Arduino MIDI

Pieter P, 08-03-2017

This is a guide that covers the basics of the Musical Instrument Digital Interface (MIDI) protocol and its implementation on the Arduino platform.
The format of the protocol is explained in the first chapter. Chapter two goes over the hardware. In chapter three, example code for sending MIDI is presented. Chapter four contains everything needed to build a working MIDI controller. MIDI input is covered in chapter five, and chapter six extends this by adding support for System Exclusive (SysEx) messages.

The MIDI protocol

The MIDI specification can be found here: https://www.midi.org/specifications/item/the-midi-1-0-specification

The MIDI protocol describes a set of MIDI events. For example, a note is played, or a note is turned off, a controller is moved and set to a new value, a new instrument is selected, etc. These events correspond to MIDI messages that can be sent over the MIDI hardware connection.

There are two main types of messages: channel messages and system messages. Most performance information will be sent as channel messages, while system messages are used for things like proprietary handshakes, manufacturer-specific settings, sending long packets of data, real-time messages for synchronization and tuning, and other things that are not really of interest to someone who just wants to make an Arduino MIDI instrument or controller. That's why this guide will mainly focus on channel messages.

Channel messages

There are 16 MIDI channels. Each MIDI instrument can play notes on one of these channels, and they can apply different voices or patches to different channels, as well as setting some controllers like volume, pan, balance, sustain pedal, pitch bend, etc.
MIDI messages that target a specific channel are called channel messages.

A MIDI channel message consist of a header byte, referred to as the status byte, followed by one or two data bytes:

```
Status─Data
Status─Data 1─ Data 2
```

Each byte consists of 8 binary digits. To distinguish between status and data bytes, and to prevent framing errors, status bytes have the most significant bit (msb) set to one (1), and data bytes have the msb set to zero (0).

	Status byte								Data byte 1								Data byte 2 (optional)							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Value	1	x	x	x	x	x	x	x	0	x	x	x	x	x	x	x	0	x	x	x	x	x	x	x

Status bytes

The status byte of channel messages is divided into two 4-bit nibbles. The high nibble (bits 4-7) specifies the message type, and the low nibble (bits 0-3) specifies the MIDI channel. Because the most significant bit has to be one, there are 8 different message types (`0b1000` - `0b1111` or `0x8` - `0xF`), and 16 different channels (`0x0` - `0xF`). Message type `0xF` is used for system messages, so it won't be covered in this section on channel messages.

	Status byte							
Bit	7	6	5	4	3	2	1	0
Value	m	m	m	m	n	n	n	n

Where `mmmm` is the message type (`0x8` - `0xE`) and `nnnn` is the channel nibble.
Note that the channels start from `nnnn = 0` for MIDI channel 1. (`nnnn = channel - 1`)

Data bytes

Each data byte contains a 7-bit value, a number between 0 and 127 (`0b01111111` or `0x7F`). The meaning of this value depends on the message type. For example, it can tell the receiver what note is played, how hard the key was struck, what instrument to select, what value a controller is set to, etc.

Channel Messages: message types

The following section will go over the different channel messages and their status and data bytes.
Keep in mind that `nnnn = channel - 1`.

Note Off (0x8)

A note off event is used to stop a playing note. For example, when a key is released.

Data 1 (`0b0kkkkkkk`): Note number (key). See MIDI note names.
Data 2 (`0b0vvvvvvv`): Velocity (how fast the key is released).

- A velocity of 0 is not defined, and some software or devices may not register the note off event if the velocity is zero.
- Most software or devices will ignore the note off velocity.
- Instead of a note off event, a note on event with a velocity of zero may be used. This is especially useful when using a running status.

	Status byte								Note number								Velocity							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Value	1	0	0	0	n	n	n	n	0	k	k	k	k	k	k	k	0	v	v	v	v	v	v	v

Note On (0x9)

A note on event is used to play a note. For example, when a key is pressed.

Data 1 (`0b0kkkkkkk`): Note number (key). See MIDI note names.
Data 2 (`0b0vvvvvvv`): Velocity (how fast/hard the key is pressed).

- If the velocity is zero, the note on event is interpreted as a note off event. This is especially useful when using a running status.

	Status byte								Note number								Velocity							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Value	1	0	0	1	n	n	n	n	0	k	k	k	k	k	k	k	0	v	v	v	v	v	v	v

Polyphonic key pressure (0xA)

A polyphonic key pressure event is used when the pressure on a key or a pressure sensitive pad changes after the note on event.

Data 1 (`0b0kkkkkkk`): Note number (key). See MIDI note names.
Data 2 (`0b0vvvvvvv`): Pressure on the key.

- Most normal MIDI keyboards do not implement this event.
- Key pressure is sometimes referred to as after-touch or after-pressure.

	Status byte								Note number								Pressure							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Value	1	0	1	0	n	n	n	n	0	k	k	k	k	k	k	k	0	v	v	v	v	v	v	v

Control change (0xB)

A control change event is used when the value of a controller changes.

Data 1 (`0b0ccccccc`): Controller number. See Controller numbers.
Data 2 (`0b0vvvvvvv`): The value of the controller.

- Controller numbers 120-127 are reserved as "Channel Mode Messages".

	Status byte								Controller number								Value							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Value	1	0	1	1	n	n	n	n	0	c	c	c	c	c	c	c	0	v	v	v	v	v	v	v

Program change (0xC)

A program change event is used to change the program (i.e. sound, voice, tone, preset or patch) of a given channel is changed.

Data 1 (`0b0ppppppp`): Program number. See Program numbers.

- Controller numbers 120-127 are reserved as "Channel Mode Messages".

	Status byte								Program number							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Value	1	1	0	0	n	n	n	n	0	c	c	c	c	c	c	c

Channel pressure `(0xD)`

A channel pressure event is used when the pressure on a key or a pressure sensitive pad changes after the note on event. Unlike polyphonic key pressure, channel pressure affects all notes playing on the channel.

Data 1 (`0b0vvvvvvv`): Pressure value.

- Most normal MIDI keyboards do not implement this event.
- Channel pressure is sometimes referred to as after-touch or after-pressure.

	Status byte								Pressure							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Value	1	1	0	1	n	n	n	n	0	v	v	v	v	v	v	v

Pitch bend change `(0xE)`

A pitch bend change event is used to alter the pitch of the notes played on a given channel.

Data 1 (`0b0lllllll`): Least significant byte (bits 0-7) of the pitch bend value.
Data 2 (`0b0mmmmmmm`): Most significant byte (bits 8-13) of the pitch bend value.

- The center position (no pitch change) is represented by LSB = `0x0`, MSB = `0x40`

	Status byte								LSB								MSB							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Value	1	1	1	0	n	n	n	n	0	l	l	l	l	l	l	l	0	m	m	m	m	m	m	m

Running status

There are a lot of circumstances where you have to send many messages of the same type. For example, if you have a digital keyboard, pretty much all messages will be note on and note off events, or when you turn a knob on a MIDI controller, a lot of control change messages will be sent to update the controller value. To save bandwidth in these kinds of situations, you only have to send the status byte once, followed by only data bytes. This technique is called "running status".
Because note on events are most likely to be followed by note off events (or more note on events), the MIDI standard allows you to use a note on event with a velocity of zero instead of a note off event. This means that you only need one status byte for all note events, drastically reducing the data throughput, thus minimizing the delay between events.

System Messages

System messages are MIDI messages that do not carry data for a specific MIDI channel. There are three types of system messages:

System Common Messages

System Common messages are intended for all receivers in the system.
These messages are beyond the scope of this guide. If you want more information, refer to page 27 of the MIDI 1.0 Detailed Specification 4.2.

- MIDI Time Code Quarter Frame `(0xF1)`
- Song Position Pointer `(0xF2)`
- Song Select `(0xF3)`
- Tune Request `(0xF6)`
- EOX (End of Exclusive) `(0xF7)`

System Real Time Messages

System Real Time messages are used for synchronization between clock-based MIDI components.
These messages are beyond the scope of this guide. If you want more information, refer to page 30 of the MIDI 1.0 Detailed Specification 4.2.

- Timing Clock (`0xF8`)
- Start (`0xFA`)
- Continue (`0xFB`)
- Stop (`0xFC`)
- Active Sensing (`0xFE`)
- System Reset (`0xFF`)

System Exclusive Messages

System Exclusive (SysEx) messages are used for things like setting synthesizer or patch settings, sending sampler data, memory dumps, etc.
Most SysEx messages are manufacturer-specific, so it is best to consult the MIDI implementation in the manual. If you want more information on the topic, you can find it on page 34 of the MIDI 1.0 Detailed Specification 4.2.

A system exclusive message starts with a status byte `0xF0`, followed by an arbitrary number of data bytes, and ends with another status byte `0xF7`.

	SysEx start								Data								…	SysEx end							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	…	7	6	5	4	3	2	1	0
Value	1	1	1	1	0	0	0	0	0	d	d	d	d	d	d	d	…	1	1	1	1	0	1	1	1

Appendices

All numbers are in hexadecimal representation, unless otherwise specified.

MIDI note names

Middle C or C4 is defined as MIDI note `0x3C`.
The lowest note on a standard 88-key piano is A0 (`0x15`) and the highest note is C8 (`0x6C`).

