Floppy Drive Orchestra

University of Colorado Boulder

Independent Study 2017

Jeffery Lim

Jeffery.Lim@colorado.edu

Under supervision of Dr. Shalom Ruben

Contents

Chapter 1

Introduction

The goal of the floppy drive orchestra is to both develop a working orchestra with several floppy drives and to understand what allows the floppy drive orchestra function properly. Floppy drives are a hardware device invented in 1967 that was used to read and write information. THese floppy drives are now considered antique in the computer hardware world, and there is not much functionality to these drives. Several people, however, have abused the floppy drive to try and produce music from them, and thus, the floppy drive orchestra was born.

To understand the floppy drives, several tests were conducted to test their power consumption, physical limitation of the read/write head, and what ranges of frequencies does it play. The hardware to enable several floppy drives is explored, as well as the hardware on the Arduino. The MIDI, or musical instrument digital interface, is a special format that gives lots of information. Finally, the software to play music, as well as the software on the Arduino will be explored.

Chapter 2

Floppy Drive Characteristics

The first tests conducted were the analysis of a single floppy drive unit. Floppy drives come in three sizes: 8 in., 5.25 in., and 3.5 in. The large floppy drives come with different characteristics, however, the more modern 3.5 in. floppy drive will be used in this orchestra. The five characteristics to be explored are pinouts, power, stepper motor movements, stepper motor bandwidth, and frequency.

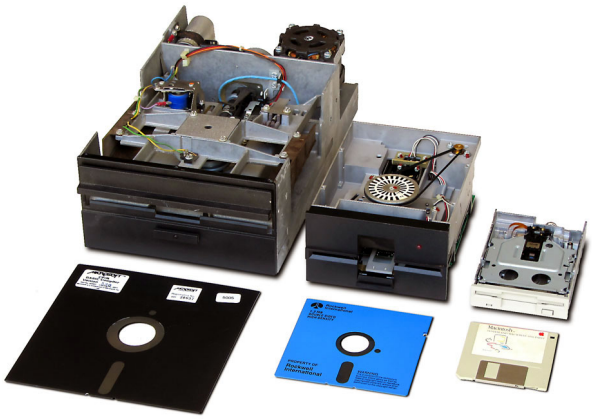


Figure 2.1: Floppy Drive of All Sizes

2.1 Floppy Pinout

The floppy pinout is shown in Figure 2.2. The bottom row, or the odd valued pins, are all grounded, where the top pins, or the even valued pins, are live. Each pin has a diferent functionality when it is grounded with the pins below. It is not necessary to connect the live pins to their respected ground pins, because the bottom row are all grounded. The actual names of the active pins are shown in Figure 2.3.

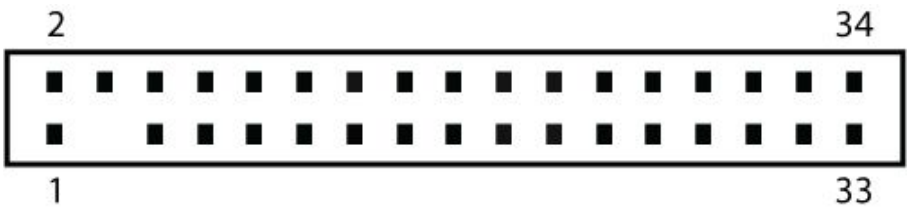


Figure 2.2: Floppy Drive Pinout

Pin	Name	Dir	Description
2	/REDWC	→	Density Select
4	n/c		Reserved
6	n/c		Reserved
8	/INDEX	←	Index
10	/MOTEA	→	Motor Enable A
12	/DRVSB	→	Drive Sel B
14	/DRVSA	→	Drive Sel A
16	/MOTEB	→	Motor Enable B
18	/DIR	→	Direction
20	/STEP	→	Step
22	/WDATE	→	Write Data
24	/WGATE	→	Floppy Write Enable
26	/TRK00	←	Track 0
28	/WPT	←	Write Protect
30	/RDATA	←	Read Data
32	/SIDE1	→	Head Select
34	/DSKCHG	←	Disk Change/Ready

Figure 2.3: Floppy Drive Pin Names

The only necessary pins to drive the floppy drive are pin number 12 or 14, 18, and 20. Pins 12 and 14 are the drive select B and A repectively. These are the drive enable pins. Some drives are B drives and others are A. In order to see the drive letter, connect pin 12 or 14 to one of the pin bottom pins: the LED will turn on for one drive letter. These pins are equivalent to an enable pin.

Pin 18 is a direction pin. The direction is what determines the direction of the motor drive. When the pin is grounded, the drive heads moves away from the pins, and when the pin is high, the drive returns towards the pins.

Pin 20 is a step pin. This pin drives the stepper motor. Every time the pin goes high, the stepper motor will go forward one tick.

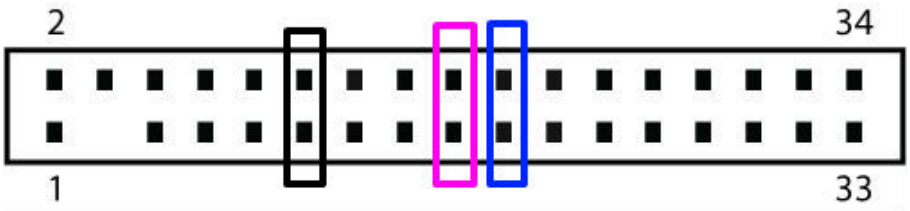


Figure 2.4: Floppy Drive Necessary Pins

2.2 Power

The floppy drive takes a mini Molex cable as seen in Figure 2.5. A mini Molex cable has 4 different lines, one red, one yellow, and two black. The two black wires are grounded, where the red line is 5V and the yellow is 12V. Older generation floppy drives used both the 5V and 12V, where the 12V was used to power the stepper motor. The modern floppy drives no longer use the 12V for stepper motors, and only use 5V. For the final floppy drive hardware, this results in only needing a 5V rail to power all of the floppy drives.

