must not be set, so only 7 bits per byte are actually usable. If the MSB is set, it falls into three possible scenarios.

1. An End of Exclusive byte marks the ordinary termination of the SysEx transfer.
2. [System Real Time](#) messages may occur within the transfer without interrupting it. The recipient should handle them independently of the SysEx transfer.
3. Other status bytes implicitly terminate the SysEx transfer and signal the start of new messages.

Troubleshooting Tip: Some inexpensive USB-to-MIDI interfaces aren't capable of handling messages longer than four bytes. If you're having trouble with SysEx transfers when using such an interface, it might be useful to inspect the messages on the bus with an application like [MIDI Ox](#).

Implementation chart

With all of these different messages, MIDI has become somewhat dialectic -- not every unit implements every message. A simple controller-to-synthesizer link might only need to use note on and off messages, while an on-stage pyrotechnic controller might completely ignore note messages, requiring time code, and specific SysEx commands to arm the smokebombs and flashpots.

To help the user understand what messages a specific device uses, the manual often includes a table that indicates what it can send and receive.

Function		Transmitted	Recognized	Remarks
Basic Channel	Default Changed	1 — 16 1 — 16	1 — 16 1 — 16	
Mode	Default Messages Altered	3 X X	1 X X	
Note Number	True Voice	X X	0—127 0—127	
Velocity	Note On Note Off	0 X	0 X	
After Touch	Keys Ch's	0 0	0 0	
Pitch Bend		0	0	
Control Change	0-127	0	0	
Program Change	True #	0 0—127	0 0—127	

System Exclusive		O	O	
Song Pos		O	O	
Song Sel		X	X	
Tune		X	X	
Clock		O	O	
Realtime Commands		O	O	
Local On/Off		X	X	
All Notes Off		O	X	
Active Sense		X	X	
Reset		X	X	
GM On		X	X	
Notes:				

Mode 1: OMNI ON, POLY Mode 2: OMNI ON, MONO O: YES

Mode 3: OMNI OFF, POLY Mode 4: OMNI OFF, MONO X: NO

AKAI MPC500 MIDI Implementation, courtesy Akapro.com

The chart above shows a typical implementation chart for an instrument. It tells us some notable things about the MIDI dialect spoken by this device:

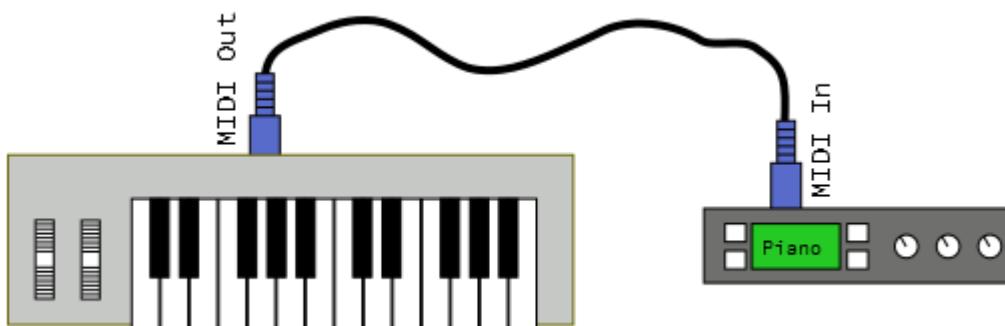
- This device transmits Note On velocity, but not note Off velocity.
- It also uses polyphonic and/or channel aftertouch (you can select which in a configuration menu).

Topologies

Now that we've looked at the types of devices that offer MIDI, and the messages they use, let's look at some of the ways they can be deployed.

Case #1: Daisy Chain

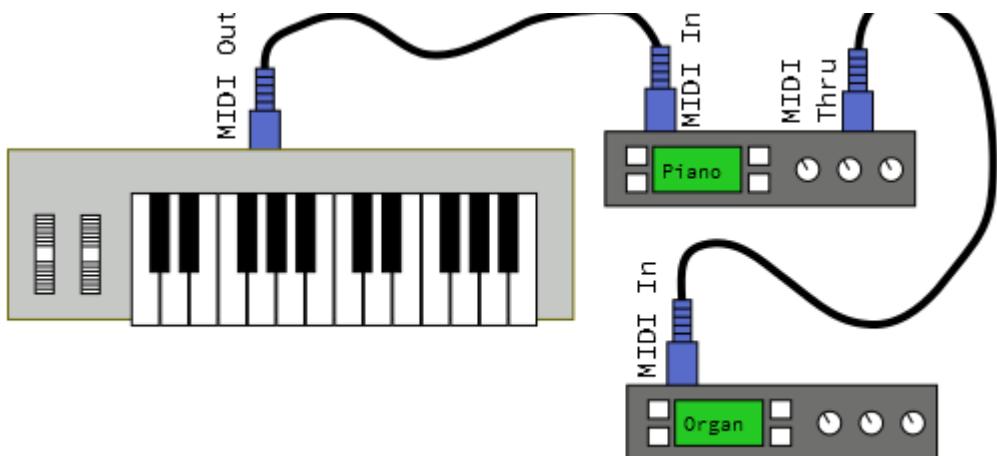
The simplest connection topology is the daisy chain, where one transmitter is connected to one or more receivers.