	Note											
Octave	**C**	**C#**	**D**	**D#**	**E**	**F**	**F#**	**G**	**G#**	**A**	**A#**	**B**
-1	00	01	02	03	04	05	06	07	08	09	0A	0B
0	0C	0D	0E	0F	10	11	12	13	14	15	16	17
1	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23
2	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
3	30	31	32	33	34	35	36	37	38	39	3A	3B
4	3C	3D	3E	3F	40	41	42	43	44	45	46	47
5	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53
6	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
7	60	61	62	63	64	65	66	67	68	69	6A	6B
8	6C	6D	6E	6F	70	71	72	73	74	75	76	77
9	78	79	7A	7B	7C	7D	7E	7F				

Controller numbers

This is an overview of the MIDI controller numbers that can be used as the first data byte of a control change event.
The second data byte is the value for the controller. This value is 7 bits wide, so has a range of [0, 127]. Controller numbers `0x00`-`0x1F` can be combined with numbers `0x20`-`0x3F` for 14-bit resolution. In this case, numbers `0x00`-`0x1F` set the MSB, and numbers `0x20`-`0x3F` the LSB.
Controller numbers 120-127 are reserved for Channel Mode Messages, which rather than controlling sound parameters, affect the channel's operating mode.

Controller	Function	Value	Used

Dec	Hex			as
0	00	Bank Select	00-7F	MSB
1	01	Modulation Wheel or Lever	00-7F	MSB
2	02	Breath Controller	00-7F	MSB
3	03	Undefined	00-7F	MSB
4	04	Foot Controller	00-7F	MSB
5	05	Portamento Time	00-7F	MSB
6	06	Data Entry MSB	00-7F	MSB
7	07	Channel Volume (formerly Main Volume)	00-7F	MSB
8	08	Balance	00-7F	MSB
9	09	Undefined	00-7F	MSB
10	0A	Pan	00-7F	MSB
11	0B	Expression Controller	00-7F	MSB
12	0C	Effect Control 1	00-7F	MSB
13	0D	Effect Control 2	00-7F	MSB
14	0E	Undefined	00-7F	MSB
15	0F	Undefined	00-7F	MSB
16	10	General Purpose Controller 1	00-7F	MSB
17	11	General Purpose Controller 2	00-7F	MSB
18	12	General Purpose Controller 3	00-7F	MSB
19	13	General Purpose Controller 4	00-7F	MSB
20	14	Undefined	00-7F	MSB
21	15	Undefined	00-7F	MSB
22	16	Undefined	00-7F	MSB
23	17	Undefined	00-7F	MSB
24	18	Undefined	00-7F	MSB
25	19	Undefined	00-7F	MSB
26	1A	Undefined	00-7F	MSB
27	1B	Undefined	00-7F	MSB
28	1C	Undefined	00-7F	MSB
29	1D	Undefined	00-7F	MSB
30	1E	Undefined	00-7F	MSB
31	1F	Undefined	00-7F	MSB
32	20	LSB for Control 0 (Bank Select)	00-7F	LSB
33	21	LSB for Control 1 (Modulation Wheel or Lever)	00-7F	LSB
34	22	LSB for Control 2 (Breath Controller)	00-7F	LSB
35	23	LSB for Control 3 (Undefined)	00-7F	LSB
36	24	LSB for Control 4 (Foot Controller)	00-7F	LSB
37	25	LSB for Control 5 (Portamento Time)	00-7F	LSB
38	26	LSB for Control 6 (Data Entry)	00-7F	LSB
39	27	LSB for Control 7 (Channel Volume, formerly Main Volume)	00-7F	LSB
40	28	LSB for Control 8 (Balance)	00-7F	LSB
41	29	LSB for Control 9 (Undefined)	00-7F	LSB
42	2A	LSB for Control 10 (Pan)	00-7F	LSB
43	2B	LSB for Control 11 (Expression Controller)	00-7F	LSB
44	2C	LSB for Control 12 (Effect control 1)	00-7F	LSB
45	2D	LSB for Control 13 (Effect control 2)	00-7F	LSB
46	2E	LSB for Control 14 (Undefined)	00-7F	LSB
47	2F	LSB for Control 15 (Undefined)	00-7F	LSB
48	30	LSB for Control 16 (General Purpose Controller 1)	00-7F	LSB
49	31	LSB for Control 17 (General Purpose Controller 2)	00-7F	LSB
50	32	LSB for Control 18 (General Purpose Controller 3)	00-7F	LSB

51	33	LSB for Control 19 (General Purpose Controller 4)	00-7F	LSB
52	34	LSB for Control 20 (Undefined)	00-7F	LSB
53	35	LSB for Control 21 (Undefined)	00-7F	LSB
54	36	LSB for Control 22 (Undefined)	00-7F	LSB
55	37	LSB for Control 23 (Undefined)	00-7F	LSB
56	38	LSB for Control 24 (Undefined)	00-7F	LSB
57	39	LSB for Control 25 (Undefined)	00-7F	LSB
58	3A	LSB for Control 26 (Undefined)	00-7F	LSB
59	3B	LSB for Control 27 (Undefined)	00-7F	LSB
60	3C	LSB for Control 28 (Undefined)	00-7F	LSB
61	3D	LSB for Control 29 (Undefined)	00-7F	LSB
62	3E	LSB for Control 30 (Undefined)	00-7F	LSB
63	3F	LSB for Control 31 (Undefined)	00-7F	LSB
64	40	Damper Pedal on/off (Sustain)	≤3F off, ≥40 on	---
65	41	Portamento On/Off	≤3F off, ≥40 on	---
66	42	Sostenuto On/Off	≤3F off, ≥40 on	---
67	43	Soft Pedal On/Off	≤3F off, ≥40 on	---
68	44	Legato Footswitch	≤3F Normal, ≥40 Legato	---
69	45	Hold 2	≤3F off, ≥40 on	---
70	46	Sound Controller 1 (default: Sound Variation)	00-7F	LSB
71	47	Sound Controller 2 (default: Timbre/Harmonic Intens.)	00-7F	LSB
72	48	Sound Controller 3 (default: Release Time)	00-7F	LSB
73	49	Sound Controller 4 (default: Attack Time)	00-7F	LSB
74	4A	Sound Controller 5 (default: Brightness)	00-7F	LSB
75	4B	Sound Controller 6 (default: Decay Time - see MMA RP-021)	00-7F	LSB
76	4C	Sound Controller 7 (default: Vibrato Rate - see MMA RP-021)	00-7F	LSB
77	4D	Sound Controller 8 (default: Vibrato Depth - see MMA RP-021)	00-7F	LSB
78	4E	Sound Controller 9 (default: Vibrato Delay - see MMA RP-021)	00-7F	LSB
79	4F	Sound Controller 10 (default undefined - see MMA RP-021)	00-7F	LSB
80	50	General Purpose Controller 5	00-7F	LSB
81	51	General Purpose Controller 6	00-7F	LSB
82	52	General Purpose Controller 7	00-7F	LSB
83	53	General Purpose Controller 8	00-7F	LSB
84	54	Portamento Control	00-7F	LSB
85	55	Undefined	---	---
86	56	Undefined	---	---
87	57	Undefined	---	---
88	58	High Resolution Velocity Prefix	00-7F	LSB
89	59	Undefined	---	---
90	5A	Undefined	---	---
91	5B	Effects 1 Depth (default: Reverb Send Level - see MMA RP-023) (formerly ExternalEffects Depth)	00-7F	---
92	5C	Effects 2 Depth (formerly Tremolo Depth)	00-7F	---
93	5D	Effects 3 Depth (default: Chorus Send Level - see MMA RP-023) (formerly Chorus Depth)	00-7F	---
94	5E	Effects 4 Depth (formerly Celeste [Detune] Depth)	00-7F	---
95	5F	Effects 5 Depth (formerly Phaser Depth)	00-7F	---
96	60	Data Increment (Data Entry +1) (see MMA RP-018)	N/A	---
97	61	Data Decrement (Data Entry -1) (see MMA RP-018)	N/A	---
98	62	Non-Registered Parameter Number (NRPN) - LSB	00-7F	LSB
99	63	Non-Registered Parameter Number (NRPN) - MSB	00-7F	MSB
100	64	Registered Parameter Number (RPN) - LSB*	00-7F	LSB

101	65	Registered Parameter Number (RPN) - MSB*	00-7F	MSB
102	66	Undefined	---	---
103	67	Undefined	---	---
104	68	Undefined	---	---
105	69	Undefined	---	---
106	6A	Undefined	---	---
107	6B	Undefined	---	---
108	6C	Undefined	---	---
109	6D	Undefined	---	---
110	6E	Undefined	---	---
111	6F	Undefined	---	---
112	70	Undefined	---	---
113	71	Undefined	---	---
114	72	Undefined	---	---
115	73	Undefined	---	---
116	74	Undefined	---	---
117	75	Undefined	---	---
118	76	Undefined	---	---
119	77	Undefined	---	---

Channel mode		Function	Value
Dec	Hex		
120	78	All Sound Off	00
121	79	Reset All Controllers	00
122	7A	Local Control On/Off	00 off, 7F on
123	7B	All Notes Off	00
124	7C	Omni Mode Off (+ all notes off)	00
125	7D	Omni Mode On (+ all notes off)	00
126	7E	Mono Mode On (+ poly off, + all notes off)	Note: This equals the number of channels, or zero if the number of channels equals the number of voices in the receiver.
127	7F	Poly Mode On (+ mono off, + all notes off)	0

Source

Program numbers

The MIDI specification doesn't specify instruments or voices for program numbers. The General MIDI 1 sound set does define a list of sounds and families of sounds.