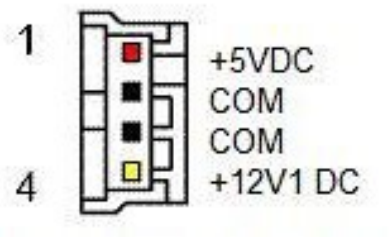


Figure 2.5: Floppy Drive Power Pinout

The power consumption was measured by using jumper cables to connect the floppy drives to a power supply, through an ammeter. Pin 12 is grounded, to enable drive B. When the drive is not running, the ammeter measured 50 mA. To test when active, pin 20 is connected to a function generator, generating a square wave. Pin 18 is grounded and set to high appropriately in order to allow the stepper motor to switch back and forth. When active, the floppy drive pulls 400 mA. The function generator was swept from all frequencies in order to see if a change in frequency would affect the current consumption, however, no change was made. This means that at most, each floppy drive requires around 400 mA.

2.3 Stepper Motor Movement

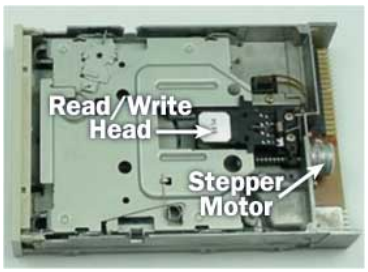


Figure 2.6: Floppy Drive With Top Off

The floppy drive read/write head has a physical limit to how far it can go. To determine this value, a 1 Hz square wave is sent through to the floppy drive's direction pin (pin 18). Pin 12 is grounded, and pin 18 is set to ground. The number of ticks is manually recorded by counting the number of audible ticks that can be heard, with the last value is when the read/write head does not move. This can be run multiple times by simply disconnected or connecting the direction pin, or pin 18.

The total number of ticks a 3.5 in. drive can go is 80, meaning the step pin (pin 20) needs to be toggled high 80 times before the drive reaches the limit. This means there needs to be a transition of high to low 80 times, or a total of 160 voltage transitions. Depending on the floppy drive, the motor will react differently when the read/write head reaches its limit. The head will either try and move, causing an audible ticking, while others will not move.

2.4 Stepper Motor Bandwidth

Since the music will be played through the read/write head, the bandwidth of the stepper motor will be a large limiting factor. For the test, a square wave from a waveform generator is con-

nected to the step pin (pin 20), and the direction pin is connected to ground and disconnected in order to allow the read/write head to switch direction.

The waveform generator is swept from 1 Hz up until the read/write head is no longer moving. The drive was able to handle up to 400 Hz, but afterwards, it was no longer consistent in terms of the speed of the drive. This test is ran again when the software was fully written. This made a significant difference because the motor was able to run at a much higher input frequency.

This limit considerably restricts what the drive can play, and higher notes will need to be addressed. However, so far, there is an assumption that the audible range contains the input frequency given to the stepper motor. In the next section, audio files of the floppy drives at each frequency are recorded and transformed.

2.5 Frequency

Although the floppy drive was unable to run past 400 Hz, it is also important to take a look at the actual audio produced from the floppy drive.

Chapter 3

Hardware

In this chapter, the hardware setup of the orchestra is built. The microcontroller chosen for this project was an Arduino. The use of shields allow building a floppy drive orchestra much simpler.

3.1 Power

As discussed in the previous chapter on floppy drive power characteristics, since each floppy drive, when running, draws 400 mA. For each floppy drive added to the system, the total power draw increases by 400 mA, so for 4 floppy drives, the total power consumption from the floppy drive system would be around 1600 mA, or 1.6 A. This means the power supply would need to be at least 2 A to take into consideration of the Arduino power consumption. This extra 400 mA would be plenty of enough to power the Arduino and run very extensive code.

The power is supplied from a 5V 2A power supply. This is directly connected into a power terminal.

3.2 Cables

All the wires bought were from Pololu, which can be found in Table B.1.

To build the wires, pre-crimped terminal wires were used. All wires bought were female wires. The power cord is simply a red and black wire connected to a 1 x 2 pin housing.
The data cables were a little bit more complicated. The floppy drive pins had a 2 x 10 pin housing. This housing would be connected to the far left side of the floppy drive pins. The only pins connected are the same as seen in Figure **??**. Pins 11 and 12 are connected to black pre-crimped terminals. Pin 18 is connected to a blue pre-crimped terminal and pin 20 is connected to a purple pre-crimped terminal. Since pin 11 is grounded, there is no need for any other ground pins to come out.

On the other end of the data cable is a 2 x 2 pin housing. The order is significant and must be followed exactly. The two black wires can be connected to any pins, as long as they are on the same side. The blue and purple pin need to be in the following order: blue on the left, and purple on the right. This needs to be consistent because all of the direction pins are on even numbered pins and the step pins are on odd numbered pins.

3.3 Schematic

The following is the schematic of the shield. The shield purchased can be found in Table B.1. There are many ways to wire the shield, however, the easiest method is the following.
With the shield, solder on all the components that come with the shield. Once finished, then following the layout of the shield as seen in Figure 3.1.

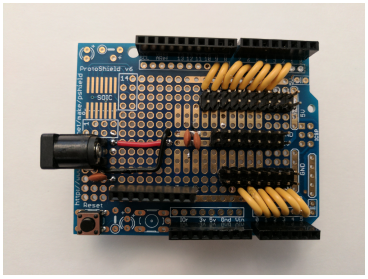


Figure 3.1: Top of the Shield

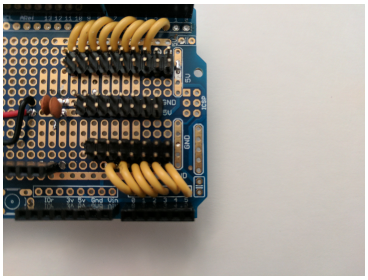


Figure 3.2: Top of the Shield

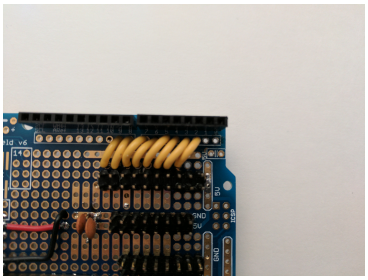


Figure 3.3: Top of the Shield

On the bottom, ensure that the wires are hooked accordingly. The grounded wires are colored black, while the 5V lines are colored red.