In this example, the MIDI out of a controller is connected to the MIDI in of a tone generator module. It allows the player to use the keys on the controller to play sounds from the module. The controller sends note-on, note-off and controller messages, to be interpreted into sound by the module. The communication path is unidirectional.

If the module is multitimbral, it can be set to respond on several MIDI channels, allowing the player to switch between sounds by changing the transmission channel.

We can add more downstream modules using the thru ports on the interceding devices. Thru transmits a copy of the messages received by the in port.

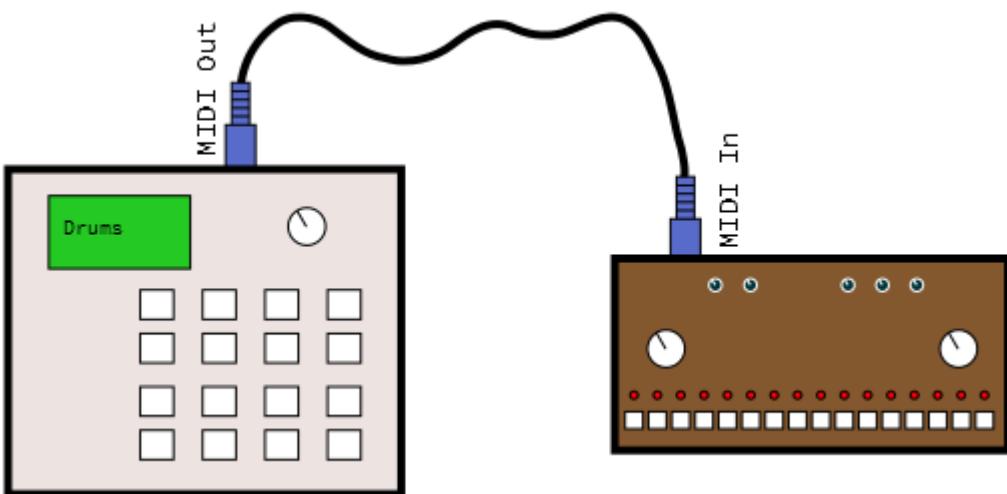


Depending on how the controller and modules are configured, there are several possibilities.

- The modules could simply all respond together, in unison.
- Each module could be set to respond on a different MIDI channel. By changing the channel that the controller sends, the modules can be played individually.
- Each module could be configured to respond to particular key or velocity ranges, forming splits and layers.

Case #2: Synchronization

In this example, we want to build a percussion ensemble by connecting drum machines together.



The drum machine on the left will serve as the master clock, and the one on the right is set to receive clock messages (sometimes called **clock slave** or **external sync** mode). When the operator presses "play" on the left machine, both machines will start playing. As the tempo of the left machine is adjusted, both machines accelerate and decelerate together.