Program	Family Name
1-8	Piano
9-16	Chromatic Percussion
17-24	Organ
25-32	Guitar
33-40	Bass
41-48	Strings
49-56	Ensemble
57-64	Brass
65-72	Reed
73-80	Pipe
81-88	Synth Lead
89-96	Synth Pad
97-104	Synth Effects

105-112	Ethnic
113-120	Percussive
121-128	Sound Effects

Program	**Instrument Name**
1	Acoustic Grand Piano
2	Bright Acoustic Piano
3	Electric Grand Piano
4	Honky-tonk Piano
5	Electric Piano 1
6	Electric Piano 2
7	Harpsichord
8	Clavi
9	Celesta
10	Glockenspiel
11	Music Box
12	Vibraphone
13	Marimba
14	Xylophone
15	Tubular Bells
16	Dulcimer
17	Drawbar Organ
18	Percussive Organ
19	Rock Organ
20	Church Organ
21	Reed Organ
22	Accordion
23	Harmonica
24	Tango Accordion
25	Acoustic Guitar (nylon)
26	Acoustic Guitar (steel)
27	Electric Guitar (jazz)
28	Electric Guitar (clean)
29	Electric Guitar (muted)
30	Overdriven Guitar
31	Distortion Guitar
32	Guitar harmonics
33	Acoustic Bass
34	Electric Bass (finger)
35	Electric Bass (pick)
36	Fretless Bass
37	Slap Bass 1
38	Slap Bass 2
39	Synth Bass 1
40	Synth Bass 2
41	Violin
42	Viola
43	Cello
44	Contrabass
45	Tremolo Strings
46	Pizzicato Strings
47	Orchestral Harp
48	Timpani

49	String Ensemble 1
50	String Ensemble 2
51	SynthStrings 1
52	SynthStrings 2
53	Choir Aahs
54	Voice Oohs
55	Synth Voice
56	Orchestra Hit
57	Trumpet
58	Trombone
59	Tuba
60	Muted Trumpet
61	French Horn
62	Brass Section
63	SynthBrass 1
64	SynthBrass 2
65	Soprano Sax
66	Alto Sax
67	Tenor Sax
68	Baritone Sax
69	Oboe
70	English Horn
71	Bassoon
72	Clarinet
73	Piccolo
74	Flute
75	Recorder
76	Pan Flute
77	Blown Bottle
78	Shakuhachi
79	Whistle
80	Ocarina
81	Lead 1 (square)
82	Lead 2 (sawtooth)
83	Lead 3 (calliope)
84	Lead 4 (chiff)
85	Lead 5 (charang)
86	Lead 6 (voice)
87	Lead 7 (fifths)
88	Lead 8 (bass + lead)
89	Pad 1 (new age)
90	Pad 2 (warm)
91	Pad 3 (polysynth)
92	Pad 4 (choir)
93	Pad 5 (bowed)
94	Pad 6 (metallic)
95	Pad 7 (halo)
96	Pad 8 (sweep)
97	FX 1 (rain)
98	FX 2 (soundtrack)
99	FX 3 (crystal)
100	FX 4 (atmosphere)

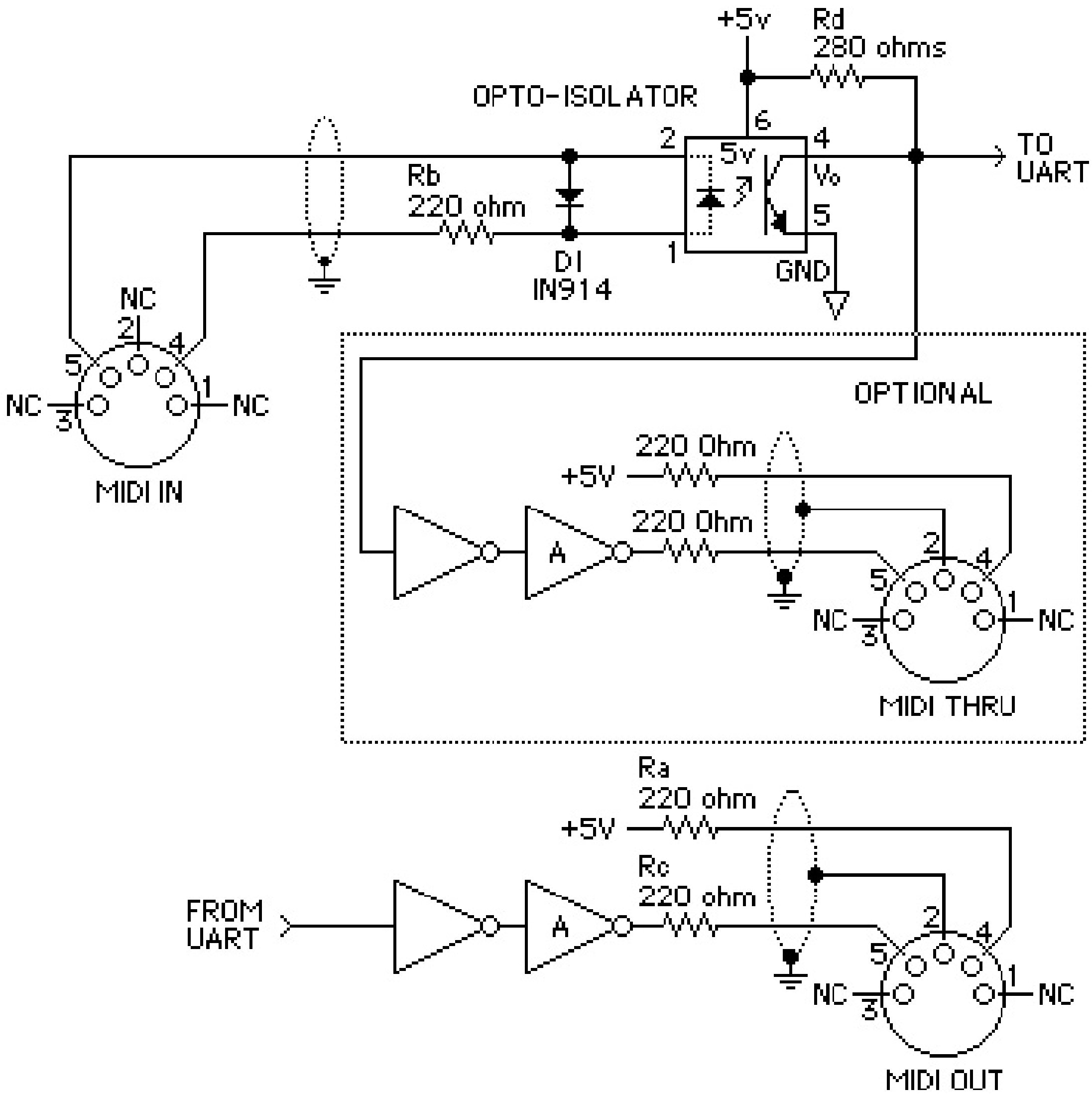
101	FX 5 (brightness)
102	FX 6 (goblins)
103	FX 7 (echoes)
104	FX 8 (sci-fi)
105	Sitar
106	Banjo
107	Shamisen
108	Koto
109	Kalimba
110	Bag pipe
111	Fiddle
112	Shanai
113	Tinkle Bell
114	Agogo
115	Steel Drums
116	Woodblock
117	Taiko Drum
118	Melodic Tom
119	Synth Drum
120	Reverse Cymbal
121	Guitar Fret Noise
122	Breath Noise
123	Seashore
124	Bird Tweet
125	Telephone Ring
126	Helicopter
127	Applause
128	Gunshot

Source

MIDI hardware

The MIDI hardware link is just a 5mA current loop that asynchronously sends and receives 8-bit bytes at a baud rate of 31250 symbols per second. This means that the Arduino's hardware UART can be used for transmitting and receiving MIDI. DIN 5 pin (180 degree) female receptacles are used for MIDI in, out and through connectors.

This is the original schematic that can be found in the 1996 MIDI 1.0 Detailed Specification 4.2:



MIDI Standard Hardware

NOTES:

1. Opto-isolator currently shown is Sharp PC-900
   (HP 6N138 or other opto-isolator can be used with appropriate changes.)

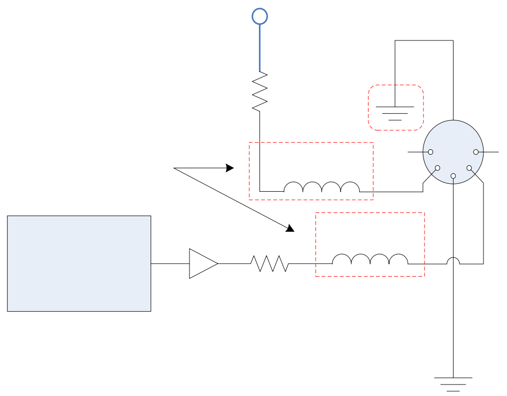
2. Gates "A" are IC or transistor.