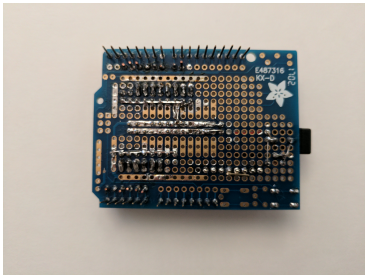


Figure 3.4: Bottom of the Shield

Chapter 4

MIDI Files

MIDI, or Musical Instrument Digital Interface, is a standard that has its own protocols, interface, and connectors. It allows a single file to contain multiple tracks for several instruments in its own channel. This allows a MIDI file to play several instruments at once, up to a maximum of 16 instruments. This advantage of the MIDI file allows multiple floppy drives to be played at once, like how a MIDI file would normally send notes to several instruments.

4.1 MIDI Notes

The MIDI interface has notes that are mapped to specific piano keys at their respected frequencies. Each note is represented by a byte, or 8 bits, however, it does not use the top bit, meaning the notes go from 0 to 127. In Figure 4.1, the respected MIDI note number has an associated piano key, since the MIDI note is based on music pitch. In Table 4.1, the actual frequency of the MIDI note number is displayed in hertz. Because of the earlier findings on the floppy drive characteristics, there are a lot of notes that may not be played. Assuming the floppy drive limit is 400 Hz, notes 0 to 67, or at least half of the range can be played.

Note	Octave										
	-1	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
C	0	12	24	36	48	**60**	72	84	96	108	120
C#	1	13	25	37	49	**61**	73	85	97	109	121
D	2	14	26	38	50	**62**	74	86	98	110	122
D#	3	15	27	39	51	**63**	75	87	99	111	123
E	4	16	28	40	52	**64**	76	88	100	112	124
F	5	17	29	41	53	**65**	77	89	101	113	125
F#	6	18	30	42	54	**66**	78	90	102	114	126
G	7	19	31	43	55	**67**	79	91	103	115	127
G#	8	20	32	44	56	**68**	80	92	104	116	
A	9	21	33	45	57	**69**	81	93	105	117	
A#	10	22	34	46	58	**70**	82	94	106	118	
B	11	23	35	47	59	**71**	83	95	107	119	

Figure 4.1: MIDI Note Number to Piano Key

Table 4.1: MIDI Note to Frequency

MIDI Note	Hz	MIDI Note	Hz	MIDI Note	Hz	MIDI Note	Hz
0	8.18	32	51.91	64	329.63	96	2093.00
1	8.66	33	55.00	65	349.23	97	2217.46
2	9.18	34	58.27	66	369.99	98	2349.32
3	9.72	35	61.74	67	392.00	99	2489.02
4	10.30	36	65.41	68	415.30	100	2637.02
5	10.91	37	69.30	69	440.00	101	2793.83
6	11.56	38	73.42	70	466.16	102	2959.96
7	12.25	39	77.78	71	493.88	103	3135.96
8	12.98	40	82.41	72	523.25	104	3322.44
9	13.75	41	87.31	73	554.37	105	3520.00
10	14.57	42	92.50	74	587.33	106	3729.31
11	15.43	43	98.00	75	622.25	107	3951.07
12	16.35	44	103.83	76	659.26	108	4186.01
13	17.32	45	110.00	77	698.46	109	4434.92
14	18.35	46	116.54	78	739.99	110	4698.64
15	19.45	47	123.47	79	783.99	111	4978.03
16	20.60	48	130.81	80	830.61	112	5274.04
17	21.83	49	138.59	81	880.00	113	5587.65
18	23.12	50	146.83	82	932.33	114	5919.91
19	24.50	51	155.56	83	987.77	115	6271.93
20	25.96	52	164.81	84	1046.50	116	6644.88
21	27.50	53	174.61	85	1108.73	117	7040.00
22	29.14	54	185.00	86	1174.66	118	7458.62
23	30.87	55	196.00	87	1244.51	119	7902.13
24	32.70	56	207.65	88	1318.51	120	8372.02
25	34.65	57	220.00	89	1396.91	121	8869.84
26	36.71	58	233.08	90	1479.98	122	9397.27
27	38.89	59	246.94	91	1567.98	123	9956.06
28	41.20	60	261.63	92	1661.22	124	10548.08
29	43.65	61	277.18	93	1760.00	125	11175.30
30	46.25	62	293.66	94	1864.66	126	11839.82
31	49.00	63	311.13	95	1975.53	127	12543.85

4.2 MIDI

MIDI data is sent serially, meaning the data is sent one bit at a time. As a result, the Arduino will be accepting the data from a serial port. In order for the MIDI data to be understood, the messages sent need to be understood. MIDI files are separated into two parts: a header chunk and a track chunk. Each chunk is separated into three sections known as type of chunk, length of the data, and data.

4.2.1 Header Chunk

The first 4 bytes are an ASCII representation of what type of chunk it is. This is always MThd for the header. The next 4 bytes is the length of the data section. The data section is a total of 6 bytes. The first 16 bits is the MIDI format. MIDI files comes in three formats: Format 0, Format 1, and Format 2. Format 0 is a single track format, where there is only one track or one instrument. Format 1 has multiple tracks, each to be played simultaneously. Format 2 has multiple tracks, each to be played independently. The next 16 bits are the number of tracks. The last 16 bits are the time divisions. There are two formats depending on the upper bit.

Table 4.2: Header Chunk

Header Chunk				
Chunk Type			Length	Data
ASCII (4 Bytes)	Length of Data(4 Bytes)	Format (16 bits)	Tracks (16 bits)	Division (16 bits)

If the upper bit is 0, then the value represents the number of time ticks per quarter note. If the upper bit is 1, then bits 0 to 7 represent the number of delta-time units per SMTPE frame. Bits 8 to 14 form the number of SMTPE frames per second.

4.2.2 Track Chunk

The track chunk starts out the same as the header chunk. The first 4 bytes are the ASCII representation and is always MTrk. It then gives the length of data of the data section. The last section is the data, which contains delta times and events. There are three types of events, however, the one that will be focused on are the MIDI events.

