This document will present the broad brush strokes of the MIDI protocol. It's not meant to be an exhaustive treatise, but just enough to give you a basic understanding of what's happening when two devices communicate via MIDI. In particular, we'll concentrate on those aspects of MIDI that are used by ControllerHub 8.

What is MIDI?

MIDI is like USB or Ethernet, in that it is a way to send digital messages across a single wire. At any instant in time, the wire is either energized or inactive. Think of turning your lights on and off with a switch on the wall. There's a wire between the switch and the light running through the wall, and this wire is either energized or inactive depending on the state of the switch. There's nothing in between those two states; they are the only two possibilities. At any instant in time, the wire can be said to carry a single "bit" of information. Either on or off, yes or no, 1 or 0, true or false, energized or inactive.

Well, that's not very much information! How does this help us send a message? OK, now think of two ships at sea, communicating with lights. It's true that each ship's light can only be on or off at any given instant in time, but it's the *sequence* of on and off states that conveys information. It's the same with MIDI, USB, or Ethernet, or Morse code. Even though the wire can only be on or off at any given instant, it's the sequence of on states and off states that carries information. The on or off states are presented one at a time in a series, so this type of message passing is said to be *serial* communication.

Every form of serial communication must have some agreed upon *protocol* that both ends use, otherwise the series of on and off states would seem random. The protocol is like the rule book – if both the sender and receiver play by the same set of rules, then information can be exchanged.

Before getting back to MIDI, let's talk about groups of bits:

Combinations of multiple bits

If a single bit can have two possible states, 1 or 0 (on or off), then how many possible states can a group of two bits have? Let's count them:

00, 01, 10, 11 Four possible combinations.

How many possible combinations for a group of 3 bits?

000, 001, 010, 011, 100, 101, 110, 111 Eight possibilities.

How about 4 bits?

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

Sixteen.

Notice how every time we add a bit, the number of possible states doubles. This makes sense. When we add a new bit to an existing group of bits, the new bit can be either 0 or 1, so we have:

"0" plus the number of possibilities of the original group
+
"1" plus the number of possibilities of the original group

= double the number of possibilities of the original group.

A shorthand way of saying this is that the number of possible states of a group of bits is equal to 2 raised to the number of bits. One bit can have $2^1 = 2$ states, two bits can have $2^2 = 4$ states, three bits can have $2^3 = 8$ states, etc.

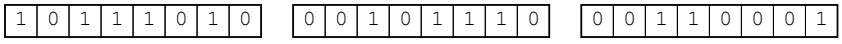
A group of 8 bits is called a *byte*. If each individual bit can be either a 0 or a 1 (off or on), then a group of 8 bits can be in any of $2^8 = 256$ states:

00000000, 00000001, 00000010, 00000011, 00000100, 00000101, and so on, up to 11111111

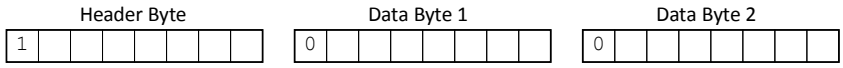
Most MIDI messages are three bytes long (there are exceptions which I'll get to shortly, but for now we'll consider the most probable case). In other words, the number of bits in the sequence of on/off bits sent in a MIDI message is 24. If you observe a MIDI message using an oscilloscope, you can't see the partitioning of the 24 bits into 3 bytes; all 24 bits are sent one right after the other without any breaks in between. But it's helpful for us humans to think of the 24 bits as being broken into bytes.