3. Resistors are 5%

The current loop consists of a an open collector output on the transmitting end (MIDI out and MIDI through), and an opto-isolator at the receiving end (MIDI in). When a 'zero' is sent, the open collector output sinks current, turning on the LED of the opto-isolator. This will in turn bring low the open collector output of the opto-isolator, resulting in a low signal.
The reason for using a current loop instead of a voltage, is that the sender and the receiver can

be at different potentials, because everything is galvanically isolated. This also prevents ground loops, which can result in noise.
Note that the ground and shielding (pin 2 on the 5-pin DIN connector) is connected to the ground of the MIDI out and through circuits, but not to the ground of the receiver in the MIDI in circuit.

The standard was updated in 2014 to include specifications for 3.3V MIDI devices.
(*MIDI 1.0 Electrical Specification Update (CA-033)* (2014). MMA Technical Standards Board / AMEI MIDI Committee.)



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

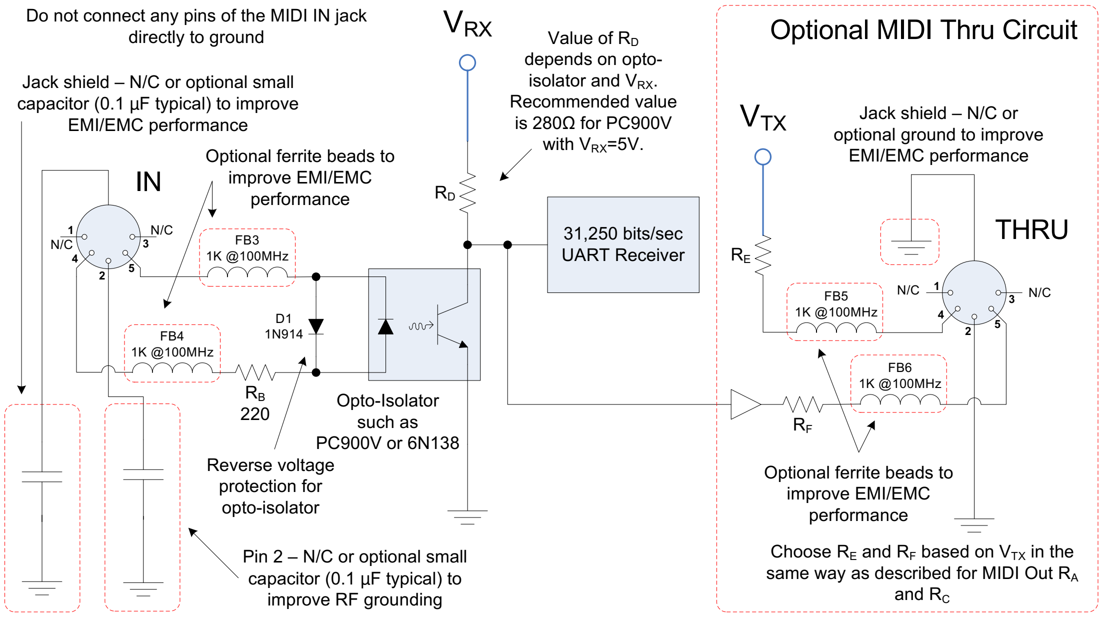
The buffer between the UART transmitter and R_C is optional and system-dependent.

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



Sending MIDI over Serial

The easiest way to send out MIDI packets is to use the `Serial.write(uint8_t data);` function. This function writes out one 8-bit byte over the Serial connection (either hardware UART0 or the virtual COM port over USB).
To send out a MIDI packet, we just have to write out the three bytes that make up the packet: first the status byte, then the two data bytes.

```
void sendMIDI(uint8_t statusByte, uint8_t dataByte1, uint8_t dataByte2) {
  Serial.write(statusByte);
  Serial.write(dataByte1);
  Serial.write(dataByte2);
}
```

In order to support MIDI packets with only one data byte as well, we can just overload the sendMIDI function. This means that we create two functions with the same name, but with different parameters.

```
void sendMIDI(uint8_t statusByte, uint8_t dataByte) {
  Serial.write(statusByte);
  Serial.write(dataByte);
}
```

In its current form, the sendMIDI function is quite silly. Although it sends out MIDI packets, it doesn't automatically create these packets for us, we still have to put together the status and data bytes ourselves, and we have to make sure that it is a valid MIDI packet before calling sendMIDI. Let's create a more useful function that takes a message type, channel number and data as inputs, creates a MIDI packet, and sends it over the Serial port.

```
void sendMIDI(uint8_t messageType, uint8_t channel, uint8_t data1, uint8_t data2) {
  channel--;  // Decrement the channel, because MIDI channel 1 corresponds to binary channel 0
  uint8_t statusByte = messageType | channel;  // Combine the messageType (high nibble)
                                               // with the channel (low nibble)
                                               // Both the message type and the channel should be
4 bits wide
  Serial.write(statusByte);
  Serial.write(data1);
  Serial.write(data2);
}
```

We now have a working function that sends MIDI packets, and takes a somewhat sensible input, not just the bytes of the packet. But there's still no guarantee that it is a valid MIDI message. Remember that the status byte should have a most significant bit equal to 1, and the data bytes a most significant bit equal to 0. We'll use some bitwise math to make sure that this is always the case, no matter what data the user enters.

```
void sendMIDI(uint8_t messageType, uint8_t channel, uint8_t data1, uint8_t data2) {
  channel--;                                   // Decrement the channel, because MIDI channel 1
                                               // corresponds to binary channel 0
  uint8_t statusByte = messageType | channel;  // Combine the messageType (high nibble)
                                               // with the channel (low nibble)
                                               // Both the message type and the channel
                                               // should be 4 bits wide
  statusByte |= 0b10000000;                    // Set the most significant bit of the status byte
  data1      &= 0b01111111;                    // Clear the most significant bit of the data
bytes
  data2      &= 0b01111111;
  Serial.write(statusByte);                    // Send over Serial
  Serial.write(data1);
  Serial.write(data2);
}
```

```
void sendMIDI(uint8_t messageType, uint8_t channel, uint8_t data) {
  channel--;                                   // Decrement the channel, because MIDI channel 1
                                               // corresponds to binary channel 0
  uint8_t statusByte = messageType | channel;  // Combine the messageType (high nibble)
                                               // with the channel (low nibble)
                                               // Both the message type and the channel
                                               // should be 4 bits wide
```

```
  statusByte |= 0b10000000;                    // Set the most significant bit of the status byte
  data       &= 0b01111111;                    // Clear the most significant bit of the data byte
  Serial.write(statusByte);                    // Send over Serial
  Serial.write(data);
}
```

Before sending the packet, we set the most significant bit of the status byte by performing a bitwise OR operation:

```
0bxsss ssss
0b1000 0000
----------- |
0b1sss ssss
```

Where `0bsss ssss` is the status, and `x` is either 1 or 0. As you can see, no matter the value of `x`, the result will always be `0b1sss ssss`.
We also clear the most significant bits of the data bytes by performing a bitwise AND operation:

```
0bxddd dddd
0b0111 1111
----------- &
0b0ddd dddd
```

Where `0bddd dddd` is the data, and `x` is either 1 or 0. No matter what the value of `x` is, the result will always be `0b0ddd dddd`

You could go even further by making sure that the message type and the channel don't interfere with each other. However, that might be overly defensive.

```
void sendMIDI(uint8_t messageType, uint8_t channel, uint8_t data1, uint8_t data2) {
  channel--;                                   // Decrement the channel, because MIDI channel 1
                                               // corresponds to binary channel 0
  messageType &= 0b11110000;                   // Make sure that only the high nibble
                                               // of the message type is set
  channel     &= 0b00001111;                   // Make sure that only the low nibble
                                               // of the channel is set
  uint8_t statusByte = messageType | channel;  // Combine the messageType (high nibble)
                                               // with the channel (low nibble)
                                               // Both the message type and the channel
                                               // should be 4 bits wide
  statusByte  |= 0b10000000;                   // Set the most significant bit of the status byte
  data1       &= 0b01111111;                   // Clear the most significant bit of the data
bytes
  data2       &= 0b01111111;
  Serial.write(statusByte);                    // Send over Serial
  Serial.write(data1);
  Serial.write(data2);
}
```

Improving readability

To send a Control Change (0xB0) message on channel 3 for controller 80 with a value of 64, you would call `sendMIDI(0xB0, 3, 80, 64);`
To make it a little more obvious what's going on, we could declare some constants for the different message types:

```
const uint8_t NOTE_OFF = 0x80;
const uint8_t NOTE_ON = 0x90;
const uint8_t KEY_PRESSURE = 0xA0;
const uint8_t CC = 0xB0;
const uint8_t PROGRAM_CHANGE = 0xC0;
const uint8_t CHANNEL_PRESSURE = 0xD0;
const uint8_t PITCH_BEND = 0xE0;
```