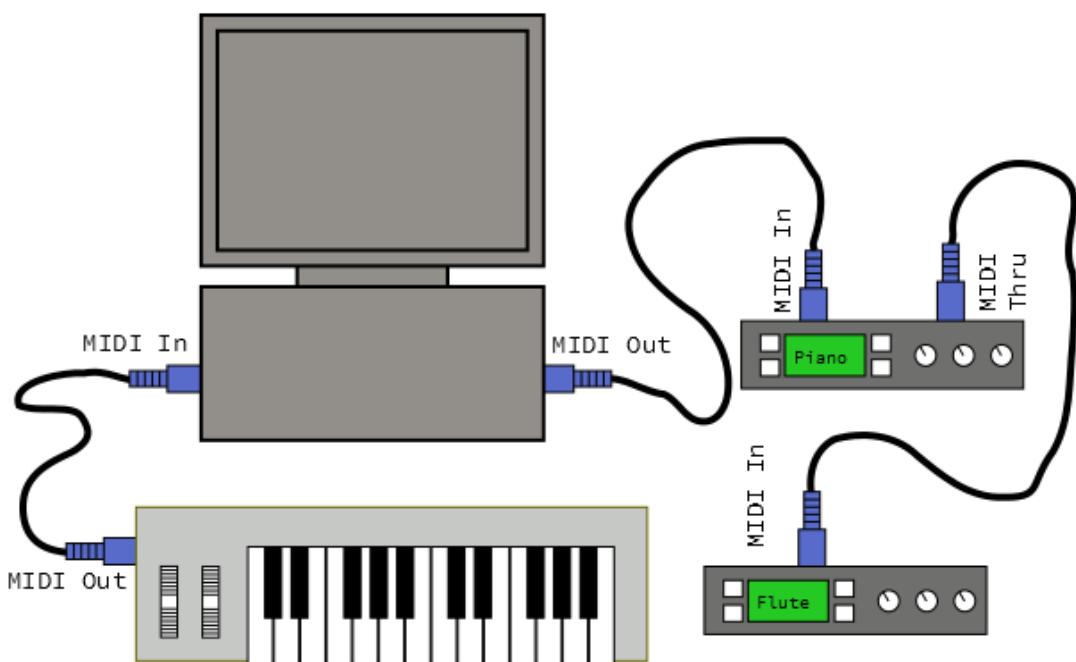
System realtime messages keep the machines in sync.

- A start (0xFA) byte is sent to initiate playing, from the beginning of the sequence.
- Clock (0xF8) bytes are sent regularly, serving as a shared metronome.
- A stop (0xFC) byte halts playback.
- Continue (0xFB) can be used to start the sequence from where it was last stopped.

Clocks may be transmitted while stopped -- this allows tempo indicator LEDs to continue flashing at the correct rate.

Case #3: Computer Sequencer

The last topology we'll look at adds a personal computer in the middle of a daisy chain.



The MIDI controller keyboard is connected to the computer, and the sound generators are connected downstream from the computer. Adding a computer in the middle of the chain allows a lot of power and flexibility. MIDI sequencer software allows the user to record performances from the controller, play them back on the sound modules, and edit or arrange them into songs. Sequencers usually have **tracks**, which allow multiple performances to be layered. The construction of such a layered performance is known as **overdubbing**.

While the sequencer is recording, it receives note on and off messages from the controller. It stores them in memory, simultaneously transmitting them to the sound modules, so the performer can hear what they are playing.

During playback, the MIDI controller is unused, and the computer plays the recorded messages, again triggering the tone modules.

Implementing MIDI

Working with MIDI messages directly is much easier if you have a strong grasp on using [binary operators](#) to manipulate bytes. You can use bitwise AND and OR operations to mask off the status bit, and extract the channel nybble value. Bitwise-shifts are useful to manipulate the 14-bit bender data.

The code snippets below were written for Arduino. You'll notice that bytes are transmitted using the `Serial.write()` method, which transmits the raw binary data, rather than converting it into a string like `Serial.print()` does. We're also using the Arduino-defined `byte` data type, which is defined as an unsigned 8-bit value. This allows us to properly access the status bit, whereas the `char` type is signed -- doing binary operations on signed values sometimes leads to subtle errors in the manipulating the MSB.

The software examples below are intentionally incomplete, as we'll introduce a [more complete and user-friendly option](#) below.

Message Generation

To send a MIDI message, simply send the appropriate bytes in the right order.

System Realtime messages are the easiest to send, just feed the byte to the output.

```
void send_sys_rt(byte rt_byte)
{
    // We are assuming that the input is in a valid range:
    // System RT bytes are between 0xF8 and 0xFF.
    //
    // More realistic code might validate each input and return an error if out
    of range

    Serial.write(rt_byte);
}
```

Channel messages are a little more complex. To form the first byte, you'd set the MS nybble to the status value, then subtract one from the channel, and put that value in the channel nybble. Send the first byte, then follow it with the appropriate number of data bytes. This snippet also includes running status -- the sender keeps track of the last status byte sent in a global variable.

```
static byte last_status = 0;

void send_channel_message(byte command,
                           byte channel,
                           int num_data_bytes,
                           byte * data)
{
    // We are assuming that all of the inputs are in a valid range:
    // Acceptable commands are 0x80 to 0xF0.
    // Channels must be between 1 and 16.
    // num_data_bytes should be either 1 or 2
    // data should be an array of 1 or 2 bytes, with each element constrained
    // to the 0x0 to 0x7f range.
    //
    // More realistic code might validate each input and return an error if out
    of range

    byte first;

    // Combine MS-nybble of command with channel nybble.
    first = (command & 0xf0) | ((channel - 1) & 0x0f);

    if(first != last_status)
    {
        Serial.write(first);
        last_status = first;
    }

    // Then send the right number of data bytes
    for(int i = 0; i < num_data_bytes; i++)
    {
        Serial.write(data[i]);
    }
}
```