Table 4.3: Track Chunk

Track Chunk		
Chunk Type	Length	Data
ASCII (4 Bytes)	Length of Data(32 bits)	Delta Time and Events

4.2.3 MIDI Message

The events that the track chunk talks about are the MIDI messages that are important to playing notes. Each message comes in 3 bytes. The first byte are a status byte that contains a command and channel. There are several commands, but for the floppy drive orchestra, there is only two commands that are important: note on and note off. The channel bits are the MIDI channels that the message is suppose to go to.

The last two bytes are the note and velocity The velocity represents the loudness of the note. A zero velocity would have no sound, and anything above would mean to play the note. Since the floppy drive cannot produce a louder noise, we can effectively ignore it, unless it is equal to zero.

Table 4.4: MIDI Message

MIDI Message				
Status			Data	Data
Command (4 bits)		Channel(4 bits)	Note (8 bits)	Velocity (8 bits)
Note On	1001	nnnn	0xxxxxxx	0vvvvvvv
Note Off	1000	nnnn	0xxxxxxx	0vvvvvvv

4.2.4 Delta Time

Delta time is the number of ticks needed until the next event. This is a crucial part of the data stream because it lets the instrument or player know that there needs to be waiting until the next note plays. This part is not important to running the floppy drive orchestra from the computer because the player will handle the ticks, however, if the Arduino is handling everything, it needs to keep track of the ticks.

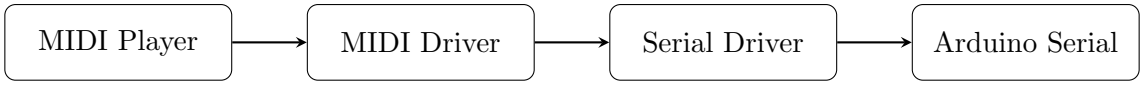
Chapter 5

Software

In order to run the software, there needs to be information to be sent from a computer to the Arduino.

5.1 Using a PC

The data flow path of the driver is as following.



5.1.1 MIDI to Serial Driver

The MIDI to serial driver is a key player in getting music to play on the floppy drives. The driver turns the MIDI outputs from the player, into data that is transmitted using serial. The driver used for this project was Hairless MIDI to Serial Bridge, found at [1].

This project has platforms for all three operating systems and has a GUI that is easy to interface with as evident in Figure 5.1. The Serial¡-¿MIDI Bridge On turns the driver on or off, being necessary with reprogramming the Arduino since the Arduino requires the serial port to be programmed. The Serial Port drop down menu will have the name of the Arduino or the COM port being used by the Arduino. The MIDI In drop down menu is where the MIDI player or piano should be located.

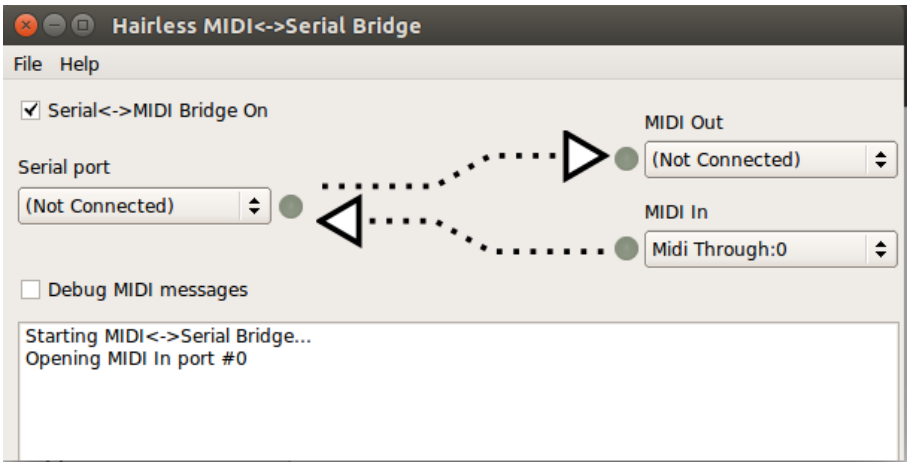


Figure 5.1: Hairless MIDISerial

The second MIDI to Serial Driver that works on Linux is known as ttymidi. This is a MIDI to serial driver that does not have a GUI and is all done on the command line. It is significantly more difficult to install and understand.

5.1.2 Virtual Keyboard

There is a program called VMPK or Virtual MIDI Piano Keyboard that sends MIDI file based on the note played [3]. This is a great way of testing the MIDI parser on the Arduino and testing the floppy drive's responsiveness to different notes. It will work on Linux or Mac.
The only necessary step is to go to EDIT -> MIDI Connections and make sure that Enable MIDI input is selected as following:

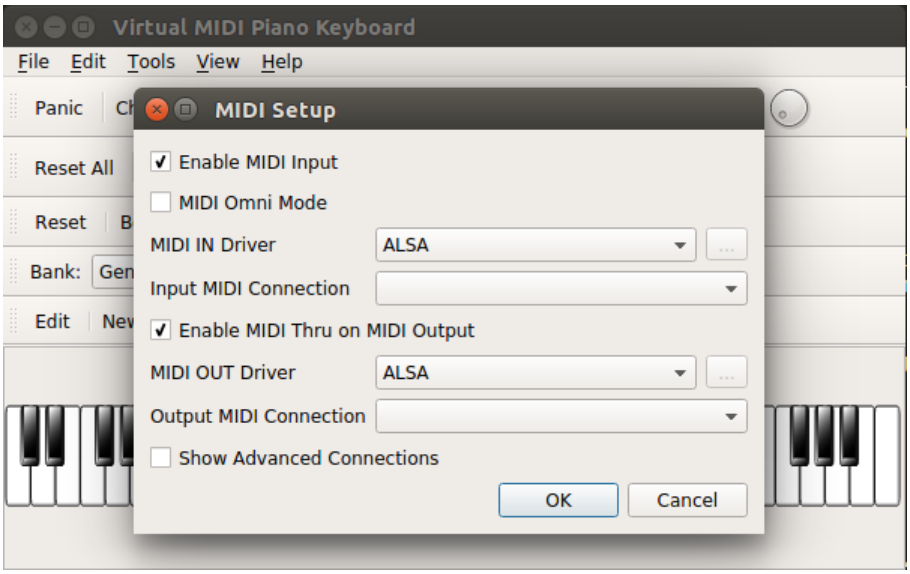


Figure 5.2: Hairess MIDI Setup

Once it is enabled, Hairless MIDI should show an option to set MIDI IN to VMPK. This will route the MIDI notes played from the piano to the Arduino.