You can now use `sendMIDI(CC, 3, 80, 64);` which will make the code much easier to read.
When writing code, it's always a good idea to keep so-called *magic numbers* to a minimum. These are seemingly arbitrary numeric literals in your code that don't have a clear meaning.
For example, this code snippet plays a chromatic glissando (all keys, one after the other) on an honky-tonk piano:

```
sendMIDI(0xC0, 1, 4);
for (uint8_t i = 21; i <= 108; i++) {
   sendMIDI(0x90, 1, i, 64);
   delay(100);
```

```
    sendMIDI(0x80, 1, i, 64);
}
```

To someone who has never seen the code, or someone who doesn't know all MIDI message type codes by heart, it's not clear what all these numbers mean. A much better sketch would be:

```
const uint8_t honkyTonkPiano = 4;  // GM defines the Honky-tonk Piano as instrument #4

const uint8_t note_A1 = 21;  // lowest note on an 88-key piano
const uint8_t note_C9 = 108; // highest note on an 88-key piano

uint8_t channel = 1;    // MIDI channel 1
uint8_t velocity = 64;  // 64 = mezzo forte
```

```
sendMIDI(PROGRAM_CHANGE, channel, honkyTonkPiano);
for (uint8_t note = note_A1; note <= note_C9; note++) {  // chromatic glissando over all 88 piano keys
  sendMIDI(NOTE_ON, channel, note, velocity);
  delay(100);
  sendMIDI(NOTE_OFF, channel, note, velocity);
}
```

This snippet does exactly the same thing as the previous example, but it's much easier to read and understand.

Using structs

Another approach would be to compose the MIDI message in a buffer, and then just write out that buffer. We can define a struct with the different fields of a MIDI event. Take a look at this struct:

```
typedef struct MIDI_message_3B {
  unsigned int channel : 4;   // second nibble : MIDI channel (0-15)
  unsigned int status : 3;    // first  nibble : status message
  unsigned int _msb0 : 1;     // most significant bit of status byte : should be 1 according to MIDI specification
  unsigned int data1 : 7;     // second byte   : first value
  unsigned int _msb1 : 1;     // most significant bit of first data byte : should be 0 according to MIDI specification
  unsigned int data2 : 7;     // third  byte   : second value
  unsigned int _msb2 : 1;     // most significant bit of second data byte : should be 0 according to MIDI specification
  MIDI_message_3B() : _msb0(1), _msb1(0), _msb2(0) {}  // set the correct msb's for MIDI
};
```

You might have noticed that the bit fields are in the wrong order: for example, the normal order of the status byte would be `1.mmm.cccc` with mmm the message type and cccc the channel. However, the order in our struct is `cccc.mmm.1`. To understand what's going on, you have to know that Arduinos are Little Endian. This means that the first bit field takes up the least significant bits in each byte. In other words, the bit fields within each byte are in reversed order, compared to the conventional Big Endian notation (that is used in the MIDI specification).

You can now fill up all fields of the struct, to create a valid MIDI packet. You don't have to worry about the most significant bits of each byte, bitmasking is done automatically, because of the bit fields. These bits are set to the correct value when a message is created, in the initializer list of the constructor. The only thing you need to keep in mind is that the channels are zero-based. Also note that the message types are no longer 0x80, 0x90 etc., but 0x8, 0x9 ...

```
const uint8_t NOTE_ON = 0x9;

MIDI_message_3B msg; // Create a variable called 'msg' of the 'MIDI_message_3B' type we just defined
msg.status  = NOTE_ON;
msg.channel = channel - 1;  // MIDI channels start from 0, so subtract 1
msg.data1   = note_A1;
msg.data2   = velocity;
```

Finally, you can just write out the message over the Serial port. We'll create another overload of the sendMIDI function:

```
void sendMIDI(MIDI_message_3B msg) {
  Serial.write((uint8_t *)&msg, 3);
}
```

We're using the `write(uint8_t* buffer, size_t numberOfBytes)` function. The first argument is a pointer to a buffer (or array) of data bytes to write out. The pointer points to the first element of this array. The second argument is the number of bytes to send, starting from that first element. There's one minor problem: `msg` is not an array, it's an object of type MIDI_message_3B. The `write` function expects a pointer to an array of bytes (uint8_t). To get around this, we can just take the address of `msg`, using the *address-of* operator (&) and cast it to a pointer to an array of uint8_t's using `(uint8_t*)`. We need to write out the entire MIDI packet, which is 3 bytes long, so the second argument is just `3`.
To use the function, just use:

```
sendMIDI(msg);
```

In fact, we could do even better. Now every time the sendMIDI function is called, the `msg` object is copied. This takes time and memory. To prevent it from being copied, we can pass only a reference to `msg` to the function. Here's what that looks like:

```
void sendMIDI(MIDI_message_3B &msg) {
  Serial.write((uint8_t *)&msg, 3);
}
```

```
sendMIDI(msg);
```

You can do the same thing for two-byte MIDI packets:

```
typedef struct MIDI_message_2B {
  unsigned int channel : 4;   // second nibble : MIDI channel (0-15)
  unsigned int status : 3;    // first  nibble : message type
  unsigned int _msb0 : 1;     // most significant bit of status byte : should be 1 according to
MIDI specification
  unsigned int data : 7;      // second byte    : first value
  unsigned int _msb1 : 1;     // most significant bit of first data byte : should be 0 according
to MIDI specification
  MIDI_message_2B() : _msb0(1), _msb1(0) {}  // set the correct msb's for MIDI
};

void sendMIDI(MIDI_message_2B &msg) {
  Serial.write((uint8_t *)&msg, 2);
}
```

Running status

As discussed in chapter 1, you can use running statuses to save bandwidth. The implementation is relatively easy: remember the last status byte (header) that was sent, and then compare every following status byte to this header. If it's the same status, send the data bytes only, otherwise, send the new status byte, and remember this header.
To remember the previous header, a `static` variable is used. Static variables are not destroyed when they go out of scope, so the value is retained the next time the `sendMIDI` function is executed.

```
void sendMIDI(uint8_t messageType, uint8_t channel, uint8_t data1, uint8_t data2) {
  channel--;                                    // Decrement the channel, because MIDI channel 1
                                                // corresponds to binary channel 0
  uint8_t statusByte = messageType | channel;   // Combine the messageType (high nibble)
                                                // with the channel (low nibble)
                                                // Both the message type and the channel
                                                // should be 4 bits wide
  statusByte |= 0b10000000;                     // Set the most significant bit of the status byte
  data1      &= 0b01111111;                     // Clear the most significant bit of the data
bytes
  data2      &= 0b01111111;

  static uint8_t runningHeader;
  if (statusByte != runningHeader) {            // If the new header is different from the
previous
```

```
    Serial.write(statusByte);                 // Send the status byte over Serial
    runningHeader = statusByte;               // Remember the new header
  }
  Serial.write(data1);                        // Send the data bytes over Serial
  Serial.write(data2);
}
```

```
void sendMIDI(uint8_t messageType, uint8_t channel, uint8_t data) {
  channel--;                                  // Decrement the channel, because MIDI channel 1
                                              // corresponds to binary channel 0
  uint8_t statusByte = messageType | channel; // Combine the messageType (high nibble)
                                              // with the channel (low nibble)
                                              // Both the message type and the channel
                                              // should be 4 bits wide
  statusByte |= 0b10000000;                   // Set the most significant bit of the status byte
  data       &= 0b01111111;                   // Clear the most significant bit of the data byte

  static uint8_t runningHeader;
  if (statusByte != runningHeader) {          // If the new header is different from the
previous
    Serial.write(statusByte);                 // Send over Serial
    runningHeader = statusByte;               // Remember the new header
  }
  Serial.write(data);                         // Send the data byte over Serial
}
```

Going even further, we can replace note off events by note on events with a velocity of zero:

```
void sendMIDI(uint8_t messageType, uint8_t channel, uint8_t data1, uint8_t data2) {
  if (messageType == NOTE_OFF) {              // Replace note off messages
    messageType = NOTE_ON;                    // with a note on message
    data2 = 0;                                // with a velocity of zero.
  }
  channel--;                                  // Decrement the channel, because MIDI channel 1
                                              // corresponds to binary channel 0
  uint8_t statusByte = messageType | channel; // Combine the messageType (high nibble)
                                              // with the channel (low nibble)
                                              // Both the message type and the channel
                                              // should be 4 bits wide
  statusByte |= 0b10000000;                   // Set the most significant bit of the status byte
  data1      &= 0b01111111;                   // Clear the most significant bit of the data
bytes
  data2      &= 0b01111111;

  static uint8_t runningHeader;
  if (statusByte != runningHeader) {          // If the new header is different from the
previous
    Serial.write(statusByte);                 // Send the status byte over Serial
    runningHeader = statusByte;               // Remember the new header
  }
  Serial.write(data1);                        // Send the data bytes over Serial
  Serial.write(data2);
}
```