Finally, sending system exclusive messages requires sending an identifier and data payload that are both of variable length. It must begin and also end with the proper status bytes.

```
void send_system_exclusive(int num_id_bytes,
```

```

        byte * id,
        int num_data_bytes,
        byte * data)
{
    // We are assuming that all of the inputs are in a valid range:
    // num_id_bytes should be either 1 or 3
    // num_data_bytes is not constrained, but a practical limitation might be
1000 bytes.
    // Id and data values should be constrained to the 0x0 to 0x7f range.
    //
    // More realistic code might validate each input and return an error if out
of range

    // send start-of-exclusive
    Serial.write(0xF0);

    // Then send the identifier bytes
    for(int i = 0; i < num_data_bytes; i++)
    {
        Serial.write(data[i]);
    }
    // Then send the right number of data bytes
    for(int i = 0; i < num_data_bytes; i++)
    {
        Serial.write(data[i]);
    }

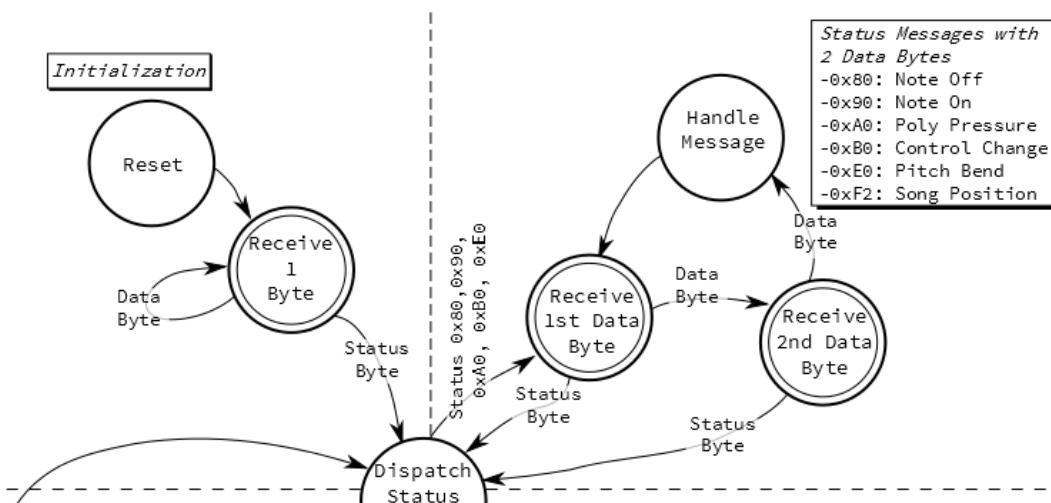
    // send end-of-exclusive
    Serial.write(0xF7);
}

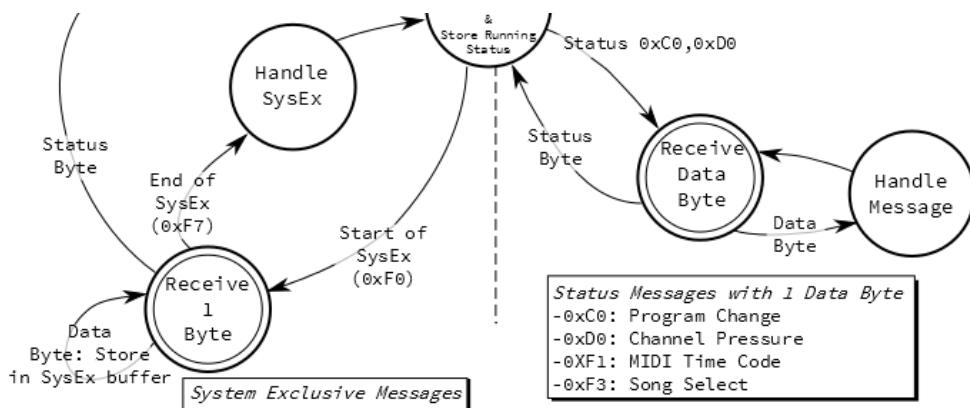
```

Message Parsing

Receiving and parsing the incoming MIDI stream requires more sophistication. While features like [running status](#), [implicit note off](#) and active sense are optional from the sender, the receiver needs to be prepared to deal with them.