5.1.3 MIDI Player

The MIDI player is just like any audio player, with the exception of the ability to parse through the MIDI file format. There are specifications that must be met in order to use a MIDI player with the serial driver. The Hairless MIDI to Serial Bridge website, [1] does a good job of explaining further what needs to be done in order to work with MIDI players, but does not provide any programs that work with them. The MIDI player does some back-work for the Arduino: it handles the header and the delta time ticks. It sends the note on and note off messages at the correct time and keeps track of the beat.

Hairless MIDI

In Linux, one must use ALSA, or advanced linux sound architecture. This architecture gives sound cards a good API with lots of features. The most important feature is the ability to work with MIDI. The two major programs necessary are pmidi, which can be downloaded at [2], and aconnect, which is preinstalled on most flavors of Linux.
The first step is to know which port to send pmidi data through. The following command lists out the different ports available.

```
$ pmidi -l
 Port     Client name                    Port name
 14:0     Midi Through                   Midi Through Port-0
```

Midi Through Port-0 is the port needed to send MIDI data through. The next step is to go to Hairless MIDI and set the MIDI In to Midi Through Port-0. This connects serial to Midi Through Port-0. Any data sent to this port will be sent out to the USB Port.

The final step is to have pmidi play the music file. The command to do so is:

```
$ pmidi -p 14:0 [music file]
```

The -p signifies the port number, this case, 14:0, for Midi Through Port and the music file is the directory to the .mid file.

5.1.4 ttymidi

In order to use ttymidi, first download it from the link [**?**]. Once it has been downloaded, ensure the system has libasound2, a library Used to work with ALSA. To install this library:

```
$ sudo apt-get install libasound2-dev
```

Next, edit the Makefile to add -lpthread to the line after all:

```
all:
    gcc src/ttymidi.c -o ttymidi -lasound -lpthread
```

5.2 Arduino

The code written for the project was done entirely in arduino.

5.3 SD Card(?)

Chapter 6

Arduino

6.1 Parameters

There are several global parameters used in order to do the control of the floppy drives. Each parameter are arrays of a set size, set specifically to the number of floppy drives in the system. For example, array[0] would be for the first floppy drive, and array[1] would be for the second.

```
1  // floppy - Independent Study Spring 2017 with Professor Shalom Ruben
2
3  #include <TimerOne.h>
4  #include <avr/io.h>
5  #include <avr/interrupt.h>
6
7  #define NUMDRIVES 7 //Number of Drives being used
8
9  #define MAXSTEPS 150 //Maximum number of steps the head can go
10 #define RESOLUTION 56 //us resolution of timer
11
12 //(125 cycles) * (128 prescaler) / (16MHz clock speed) = 1ms
13 //(1 * 128) / 16MHz
14
15 volatile int stepPins[NUMDRIVES + 1] = {3, 5, 7, 9, A5, A3, A1}; //ODD PINS
16 volatile int dirPins[NUMDRIVES + 1]  = {2, 4, 6, 8, A4, A2, A0}; //EVEN PINS
17
18 //drive head position
19 volatile int headPos[NUMDRIVES + 1];
20
21 //period counter to match note period
22 volatile int periodCounter[NUMDRIVES + 1];
23
24 //the note period
25 volatile int notePeriod[NUMDRIVES + 1];
26
27 //State of each drive
28 volatile boolean driveState[NUMDRIVES + 1]; //1 or 0
29
30 //Direction of each drive
31 volatile boolean driveDir[NUMDRIVES + 1]; // 1 -> forward, 0 -> backwards
32
33 //MIDI bytes
```

There are a couple of preset defines. NUMDRIVES is simply the number of drives the Arduino can use. This number is currently at 7 because of the number of pins available on the Arduino. MAXSTEPS is the maximum number of alternating ticks that can go out of the step pin. This number is around double the maximum number of ticks the read/write head can move.

RESOLUTION is how many microseconds the internal timer should be ticking at.

The first two arrays are stepPins and dirPins, an array of the pin numbers for the step and direction pins of the floppy drive. These values are preset

The array headPos stands for head position of each floppy drive. This number is to keep track of how far the head of the floppy drive is. Once it reaches the maximum number, the direction pin is flipped to move the head the other direction.

The array notePeriod is the requested period for the specific drive. The software will use this number as reference to know how often the floppy drive should be moving.

The array periodCounter is a counter used to keep a counter to match the requested period to the current period.

The next two booleans are driveState and driveDir. These keeps track of the current state of the pins that are driving the step and direction pin. These pins are changed as necessary depending on the period counter and the head position.

The last 4 parameters are each bytes that will contain the bytes received from serial.

6.2 Timer1 Library

The purpose of the timer library is to enable timer interrupts. Timer interrupts are software interrupts that stop the processor to address some service at a certain time. The way the library works is it utilizes timer1 of the Arduino and sets a resolution. This resolution determines how often the timer triggers an interrupt. If the value is set to 100 us, it will trigger every 100 us. An ISR or interrupt service routine will run a specific code.

6.3 Setup

In the setup section of the Arduino code, the Arduino runs through a couple of steps to properly ensure that the system is ready to go. The first step it goes through is initializing all the parameters to their proper values.

```
1  byte midiStatus, midiChannel, midiCommand, midiNote, midiVelocity;
2
3  //Freq = 1/(RESOLUTION * Tick Count)
4  static int noteLUT[127];
5
6  void setup()  {
7
8    //Init parameters
9    int i,j;
10   for(i = 0; i < NUMDRIVES + 1; i++){
11     driveDir[i] = 1; //Set initially at 1 to reset all drives
12     driveState[i] = 0;
13
14     periodCounter[i] = 0;
```

The next step is to generate the note lookup table. This is generated by first generating the frequency value of each MIDI note. The equation can be seen in the code, as well as on the Wikipedia page on MIDI tuning standards [5].

```
1
2       headPos[i] = 0;
3
4     }
5
6     //Midi note setup found on wikipedia page on tuning standards
7     double midi[127];
8     int a = 440; // a is 440 hz...
9     for (i = 0; i < 128; ++i)
10    {
11      midi[i] = a*pow(2, ((double)(i-69)/12));
12    }
```

The note lookup calculates the half period of each note and turns the frequency into seconds.

```
1     //1/(resolution * noteLUT) = midi -> midi*resolution = 1/noteLUT
2     for(i = 0; i < 127; i++){
3       noteLUT[i] = 1/(2*midi[i]*RESOLUTION*.000001);
4     }
```

The next section forces all the drives to reset and moves the read/write head back to the starting position.