To ensure that the receiver will know what to do with the data, even if it missed the first header byte, it is a good idea to send a header byte regularly. This can be done by remembering the time the last header was sent:

```
void sendMIDI(uint8_t messageType, uint8_t channel, uint8_t data1, uint8_t data2) {
  if (messageType == NOTE_OFF) {              // Replace note off messages
    messageType = NOTE_ON;                    // with a note on message
    data2 = 0;                                // with a velocity of zero.
  }
  channel--;                                  // Decrement the channel, because MIDI channel 1
                                              // corresponds to binary channel 0
  uint8_t statusByte = messageType | channel; // Combine the messageType (high nibble)
                                              // with the channel (low nibble)
                                              // Both the message type and the channel
                                              // should be 4 bits wide
  statusByte |= 0b10000000;                   // Set the most significant bit of the status byte
  data1      &= 0b01111111;                   // Clear the most significant bit of the data
bytes
  data2      &= 0b01111111;

  static unsigned long lastHeaderTime = millis();
  static uint8_t runningHeader;
  if (statusByte != runningHeader             // If the new header is different from the
previous
    || (millis() - lastHeaderTime) > 1000) {  // Or if the last header was sent more than 1 s
ago
    Serial.write(statusByte);                 // Send the status byte over Serial
```

```
    runningHeader = statusByte;                // Remember the new header
    lastHeaderTime = millis();
  }
  Serial.write(data1);                         // Send the data bytes over Serial
  Serial.write(data2);
}
```

MIDI Controllers

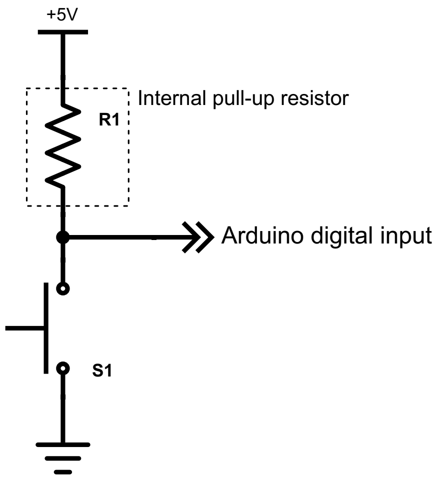
The MIDI protocol is often used for MIDI controllers, devices with physical knobs and buttons to control settings in a Digital Audio Workstation (DAW), or to enter notes in audio or music notation software. This is often much faster and more intuitive than using the mouse and keyboard for everything. MIDI controllers are also used during live performances, to control effect modules, samplers, synthesizers, DJ software, etc.
This chapter will cover how to write the code for a working MIDI controller using Arduino.

Buttons

For sending the state of a button, note events are used. When the button is pressed, a note on event is sent, when it's released, a note off event is sent.

Hardware

Connecting a button to the Arduino is pretty straightforward: Connect one lead of the button to a digital input pin, and connect the other lead to ground.



The internal pull-up resistor* of the input pin will be used, so if the button is released (if it doesn't conduct), the input will be "pulled up" to 5V, and it will read a digital 1. When the button is pressed, it connects the input pin directly to ground, so it will read a digital 0.

(*) The microcontroller has built-in pull-up resistors, to make working with buttons and open-collector outputs a whole lot easier. This resistor can be enabled in software, using `pinMode(pushButtonPin, INPUT_PULLUP)`. This means that you don't have to add a resistor externally.

Software

The MIDI controller only has to send events when the state of the button changes. To do this, the input will constantly be polled in the `loop`, and then the previous state is kept in a static variable. When the new input state does not equal the previous state, the state of the button has changed, and a MIDI event will be sent.
If the new state is low, the button has been pressed and a note on event is sent. If it's high, it has been released, and a note off event is sent.

```
const uint8_t pushButtonPin = 2;

const uint8_t channel = 1;      // MIDI channel 1
const uint8_t note = 0x3C;      // Middle C (C4)
const uint8_t velocity = 0x7F;  // Maximum velocity

void setup() {
  pinMode(pushButtonPin, INPUT_PULLUP);  // Enable the internal pull-up resistor
  Serial.begin(31250);
}

void loop() {
  static bool previousState = HIGH;                // Declare a static variable to save the
previous state
```

```
                                                    // and initialize it to HIGH (not pressed).
  bool currentState = digitalRead(pushButtonPin);  // Read the current state of the input pin
  if (currentState != previousState) {              // If the current state is different from the
previous state
    if (currentState == LOW) {                        // If the button is pressed
      sendMIDI(NOTE_ON, channel, note, velocity);       // Send a note on event
    } else {                                          // If the button is released
      sendMIDI(NOTE_OFF, channel, note, velocity);      // Send a note off event
    }
    previousState = currentState;                     // Remember the current state of the button
  }
}
```

Keep in mind that the declaration and initialization of a static local variable happen only once, the value is retained the next time the function is executed.

In principle, this approach should work, however, in practice, there will be contact bounce. When you press or release a button, it actually changes state many times really quickly, before settling to the correct state. This is called bounce, and can be a real problem if you want to reliably detect button presses. By including a timer in the code, you can make sure that the button is stable for at least a couple of tens of milliseconds before registering the state change. Here's what that looks like:

```
const unsigned long debounceTime = 25;  // Ignore all state changes that happen 25 milliseconds
                                        // after the button is pressed or released.

void loop() {
  static bool previousState = HIGH;                // Declare a static variable to save the
previous state of the input
                                                    // and initialize it to HIGH (not pressed).
  static bool buttonState   = HIGH;                // Declare a static variable to save the state
of the button
                                                    // and initialize it to HIGH (not pressed).
  static unsigned long previousBounceTime = 0;     // Declare a static variable to save the time
the button last
                                                    // changed state (bounced).
  bool currentState = digitalRead(pushButtonPin);  // Read the current state of the input pin
  if (currentState != buttonState) {                      // If the current state is different
from the button state
    if (millis() - previousBounceTime > debounceTime) {    // If the input has been stable for at
least 25 ms
      buttonState = currentState;                            // Remember the state that the
(debounced) button is in
      if (buttonState == LOW) {                              // If the button is pressed
        sendMIDI(NOTE_ON, channel, note, velocity);            // Send a note on event
      } else {                                               // If the button is released
        sendMIDI(NOTE_OFF, channel, note, velocity);           // Send a note off event
      }
    }
  }
  if (currentState != previousState) {                    // If the state of the input changed (if
the button bounces)
    previousBounceTime = millis();                         // Remember the current time
    previousState = currentState;                          // Remember the current state of the
input
  }
}
```

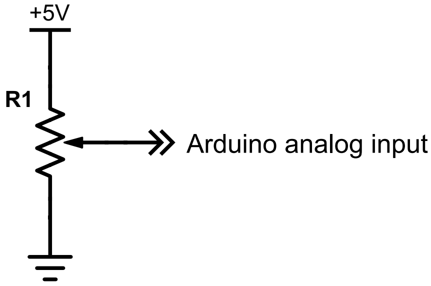
`buttonState` keeps the state of the ideal, debounced button, while `previousState` keeps the previous state of the actual input.

Potentiometers and faders

MIDI controllers often feature potentiometers and faders for continuous controllers like volume, pan, modulation, etc.

Hardware

The variable resistors (potentiometers or faders) are just used in a voltage divider configuration, with the two outer pins connected to ground and 5V, and the center pin connected to an analog input pin on the Arduino. Keep in mind that you need a potentiometer with a linear taper (not a



logarithmic or audio taper).

Control Change

For most continuous controllers, control change events are used. Most software only supports 7-bit controllers. This allows for a total of 1920 controllers (120 on each of the 16 MIDI channels).

A continuous controller can be implemented as follows: sample the analog input in the loop, convert from the 10-bit analog value to a 7-bit Control Change value, if it's a different value than last time, send a control change message with the new value.

```
const uint8_t analogPin = A0;

const uint8_t channel = 1;        // MIDI channel 1
const uint8_t controller = 0x10;  // General Purpose Controller 1

void setup() {
  Serial.begin(31250);
}

void loop() {
  static uint8_t previousValue = 0b10000000;       // Declare a static variable to save the
previous CC value
                                                   // and initialize it to 0b10000000 (the most
significant bit is set,
                                                   // so it is different from any possible 7-bit
CC value).

  uint16_t analogValue = analogRead(analogPin);    // Read the value of the analog input
  uint8_t CC_value = analogValue >> 3;             // Convert from a 10-bit number to a 7-bit
number by shifting
                                                   // it 3 bits to the right.

  if (CC_value != previousValue) {                 // If the current value is different from the
previous value
    sendMIDI(CC, channel, controller, CC_value);     // Send the new value over MIDI
    previousValue = CC_value;                        // Remember the new value
  }
}
```

The problem is that there can be quite a lot of noise on the analog inputs. So if the value fluctuates a lot, it will constantly send new CC messages, even if the knob is not being touched. To prevent this, a running average filter can be used on the input.

```
const uint8_t averageLength = 8;  // Average the analog input over 8 samples (maximum = 2^16 /
2^10 = 2^6 = 64)

void loop() {
  static uint8_t previousValue = 0b10000000;     // Declare a static variable to save the
previous CC value
                                                 // and initialize it to 0b10000000 (the most
significant bit is set,
                                                 // so it is different from any possible 7-bit CC
value).

  uint16_t analogValue = analogRead(analogPin);  // Read the value of the analog input
  analogValue = runningAverage(analogValue);     // Average the value
  uint8_t CC_value = analogValue >> 3;           // Convert from a 10-bit number to a 7-bit
number by shifting
                                                 // it 3 bits to the right.

  if (CC_value != previousValue) {               // If the current value is different from the
previous value
    sendMIDI(CC, channel, controller, CC_value);   // Send the new value over MIDI
    previousValue = CC_value;                      // Remember the new value
  }
}

uint16_t runningAverage(uint16_t value) {  // https://playground.arduino.cc/Main/RunningAverage
```

```
  static uint16_t previousValues[averageLength];
  static uint8_t index = 0;
  static uint16_t sum = 0;
  static uint8_t filled = 0;

  sum -= previousValues[index];
  previousValues[index] = value;
  sum += value;
  index++;
  index = index % averageLength;
  if (filled < averageLength)
    filled++;

  return sum / filled;
}
```

Pitch Bend

If a higher resolution is required, for example for volume faders, pitch bend events are used. This means that they have a 14-bit accuracy, however, most devices only use the 10 most significant bits. There can be only one pitch bend controller on each of the 16 MIDI channels.