A simplified [finite state machine](#) for parsing MIDI is shown below.

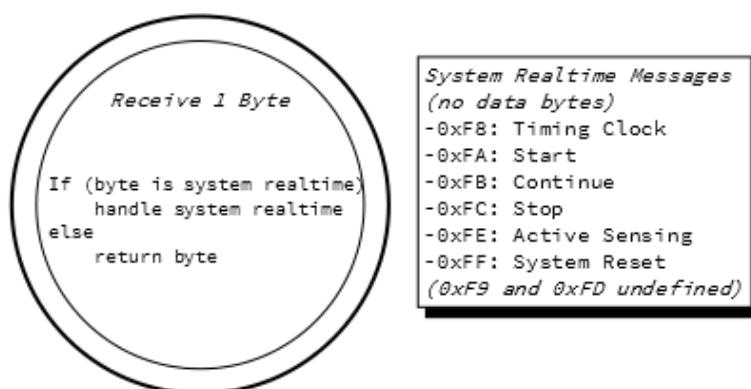




There are a few things to note

- The system initializes and starts receiving bytes. Until it sees a status byte, the input is meaningless and is discarded. Realistically, status bytes occur fairly frequently, so this state is a temporary condition.
- The "Dispatch Status" state decides how to handle messages based on how long they are.
- Most status messages have one or two bytes of data.
- System Exclusive messages can be any length, with the end marked by a "end of exclusive" byte.
- Running status is handled by the Dispatch Status state, in conjunction with the status handling branches.
- Dispatch Status stores the status byte, so it can be recalled when running status is used.
- When the one-byte and two-byte branches of the FSM see repeated data bytes, they repeatedly apply the stored status.
- At any point that the FSM is expecting a data byte, a new status could arrive instead. The previous status is discarded, and the new status dispatched instead.

You'll notice that all of the "receive bytes" states are illustrated with a double-circle. This is shorthand for more sophisticated behavior. As we discussed previously, [system realtime bytes](#) can occur at any point. We could be expecting the velocity byte of a Note On message and get a clock byte instead. The clock is allowed to non-disruptively interrupt the parsing. The details of the receive states are shown below.



From the perspective of the parsing FSM, system realtime bytes don't count as data or status bytes, they just get handled as they arrive.

The diagram shown above illustrates how to decode the bytes in the MIDI stream, but it doesn't make any implication about how the receiver will respond to them. The verb "handle" is intentionally vague, as each type of device might respond differently.

Arduino Library

The FSM above is pretty complicated. Writing and debugging it means developing a pretty serious piece of software.

But there's an easier way. Github user FortySevenEffects has written an [Arduino MIDI library](#) that makes reading and writing MIDI messages much easier. The library is flexible, and it can be configured to fit different applications.

- It can use hardware or software serial ports (leaving the hardware serial port for printing debug messages!).
- Incoming messages can be received by polling, or callbacks for specific messages can be installed.
- The library can filter for messages on a specific midi channel, or receive on all channels.
- It implements an optional "soft thru" port that can be configured to echo the input back to the output port.

It also has [detailed documentation](#) in [doxygen](#) format.

For some practical examples that demonstrate this library, take a look at the [hookup guide](#) for the [SparkFun MIDI Shield](#).

Devising Your Own Implementation Chart

In the messages section, we discussed the [MIDI implementation chart](#). If you're implementing a MIDI device, you should consider writing your own chart, so you know what messages to handle. Most devices won't implement every message, and for messages it does implement, it may still choose to ignore some. For instance, a tone generator will handle note on and off messages, but only those on its MIDI channel. For statuses that are not needed, you can simply discard the messages.

If you elect to discard messages, it's advisable that you still handle the System Reset message (0xff) - it's used as a "panic switch" in situations that you need things to be silent, ASAP!

Other MIDI Technologies

In addition to the communication protocol, MIDI also specifies some related technologies, with the goal of achieving greater intercompatibility. MIDI has been extended several times, to clarify parts of the messaging protocol, and to help standardize areas related to electronic musical instruments. While not an exhaustive list, we'll discuss a couple of the common extensions below.

Standard MIDI Files

As MIDI sequencers caught on, people wanted to be able to move their song data between different sequencers. Each sequencer implemented a proprietary file format -- some were able to import data from others, but it was far from universal.