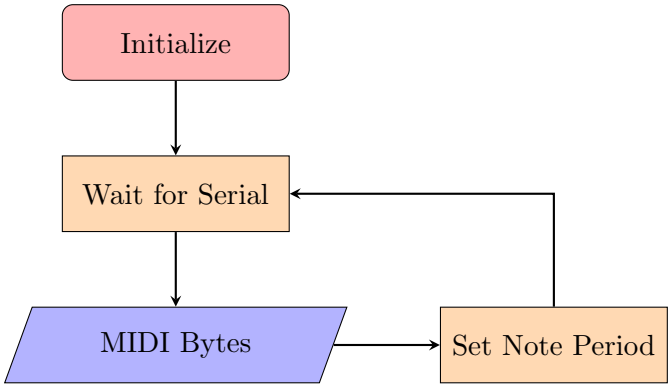
```
1     for(i = 0; i < NUMDRIVES + 1; i++){
2       pinMode(stepPins[i], OUTPUT);
3       pinMode(dirPins[i], OUTPUT);
4     }
5
6     //Drive Reset
7     for(i = 0; i < 80; i++){
8       for(j = 0; j < NUMDRIVES + 1; j++){
9         digitalWrite(dirPins[j], driveDir[j]);
10        digitalWrite(stepPins[j], 0);
```

Finally, timer1 is initialized with the appropriate service function and the serial communication is set to the default rate of 115200. It is not recommended to change this value because Hairless MIDI defaults to this value.

```
1     }
2
3     delay(1000);
4
5     //Set timer1 interrupt and initialize
6     Timer1.initialize(RESOLUTION);
```

6.4 Serial Reader

The MIDI to serial driver sends MIDI data over serial, which is then used to set the proper note period. The data flow goes as following.