So here's how we'll show a three-byte message:

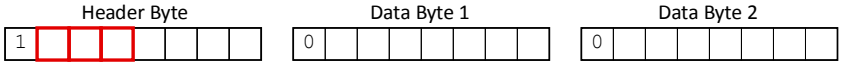
Again, there aren't really any gaps in between the bytes but it's helpful for us to think of them this way. And this message is just an empty place holder; there aren't any zeroes or ones being shown. Here's a real message populated with zeroes & ones:



The first thing to know about a MIDI message is that the first bit of each byte is used to help identify the first byte in each message. The first byte is special; it's called the Header Byte. In order to identify the header byte, the MIDI protocol says that the first bit of the header byte will be set to a 1, while the first bit of all other message bytes (called data bytes) will be set to zero. Like this:



The **next three bits** of the header byte give the Message Type:

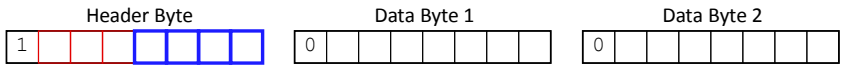


Since there are 3 message type bits, there are $2^3 = 8$ possible message types. I won't list them all, but here are the ones most often used:

- **000** = Note Off
- **001** = Note On
- **011** = Continuous Controller (CC)
- **100** = Preset Change
- **111** = System Exclusive (SysEx)

Only the CC and SysEx message types are used by ControllerHub 8, though all MIDI message types may be passed through ControllerHub 8.

The **last four bits** of the header byte give the MIDI Channel Number of the intended recipient:



Since there are 4 MIDI Channel Number bits, there are $2^4 = 16$ possible MIDI Channels. The channel number is set by the user on both ends of the cable. For example, suppose the user has 3 MIDI devices to control. The first task is to set each of the three MIDI devices to a unique channel number. The manual for each device will explain how this is done. Then, the sending device sets the channel number of each message it sends to match one of the channel numbers assigned to the device intended to be the recipient of the message.

Earlier I said that the most common number of bytes in a MIDI message is three, but more accurately the total number of bytes in a MIDI message changes depending on message type. So, too, does the content of those bytes. While Note On, Note Off, and CC messages all have three total bytes, a Preset Change message only has two bytes, while a SysEx message can have any number of bytes. Here are explanations of the data bytes of the MIDI message types listed above:

Note On Message Bytes:

- Header byte – specifies message type as Note On
- Data Byte 1 - Which note was played?
- Data Byte 2 – How hard was it played?

Note Off Message Bytes:

- Header byte – specifies message type as Note Off
- Data Byte 1 - Which note was released?
- Data Byte 2 – How fast was it released?

Continuous Controller (CC) Message Bytes:

- Header byte – specifies message type as CC
- Data Byte 1 - Which CC Number is being sent?
- Data Byte 2 – What is the current controller's Value?

Preset Change Message Bytes:

- Header byte – specifies message type as Preset Change
- Data Byte 1 - Which preset are we jumping to?

System Exclusive (SysEx) Message Bytes:

- Header byte – specifies message type as System Exclusive
- Data Byte 1 through Data Byte *n* – Your Message Here ("design your own" message protocol)
- Data Byte *n*+1 – End Of Exclusive (EOX)

Remember that the first bit of every byte is allocated to the task of identifying the Header Byte. The Header Byte's first bit is set to 1, while the first bit of all other bytes must be set to zero. OK. But that means that all the data bytes only have 7 bits remaining to carry the message data. So instead of the 256 bit combinations normally possible with bytes, we only have $2^7 = 128$ possible combinations. This explains why almost every MIDI device has 128 presets, for example. It also explains why when a MIDI device (such as ControllerHub 8) sends the state of an expression pedal, the value runs through a range from 0 to 127 as the pedal moves from heel to toe.

Incidentally, sometimes you will see MIDI channel numbers, preset numbers, CC numbers, and CC values starting with 0; other times they will start with 1. For example, sometimes MIDI channels will be said to run from 0-15; other times they will be said to run from 1-16. Continous Controller values may be specified from 1-128, or 0-127. You won't find complete agreement from manufacturer to manufacturer. So every once in a while, there will be an "off by one" error that results from this difference. When the data is actually sent on the MIDI cable, the values always run from zero to max-1. But sometimes it seems more human to specify them as running from 1 to max.

MIDI as observed on an oscilloscope:

I'll update this doc later with some actual oscilloscope observations.