The Standard MIDI File was specified as the *lingua franca* for sequence data interchange. The files use the SMF extension.

SMF has been widely adopted, across many operating systems. Media player programs can play them back, while full-fledged sequencers can import, edit and export SMFs.

General MIDI

Moving sequence data between platforms brings along a related problem: the MIDI communication protocol does not specify how instruments interpret patch changes, associate instruments with MIDI channels, nor implement multtimbrality. General MIDI adds some standardization to those parameters, so that different

instruments will respond similarly.

General MIDI defines a standard sound set -- a mapping of program change commands to specific sound patches. On all MIDI channels except 10, Program #1 would always be a grand piano patch, #2 a "bright piano", and so on.

Channel 10 is reserved for drums, and the mapping of drum sounds to the keyboard was also defined.

General MIDI also specifies that a supporting instrument need to simultaneously allow voices on all 16 channels, and provide at least 24 voices of polyphony.

Instruments that support the General MIDI features display the General MIDI logo.



The General MIDI Logo

A sequence composed for the General MIDI sound set (likely stored in an SMF) will play back similarly on any General MIDI instrument. Of course, even with the standardization, there will be some variation between instruments.

General MIDI was also common on PC soundcards with an onboard wavetable synthesizer. This was playable as an instrument in itself and also used by video game soundtracks. Cell phones also feature a GM sound generator, and can use SMF files as ringtones. Today, computers and cell phones are capable of handling audio files for video game soundtracks and sound effects, and the General MIDI/SMF pairing is becoming less common.

Shortcomings

MIDI was quite successful in solving the initial problem: synthesizers from different manufacturers can communicate performance information. A performer can connect MIDI instruments, and they respond similarly. Simply being able to do that was revolutionary, and paved the way for widespread adoption of MIDI.

It's popularity has been a mixed blessing. As MIDI caught on, it's been revised and extended, gaining features far beyond the initial intent. In hindsight, some of these features might be overly cumbersome or anachronistic. For example, we can ship files to and from MIDI instruments using SysEx messages, but today it might be easier to build an instrument that accepts SD cards or USB thumb drives.

Let's examine a couple of the more common complaints.

The Piano Keyboard

MIDI is based on a digital representation of the piano keyboard, and it works well for instruments that

ordinarily feature a keyboard. Note on and off commands have an obvious correspondence to actuating piano keys. How they relate to how other instruments is less obvious.

As one specific example, translating a guitar performance into MIDI data is particularly tricky. Accurately converting the vibrations of a guitar string into MIDI note numbers is much harder than simply noticing that a switch has closed. While there are commercial MIDI guitars around, they're best approached as a unique instrument, not a simple replacement for a guitar.

The piano keyboard is also intrinsically linked to the [12-tone western chromatic scale](#). How scales from other cultures translate into MIDI is not always obvious. There are [System Exclusive messages](#) that transmit alternate tuning information, but their implementation is uncommon.

Digital limitations

MIDI was designed using digital technology that was inexpensive in the early 1980's. More than 30 years later, it's still being used. Many other digital technologies from that era have since disappeared -- remember 5 1/4" floppy disks and monochrome, text only displays?

Computer networking has matured considerably during those 30 years. When compared to modern protocols like Ethernet or USB, MIDI is extremely primitive. The links are unidirectional, there is no provision for data integrity checking, such as a checksum or CRC, and addresses (channels) must be manually configured. The 31,250 bits-per-second links are also comparatively slow.

The flip side of this coin is that that MIDI is easy to implement on a modest microcontroller system. Serial port peripherals are found on many MCUs. The low data rate means that it's hard to overwhelm a receiver with too much data. Since the messages are small, storing them doesn't require much memory.

Within the messages, there are many places where the underlying data formats are also constrained.

With four bits of channel data, a MIDI buss supports a maximum of 16 channels. The usual workaround for this is to run more than one buss.

Many of the values sent by MIDI are constrained: key numbers, continuous controllers, and aftertouch values are all 7-bits long, allowing a range from 0 to 127, which can be too coarse for some parameter values. MIDI has provisions for pairing continuous controllers as an MSB and LSB to form 14-bit values, but in practice this is uncommon.

In examining the output of one commercial keyboard controller, the bender was actually only sending seven bits of data, left justified in the 14-bits of the bender messages. Moving the wheel slowly results in audible stairstepping. This is probably the resolution of the underlying ADC, an onboard peripheral of the system's MCU.

MIDI just didn't leave itself much room to grow.