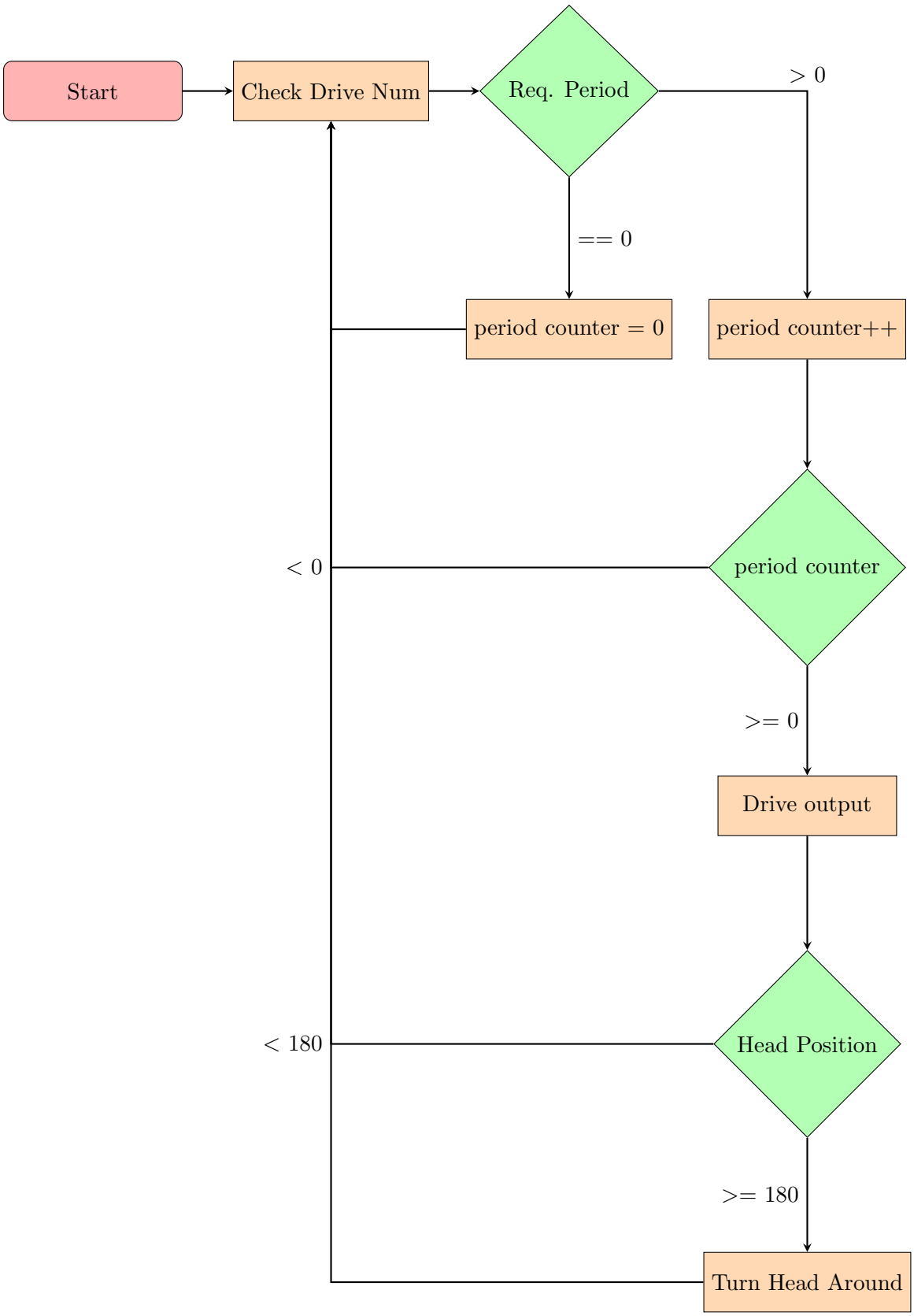


This loop is represented in the loop function of the Arduino. This code is continuously waiting for serial data, and then it is parsed. The parsing follows the MIDI file format as discussed in Section 4.2. The loop only looks for note on, note off, and channel mode messages.

```
//
//noInterrupts();
//TCCR2B = 0x00;        //Disbale Timer2 while we set it up
//TCNT2  = 248;         //Reset Timer Count to 130 out of 255
//TIFR2  = 0x00;        //Timer2 INT Flag Reg: Clear Timer Overflow Flag
//TIMSK2 = 0x01;        //Timer2 INT Reg: Timer2 Overflow Interrupt Enable
//TCCR2A = 0x00;        //Timer2 Control Reg A: Wave Gen Mode normal
//TCCR2B = 0x05;        //Timer2 Control Reg B: Timer Prescaler set to 128
//interrupts();


  //Serial for MIDI to Serial Drivers
  Serial.begin(115200);
}

void loop()  {

  //Only looking for 3 byte command
  if(Serial.available() == 3){
    midiStatus = Serial.read(); //MIDI Status
    midiNote = Serial.read(); //MIDI Note
    midiVelocity = Serial.read(); //MIDI Velocity

    //Parse out the channel and the command
    midiChannel = midiStatus & B00001111;
    midiCommand = midiStatus & B11110000;
```

6.5 ISR

The ISR or interrupt service routine is the routine that gets called every time the interrupt from the timer1 is triggered. The specific code run during this time is the code that does the driving of the floppy drives. The block diagram is as following:


Start
Check Drive Num
Req. Period
> 0
== 0
period counter = 0
period counter++
period counter
< 0
>= 0
Drive output
Head Position
< 180
>= 180
Turn Head Around

The ISR follows a checklist. It is a for loop that checks each floppy drive's status. It first checks if notePeriod, or the requested period is greater than 0. If it is 0, that means that there is not a note suppose to be playing. Otherwise, it increments the period counter. This period counter is checked if it reached the requested period. If so, it flips the step pin. This action of flipping will cause the read/write head to move. It resets the period counter so that it will start the count again. The head position is kept track as well. This is to make sure that the read/write head does not get stuck at the end. If this variable has reached the maximum value, it will revert the direction by changing the value on the direction pin.

```
1
2       //Ensure we are not going over the number of drives we have
3       if(midiChannel < NUMDRIVES + 1){
4         //Note On, set notePeriod to the midi note period
5         if(midiCommand == 0x90 && midiVelocity != 0){
6           notePeriod[midiChannel] = noteLUT[midiNote];
7         }
8         //Note off, Channel Off, velocity = 0
9         else if(midiCommand == 0x80 || (midiCommand == 0xB0 && midiNote == ...
              120) || midiVelocity == 0){
10          notePeriod[midiChannel] = 0;
11        }
12      }
13    }
14  }
15
16  //ISR(TIMER2_OVF_vect){
17    void count(){
18    int i;
19    //For each drive
20    for(i = 0; i < NUMDRIVES + 1; i++){
21      //If the desired drive is suppose to be ticking
22      if(notePeriod[i] > 0){
23        //tick the drive
24        periodCounter[i]++;
25
26        //If the drive has reached the desired period
27        if(periodCounter[i] ≥ notePeriod[i]){
28
29          //Flip the drive
30          driveState[i] ^= 1;
31          digitalWrite(stepPins[i], driveState[i]);
32
33          //reset the counter
34          periodCounter[i] = 0; //IT WAS == AND I COULDN'T FIGURE OUT WHY IT ...
                WASN'T WORKING
35
36          headPos[i]++;
```