The code is pretty similar to the previous example. Just shift the value 4 bits to the left instead of 3 bits to the right, and send a pitch bend message instead of a control change message. Also note that some of the variables are now of larger data types, to accommodate the 14-bit pitch bend values.
To send the 14-bit pitch bend value, it has to be split up into two 7-bit data bytes. This can be acchieved by shifting it 7 bits to the right, to get the 7 most significant bits. The `sendMIDI` function takes care of the bit masking of the 7 least significant bits.

```
const uint8_t analogPin = A0;

const uint8_t channel = 1;        // MIDI channel 1

void setup() {
  Serial.begin(31250);
}

const uint8_t averageLength = 16;  // Average the analog input over 16 samples (maximum = 2^16 / 
2^10 = 2^6 = 64)

void loop() {
  static uint16_t previousValue = 0x8000;        // Declare a static variable to save the 
previous value
                                                 // and initialize it to 0x8000 (the most 
significant bit is set,
                                                 // so it is different from any possible 14-bit 
pitch bend value).

  uint16_t analogValue = analogRead(analogPin);  // Read the value of the analog input
  analogValue = runningAverage(analogValue);     // Average the value
  uint16_t value = analogValue << 4;             // Convert from a 10-bit number to a 14-bit 
number by shifting
                                                 // it 4 bits to the left (adds 4 padding zeros 
to the right).

  if (value != previousValue) {                        // If the current value is different from 
the previous value
    sendMIDI(PITCH_BEND, channel, value, value >> 7);    // Send the new value over MIDI (split 
up into two 7-bit bytes)
    previousValue = value;                               // Remember the new value
  }
}

uint16_t runningAverage(uint16_t value) {  // https://playground.arduino.cc/Main/RunningAverage
  static uint16_t previousValues[averageLength];
  static uint8_t index = 0;
  static uint16_t sum = 0;
  static uint8_t filled = 0;

  sum -= previousValues[index];
  previousValues[index] = value;
  sum += value;
  index++;
  index = index % averageLength;
  if (filled < averageLength)
    filled++;

  return sum / filled;
}
```

Rotary encoders

The disadvantage of potentiometers is that the computer can't change their position. For example, if you have a potentiometer mapped to a plugin parameter, and you select a different plugin, the potentiometer doesn't automatically move to the position of the new pluggin parameter's value. Even worse, if you accidentally touch the potentiometer, it will overwrite the parameter with the position of potentiometer, regardless of the value it had before.

One solution is to use rotary encoders. This is a relative or incremental type of rotary knob, which means that it doesn't have an absolute position, it only sends incremental position changes when moved. When the encoder is turned two ticks to the right, it sends a value of +2, when it's turned 5 ticks to the left, it sends a value of -5.

Hardware

Connect the common pin of the rotary encoder to ground, and connect the A and B pins to digital input pins (preferably interrupt capable pins) of the Arduino. As a hardware debouncing measure, you could add an RC low-pass filter.

Software

The easiest way to read a rotary encoder is to use a library. This ensures compatibility on pretty much all boards, and many of these libraries are much more efficient than writing the ISR code yourself. My personal favorite is the PJRC Encoder library.

```
#include <Encoder.h>  // Include the PJRC Encoder library

const uint8_t channel = 1;        // MIDI channel 1
const uint8_t controller = 0x10;  // General Purpose Controller 1

Encoder encoder (2, 3);  // A rotary encoder connected to pins 2 and 3

void setup() {
  Serial.begin(31250);
}

void loop() {
  static int32_t previousPosition = 0;                // A static variable for saving the previous
encoder position
  int32_t position = encoder.read();                  // Read the current encoder position
  int32_t difference = position - previousPosition;  // Calculate the relative movement
  if (difference != 0) {                              // If the encoder was moved
    sendMIDI(CC, channel, controller, difference);   // Send the relative position change over
MIDI
    previousPosition = position;                      // Remember the current position as the
previous position
  }
}
```

Most rotary encoders send 4 pulses for every physical 'tick' (indent). It makes sense to divide the number of pulses by 4 before sending it over MIDI. Keep in mind that is a floor division, so we can't just update `previousPosition` with position without losing pulses. For example, if the current position is 6, and the previous position is 0, difference will be 6 pulses. 6 / 4 = 1 complete tick. Then the previous position will be set to 6. However, only 1 tick, so 4 pulses, has been sent, and 6 % 4 = 2 pulses have just been lost.
The solution is very simple:

```
void loop() {
  static long previousPosition = 0;                  // A static variable for saving the previous
encoder position
  long position = encoder.read();                    // Read the current encoder position
  long difference = position - previousPosition;    // Calculate the relative movement
  difference /= 4;                                   // One tick for every 4 pulses
  if (difference != 0) {                             // If the encoder was moved
    sendMIDI(CC, channel, controller, difference);  // Send the relative position change over
MIDI
    previousPosition += difference * 4;              // Add the pulses sent over MIDI to the
previous position
  }
}
```

There are three ways to encode negative position changes into a 7-bit MIDI data byte:

1. Two's complement

2. Signed magnitude
3. Offset binary

On the Arduino, all signed numbers are represented as two's complement. So sending a two's complement number over MIDI is as simple as just sending (the 7 least significant bits of) the signed variable.
In signed magnitude representation, bit 6 is used as a sign bit (0 = positive, 1 = negative), and the 6 least significant bits are used to store the absolute value of the signed number.
When using binary offset representation, 64 is added to the signed number to make everything positive.

Some programs don't support relative changes of more than 15 in one MIDI message, so we constrain the difference to 15.

This sketch allows you to choose what representation to use, to guarantee compatibility with most software, and also limits the relative position change per MIDI message to 15.

```
#include <Encoder.h>  // Include the PJRC Encoder library

enum relativeCCmode {
  TWOS_COMPLEMENT,
  BINARY_OFFSET,
  SIGN_MAGNITUDE
};

const uint8_t channel = 1;        // MIDI channel 1
const uint8_t controller = 0x10;  // General Purpose Controller 1

const Encoder encoder (2, 3);  // A rotary encoder connected to pins 2 and 3

const relativeCCmode negativeRepresentation = SIGN_MAGNITUDE;  // Select the way negative numbers
are represented

void setup() {
  Serial.begin(31250);
}

void loop() {
  static long previousPosition = 0;                 // A static variable for saving the previous
encoder position
  long position = encoder.read();                   // Read the current encoder position
  long difference = position - previousPosition;    // Calculate the relative movement
  difference /= 4;                                  // One tick for every 4 pulses
  difference = constrain(difference, -15, 15);      // Make sure that only 15 ticks are sent at
once
  if (difference != 0) {                            // If the encoder was moved
    uint8_t CC_value = mapRelativeCC(difference);   // Change the representation of negative
numbers
    sendMIDI(CC, channel, controller, CC_value);    // Send the relative position change over
MIDI
    previousPosition += difference * 4;             // Add the pulses sent over MIDI to the
previous position
  }
}

uint8_t twosComplementTo7bitSignedMagnitude(int8_t value) {  // Convert an 8-bit two's complement
integer to 7-bit sign-magnitude format
  uint8_t mask = value >> 7;
  uint8_t abs = (value + mask) ^ mask;
  uint8_t sign = mask & 0b01000000;
  return (abs & 0b00111111) | sign;
}

uint8_t mapRelativeCC(int8_t value) {  // Convert an 8-bit two's complement integer to a 7-bit
value to send over MIDI
  switch (negativeRepresentation) {
    case TWOS_COMPLEMENT:
      return value;  // Remember that the sendMIDI function does the bit masking, so you don't
have to worry about bit 7 being a 1.
    case BINARY_OFFSET:
      return value + 64;
    case SIGN_MAGNITUDE:
      return twosComplementTo7bitSignedMagnitude(value);
  }
}
```

Object-Oriented approach

The examples above only work for a single button, potentiometer or encoder. Just copying and pasting the code for each new component would lead to many repetitions and very messy code. That's why it's a good idea to implement the code in different classes: a class for buttons, another class for potentiometers, etc. You can then just instantiate many objects of these classes for the many buttons and knobs on your MIDI controller.