Updates and Alternatives

The path MIDI is on has forked. The first branch keeps the message format of MIDI but transports them with newer technologies. The other branch opts to invent technologies that address the bottlenecks we explored in the [last section](#). Below, we'll touch on some of the more prevalent technologies from both branches.

MIDI on Newer Transports

USB was intended to replace the confusing array of ports on the back of older PCs. At one time, each peripheral attached to a PC used different connectors; the keyboard, mouse, modem, printer and joystick all had unique plugs. USB was designed as a single hot-swappable connector that could serve all of those functions. USB became commonplace in the early 2000's.

In addition to the peripherals mentioned above, the USB specification include a device class for MIDI ports.

At around the same time, Firewire (AKA IEEE-1394) was also deployed. It was a similar to USB, but

targeted at higher bandwidth multimedia applications. As a multimedia-oriented interface, MIDI was a natural fit. In the initial deployments, Firewire had the advantage of much higher bandwidth than USB -- 400 Mbit/S for 1394, vs. 12 Mbit/S for USB.

The USB and Firewire implementations of MIDI are similar. The UART and DIN connectors are replaced with the respective buss interface and plug. The messages exchanged, however, are still MIDI messages, wrapped up as USB or Firewire transactions. The MIDI messaging is essentially unchanged. This allows for backwards compatibility, but does nothing to address the shortcomings of MIDI.

These adapted protocols ultimately met quite different fates.

USB is used widely, and continues to evolve. In the USB 2.0 revision, a 480 MBit/S high-speed option was added, addressing the speed advantage that 1394 had over USB. Many newer MIDI devices feature a USB port instead of (or in addition to) regular MIDI ports. There are also standalone USB-to-MIDI interface adapters, for backwards compatibility.



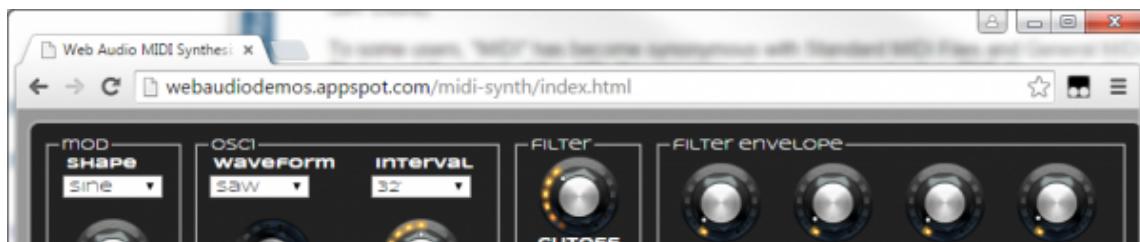
USB to MIDI adapter

Apple was the driving force behind Firewire, but they have more recently switched to the [Thunderbolt](#) and [Lightning](#) interfaces. As a result, Firewire is declining in popularity.

Web MIDI

Introduced by the [World Wide Web Consortium](#) as part of [HTML5](#), [Web MIDI](#) is an extension of HTML that allows web pages to access MIDI devices on a user's PC. The page can discover MIDI ports, then use them as inputs and outputs.

The synthesizer below is an example of a Web MIDI application. It uses Web MIDI to discover the MIDI ports on the PC. Once the user selects a port, the synthesizer is playable from an attached MIDI controller. It uses the [Web Audio API](#) to generate sounds.





If you want to try it for yourself, [click here](#).

Along with several demonstrations of Web MIDI and Web Audio, this example comes from webaudiodeemos.appspot.com. The author, Chris Wilson, hosts the source in his [GitHub repositories](#).

Other Standards

While MIDI is being adapted to new uses, sometimes it's easier to start with a blank slate.

Soundfonts and Downloadable Sounds

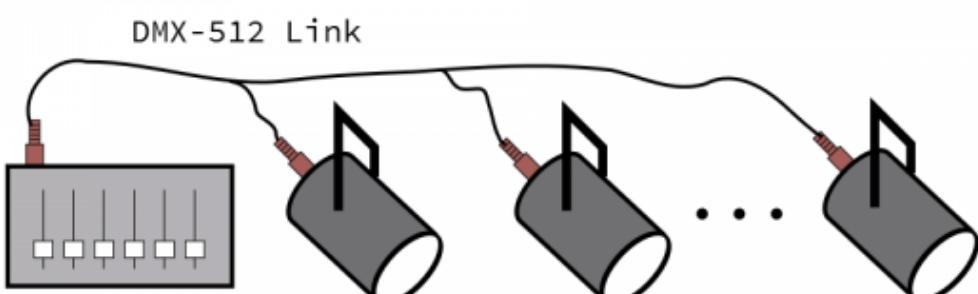
Growing on the ideas that lead to GM, Soundfonts are files that allow for the interchange of sound data between synthesizers. Rather than just using a number to request a type of sound, the sample data and related parameters are all sent together, making the sound itself transportable.