6.6 SD Card(?)

Chapter 7

Conclusion

Appendix A

A.1 Full Arduino Code

```
1  // floppy - Independent Study Spring 2017 with Professor Shalom Ruben
2
3  #include <TimerOne.h>
4  #include <avr/io.h>
5  #include <avr/interrupt.h>
6
7  #define NUMDRIVES 7 //Number of Drives being used
8
9  #define MAXSTEPS 150 //Maximum number of steps the head can go
10 #define RESOLUTION 56 //us resolution of timer
11
12 //(125 cycles) * (128 prescaler) / (16MHz clock speed) = 1ms
13 //(1 * 128) / 16MHz
14
15 volatile int stepPins[NUMDRIVES + 1] = {3, 5, 7, 9, A5, A3, A1}; //ODD PINS
16 volatile int dirPins[NUMDRIVES + 1]  = {2, 4, 6, 8, A4, A2, A0}; //EVEN PINS
17
18 //drive head position
19 volatile int headPos[NUMDRIVES + 1];
20
21 //period counter to match note period
22 volatile int periodCounter[NUMDRIVES + 1];
23
24 //the note period
25 volatile int notePeriod[NUMDRIVES + 1];
26
27 //State of each drive
28 volatile boolean driveState[NUMDRIVES + 1]; //1 or 0
29
30 //Direction of each drive
31 volatile boolean driveDir[NUMDRIVES + 1]; // 1 -> forward, 0 -> backwards
32
33 //MIDI bytes
34 byte midiStatus, midiChannel, midiCommand, midiNote, midiVelocity;
35
36 //Freq = 1/(RESOLUTION * Tick Count)
37 static int noteLUT[127];
38
39 void setup()  {
40
41   //Init parameters
42   int i,j;
43   for(i = 0; i < NUMDRIVES + 1; i++){
44     driveDir[i] = 1; //Set initially at 1 to reset all drives
45     driveState[i] = 0;
46
47     periodCounter[i] = 0;
```

```
48      notePeriod[i] = 0;
49
50      headPos[i] = 0;
51
52    }
53
54    //Midi note setup found on wikipedia page on tuning standards
55    double midi[127];
56    int a = 440; // a is 440 hz...
57    for (i = 0; i < 128; ++i)
58    {
59      midi[i] = a*pow(2, ((double)(i-69)/12));
60    }
61
62    //1/(resolution * noteLUT) = midi -> midi*resolution = 1/noteLUT
63    for(i = 0; i < 127; i++){
64      noteLUT[i] = 1/(2*midi[i]*RESOLUTION*.000001);
65    }
66
67    //Pin Setup
68    for(i = 0; i < NUMDRIVES + 1; i++){
69      pinMode(stepPins[i], OUTPUT);
70      pinMode(dirPins[i], OUTPUT);
71    }
72
73    //Drive Reset
74    for(i = 0; i < 80; i++){
75      for(j = 0; j < NUMDRIVES + 1; j++){
76        digitalWrite(dirPins[j], driveDir[j]);
77        digitalWrite(stepPins[j], 0);
78        digitalWrite(stepPins[j], 1);
79      }
80
81      delay(50);
82    }
83
84    //Set Drive Pins to forward direction
85    for(i = 0; i < NUMDRIVES + 1; i++){
86      driveDir[i] = 0;
87      digitalWrite(dirPins[i], driveDir[i]);
88    }
89
90    delay(1000);
91
92    //Set timer1 interrupt and initialize
93    Timer1.initialize(RESOLUTION);
94    Timer1.attachInterrupt(count);
95  //
96  //noInterrupts();
97  //TCCR2B = 0x00;        //Disbale Timer2 while we set it up
98  //TCNT2  = 248;         //Reset Timer Count to 130 out of 255
99  //TIFR2  = 0x00;        //Timer2 INT Flag Reg: Clear Timer Overflow Flag
100 //TIMSK2 = 0x01;        //Timer2 INT Reg: Timer2 Overflow Interrupt Enable
101 //TCCR2A = 0x00;        //Timer2 Control Reg A: Wave Gen Mode normal
102 //TCCR2B = 0x05;        //Timer2 Control Reg B: Timer Prescaler set to 128
103 //interrupts();
104
105
106   //Serial for MIDI to Serial Drivers
107   Serial.begin(115200);
108 }
109
110 void loop()  {
```

```
111
112   //Only looking for 3 byte command
113   if(Serial.available() == 3){
114     midiStatus = Serial.read(); //MIDI Status
115     midiNote = Serial.read(); //MIDI Note
116     midiVelocity = Serial.read(); //MIDI Velocity
117
118     //Parse out the channel and the command
119     midiChannel = midiStatus & B00001111;
120     midiCommand = midiStatus & B11110000;
121
122     //Ensure we are not going over the number of drives we have
123     if(midiChannel < NUMDRIVES + 1){
124       //Note On, set notePeriod to the midi note period
125       if(midiCommand == 0x90 && midiVelocity != 0){
126         notePeriod[midiChannel] = noteLUT[midiNote];
127       }
128       //Note off, Channel Off, velocity = 0
129       else if(midiCommand == 0x80 || (midiCommand == 0xB0 && midiNote == ...
              120) || midiVelocity == 0){
130         notePeriod[midiChannel] = 0;
131       }
132     }
133   }
134 }
135
136 //ISR(TIMER2_OVF_vect){
137   void count(){
138   int i;
139   //For each drive
140   for(i = 0; i < NUMDRIVES + 1; i++){
141     //If the desired drive is suppose to be ticking
142     if(notePeriod[i] > 0){
143       //tick the drive
144       periodCounter[i]++;
145
146       //If the drive has reached the desired period
147       if(periodCounter[i] ≥ notePeriod[i]){
148
149         //Flip the drive
150         driveState[i] ^= 1;
151         digitalWrite(stepPins[i], driveState[i]);
152
153         //reset the counter
154         periodCounter[i] = 0; //IT WAS == AND I COULDN'T FIGURE OUT WHY IT ...
              WASN'T WORKING
155
156         headPos[i]++;
157         //If the drive is at the maximum step, reset its direction
158         if(headPos[i] ≥ MAXSTEPS){
159           headPos[i] = 0;
160           driveDir[i] ^= 1;
161           digitalWrite(dirPins[i], driveDir[i]);
162         }
163       }
164     }
165     else{
166       //Set Counter to 0
167       periodCounter[i] = 0;
168
169     }
170   }
171
```

```
172   // TCNT2 = 248;            //Reset Timer to 130 out of 255
173    //TIFR2 = 0x00;          //Timer2 INT Flag Reg: Clear Timer Overflow Flag
174
175  }
```

Appendix B

B.1 Bill of Materials

Table B.1: MIDI Message

BOM			
ITEM	QUANTITY	COST	LINK
Arduino R3	1	$ 24.95	https://www.adafruit.com/product/50
Arduino Proto Shield	1	$ 9.95	https://www.adafruit.com/products/2077
5V 4A Power Supply	1	$ 14.95	https://www.adafruit.com/products/1466
Breadboard-friendly 2.1mm DC barrel jack	1	$ 0.95	https://www.adafruit.com/products/373
(2.54mm) Crimp Connector Housing: 2x10-Pin	2	$ 1.99	https://www.pololu.com/product/1917
(2.54mm) Crimp Connector Housing: 1x2-Pin	2	$ 0.69	https://www.pololu.com/product/1901
(2.54mm) Crimp Connector Housing: 2x2-Pin	2	$ 0.59	https://www.pololu.com/product/1910
(2.54 mm) Female Header: 1x12-Pin	2	$ 0.79	https://www.pololu.com/product/1030
(2.54 mm) Male Header: 2×40-Pin	3	$ 1.49	https://www.pololu.com/product/966
Pre-crimped Wires 10-Pack F-F (Black)	4	$ 2.49	https://www.pololu.com/product/1810
Pre-crimped Wires 10-Pack F-F (Red)	2	$ 2.49	https://www.pololu.com/product/1812
Pre-crimped Wires 10-Pack F-F (Blue)	1	$ 2.49	https://www.pololu.com/product/1816
Pre-crimped Wires 10-Pack F-F (Purple)	1	$ 2.49	https://www.pololu.com/product/1817

Bibliography

[1] http://projectgus.github.io/hairless-midiserial/

[2] http://alsa.opensrc.org/Pmidi

[3] http://www.varal.org/ttymidi/

[4] http://vmpk.sourceforge.net/

[5] https://github.com/coon42/Floppy-Music--midis-/tree/master/midi/finished/MrSolidSnake745

[6] https://en.wikipedia.org/wiki/MIDI_tuning_standard

[7] https://www.midi.org/specifications/item/table-1-summary-of-midi-message