I wrote an Arduino MIDI controller library that makes this really easy. For example, this is all the code you need for a MIDI controller with 4 potentiometers, 4 buttons and 2 rotary encoders:

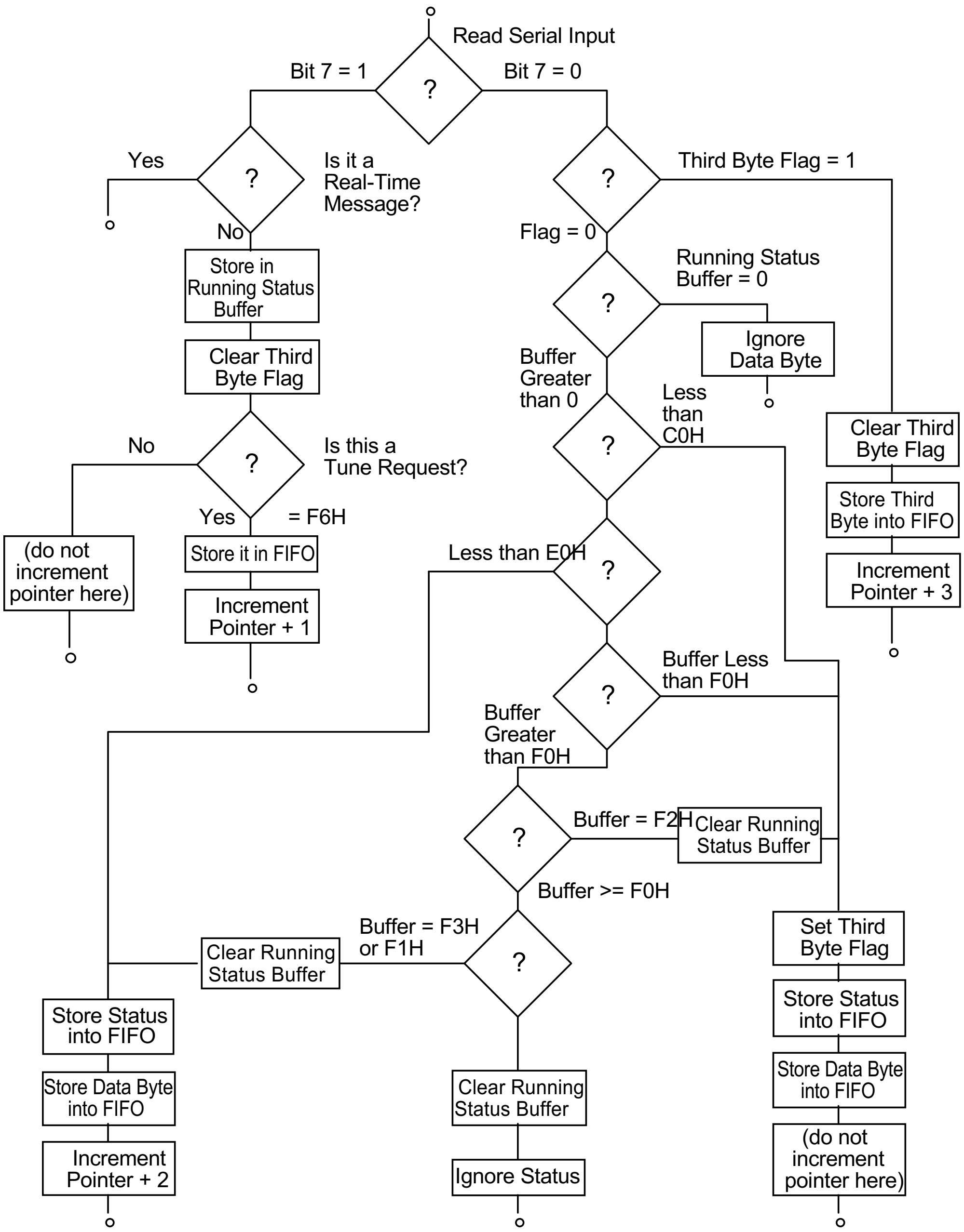
```
#include <MIDI_Controller.h>  // Include the library

/* Create four new instances of the class 'Analog' on pins A0, A1, A2 and A3,
   with controller number 0x07 (channel volume), on MIDI channels 1 through 4. */
Analog potentiometers[] = {
  {A0, 0x7, 1},
  {A1, 0x7, 2},
  {A2, 0x7, 3},
  {A3, 0x7, 4},
};
/* Create four new instances of the class 'Digital' on pins 4, 5, 6 and 7,
   with note numbers 0x10 through 0x13 (mute), on MIDI channel 1. */
Digital buttons[] = {
  {4, 0x10, 1},
  {5, 0x11, 1},
  {6, 0x12, 1},
  {7, 0x13, 1},
};
/* Create two new instances of the class 'RotaryEncoder' called 'encoders', on pins 0 & 1, and 2
& 3,
   controller numbers 0x2F and 0x30, on MIDI channel 1, at normal speed, using normal encoders
   (4 pulses per click/step), using two's complement sign representation. */
RotaryEncoder encoders[] = {
  {0, 1, 0x2F, 1, 1, NORMAL_ENCODER, TWOS_COMPLEMENT},
  {2, 3, 0x30, 1, 1, NORMAL_ENCODER, TWOS_COMPLEMENT}
};

void setup() {}

void loop() {  // Refresh all inputs
  MIDI_Controller.refresh();
}
```

As you can see, there's only the definitions of all controls, then an empty setup, and finally just a loop that refreshes all controls indefinitely. The MIDI Controller library handles everything discussed above, and even more!
It allows you to arrange controls into different banks, switch between banks, choose between many different MIDI interfaces (USB, Serial, SoftwareSerial), has support for multiplexers, button matrices, etc.

You can download the library **here** .

MIDI Input

Reading MIDI can be done using the Arduino's UART. The MIDI specification proposes an algorithm for receiving MIDI messages:



In this chapter, we won't be concerned with System or Real-Time messages. The implementation of the algorithm above is pretty straightforward. We won't use a FIFO, but handle the messages immediately.

```
void setup() {
  Serial.begin(31250);
}
```

```
void handleMIDI(uint8_t statusByte, uint8_t data1, uint8_t data2 = 0) {
  ;
}

void loop() {
  static uint8_t runningStatus = 0;
  static uint8_t data1 = 0;
  static bool thirdByte = false;

  if (Serial.available()) {
    uint8_t newByte = Serial.read();
    if (newByte & 0b10000000) {                      // Header byte received
      runningStatus = newByte;
      thirdByte = false;
    } else {
      if (thirdByte) {                               // Second data byte received
        uint8_t data2 = newByte;
        handleMIDI(runningStatus, data1, data2);
        thirdByte = false;
        return;
      } else {                                       // First data byte received
        if (!runningStatus) // no status byte
          return; // invalid data byte
        if (runningStatus < 0xC0) {                  // First data byte of Note Off/On, Key Pressure
or Control Change
          data1 = newByte;
          thirdByte = true;
          return;
        }
        if (runningStatus < 0xE0) {                  // First data byte of Program Change or Channel
Pressure
          data1 = newByte;
          handleMIDI(runningStatus, data1);
          return;
        }
        if (runningStatus < 0xF0) {                  // First data byte of Pitch Bend
          data1 = newByte;
          thirdByte = true;
          return;
        } else {
          ;                                          // System message (not implemented)
        }
      }
    }
  }
}
```

There are a few optimizations we can do. We can just check if the running status byte contains a message type for a two- or three-byte message, instead of the comparisons we have right now. Apart from that, we don't really need an extra variable for the third byte flag, we can just use bit 7 of the `data1` variable.

```
const uint8_t NOTE_OFF = 0x80;
const uint8_t NOTE_ON = 0x90;
const uint8_t KEY_PRESSURE = 0xA0;
const uint8_t CC = 0xB0;
const uint8_t PROGRAM_CHANGE = 0xC0;
const uint8_t CHANNEL_PRESSURE = 0xD0;
const uint8_t PITCH_BEND = 0xE0;

void loop() {
  static uint8_t runningStatus = 0;
  static uint8_t data1 = 0b10000000;

  if (Serial.available()) {
    uint8_t newByte = Serial.read();
    if (newByte & 0b10000000) {                        // Status byte received
      runningStatus = newByte;
      data1 = 0b10000000;
    } else {
      if (data1 != 0b10000000) {                       // Second data byte received
        handleMIDI(runningStatus, data1, newByte);
        data1 = 0b10000000;
        return;
      } else {                                         // First data byte received
        if (!runningStatus) // no status byte
          return; // invalid data byte
        if (runningStatus == PROGRAM_CHANGE
            || runningStatus == CHANNEL_PRESSURE) { // First data byte of Program Change or
Channel Pressure
          handleMIDI(runningStatus, newByte);
          return;
        } else if (runningStatus < 0xF0) {           // First data byte of Note Off/On, Key
```

```
Pressure, Control Change or Pitch Bend
          data1 = newByte;
          return;
        } else {
          ;                                         // System message (not implemented)
        }
      }
    }
  }
}
```