The [SoundFont](#) was developed by E-Mu Systems and Creative Labs, first implemented by the SoundBlaster AWE32 soundcard. The [Downloadable Sound](#) (DLS) is a similar concept for sound data transport adopted by the MMA.

DMX-512

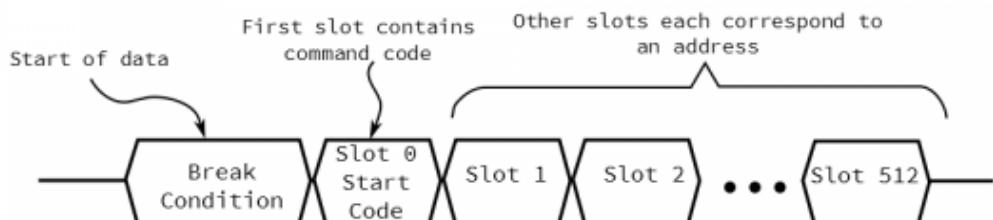
Some vendors adapted MIDI to control stage lights, but it's a secondary use of the protocol. MIDI has no intrinsic messages or structures to represent lighting-related information, and the defined maximum cable length of 50 feet is too short for large lighting rigs.

[DMX-512](#) was specified in 1986 as a lighting control protocol. Like MIDI, it uses a unidirectional daisy-chain of devices. Unlike MIDI, it uses a differential RS-485 electrical interface, allowing for much longer cable runs. It also uses a very different address scheme, supporting up to 512 devices on a single bus.



Address 1 Address 2 Address 512

The messaging protocol on the chain is actually much simpler than MIDI. The controller bursts out a stream of bytes, up to 512 bytes long. Each device on the bus has a configured address, and the address instructs which byte in the burst is meant for it.



A simple light fixture might receive a single byte controlling brightness, while a motorized fixture might receive many bytes, to control panning, color, brightness, [cucoloris](#) selection, and other effects.

Open Sound Control (OSC)

Over the years, there have been many attempts to unseat MIDI. A [number of new protocols](#) have been proposed, but most didn't catch on.

One notable exception is [Open Sound Control](#) (OSC).

OSC is a project of the UC Berkeley [Center for New Music and Audio Technology \(CMMAT\)](#). Compared to MIDI, it works at a very high level and leverages modern networking and data-structure practices. The usual transport is Ethernet, though it can be adapted to use [SLIP](#) over serial ports. The data structures are hierarchical and open-ended.

The [OSC page](#) on Wikipedia has lists of hardware and software that implement OSC.

OSC may suffer the reverse problem of MIDI: whereas MIDI is tightly constrained and messages are standardized, OSC is extremely open, allowing different instances to specify their own dialects, with less regard for interoperability. [SYN](#) is a proposed common dialect to be used between controllers and synthesizers.

Resources and Going Further

Resources

If you're interested in building your own MIDI system, we have some products to help you get started.

- If you want to build your own MIDI device, you can start with the Arduino-compatible [MIDI Shield](#). The [hookup guide](#) for the shield has several example sketches.
- If the shield is overkill, we've also got the raw [MIDI connector](#).
- If you'd like to skip the DIN connectors altogether, the [FreeSoC2](#) and [Teensy](#) both offer class-compliant USB-MIDI endpoint implementations.
- [Dr. Bleep](#) added MIDI to the second revision of the [Bleep Drum](#).

Going Further

- [MIDI.org](#) is the official website of the MIDI Manufacturer's Association.
- If you're doing MIDI with an Arduino, the 47 Effects [MIDI library](#) handles the messaging layer for you. It's

user friendly and very configurable.

- MIDI devices have long been a mainstay of DIY microcontroller projects. The [MIDIbox](#) is a platform for building a wide variety of MIDI devices.
- Some ponderings on the [finite state machine](#).
- The [MIDI Article](#) at Wikipedia.
- If you're really serious about MIDI, you might want a printed copy of the [Complete MIDI 1.0 Detailed Specification](#).

For more information about using controlling MIDI with BLE, check out the MIDI BLE tutorial linked to this blog post: