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 this independent study is to build a working implementation of the floppy drive orchestra and understand each process that allows it to produce music. Floppy drives are hardware devices that were invented in 1967 for storing information. Floppy drives are now ancient in comparison to current storage devices, so there are no reason to use them in modern computers. People have found alternative uses for them. The floppy drive has a stepper motor that makes an audible sound each time it moves. Since sound is the basis of music, floppy drives can produce music.

To properly understand the orchestra, the floppy drive physical characteristics are needed. Characteristics like the physical limitations of the read/write head and what range of frequencies can the stepper motor handle give insight to how sound is produced from the floppy drives. The necessary hardware to enable the floppy drive orchestra is simple with minimal work. The software that goes into running the orchestra, however, is the most complicated part. It required understanding music file format and understanding the controls necessary to operate several floppy drive in sync.

Chapter 2

Floppy Drive Characteristics

The tests conducted to understand the characteristics were done with a single floppy drive. Floppy drives come in three sizes: 8 in., 5.25 in., and 3.5 in. All different sizes give different characteristics, but for this implementation, the more modern 3.5 in. floppy drives were used. The five characteristics to be explored are the pin outs, power consumption, stepper motor movements, stepper motor bandwidth, and frequency.

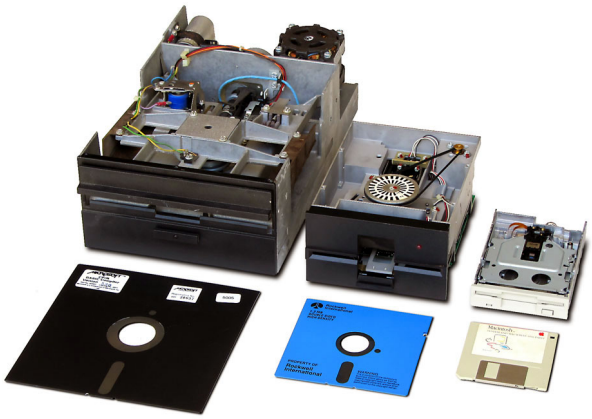


Figure 2.1: Floppy Drive of All Sizes

2.1 Floppy Pinout

The floppy drive pins are shown in Figure 2.2. All the bottom row pins are odd numbered while the top row pins are even numbered. The bottom row pins are all grounded while the top row pins are 5V. Each pin has a different functionality when it is grounded. Each function is shown in Figure 2.3.

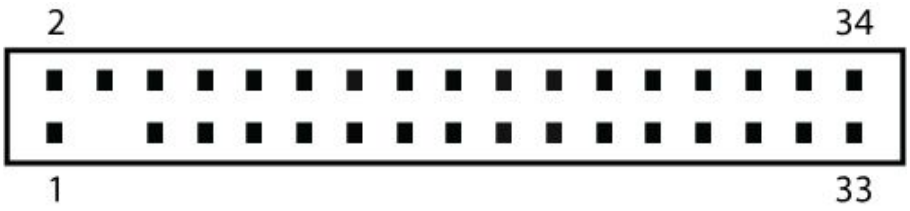


Figure 2.2: Floppy Drive Pinout

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 majority of the pins are used for memory management purposes. The only necessary pins to drive the floppy drive are pin numbers 12 or 14, 18, and 20.

Pins 12 and 14 are respectively known as drive select B and A. These pins are enable pins for the floppy drives and without connecting these pins to ground, the drive will not operate. Each drive usually has a different drive letter. In order to see which drive letter it is, connect the power (See Section 2.2 for more details on powering the floppy drives) to the floppy drive and connect pin 12 or 14 to any of the ground pins. The LED in the front of the drive should turn on for one them.

Pin 18 is the direction pin. The direction pin determines which direction the read/write head is moving. When the direction pin is grounded, the drive will move away from the pins and vice versa when the pin is held high.

Pin 20 is the step pin. This pin drives the stepper motor. Every time the voltage at the pin transitions from low to high, the stepper motor will go forward one tick.

Figure 2.4 highlights the locations of the pins for operating a single floppy drive. All the drives tested were B drives.

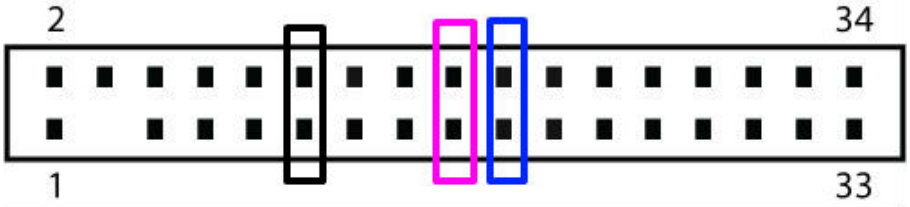


Figure 2.4: Floppy Drive Necessary Pins

2.2 Power

The floppy drive is powered by a mini Molex cable. A mini Molex cable, as seen in Figure 2.5, has one red, one yellow, and two black wires. The two black wires are grounds, 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. Modern floppy drives no longer use the 12V and use the 5V for everything. This means there is no need for any regulation, and as long as the power supply provides enough current, 5V is sufficient.

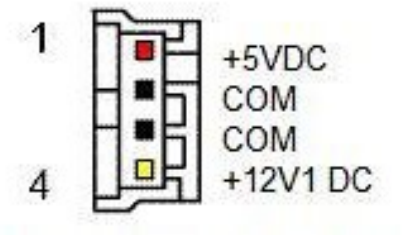


Figure 2.5: Floppy Drive Power Pinout

The power consumption was measured by using jumper cables to connect the floppy drives to a power supply with an ammeter to measure the current. Pin 12 was grounded to enable drive B and a function generator was connected to pin 20. Without turning on the function generator, the idle current is 50 mA. To test when active, pin 18 was connected to ground and disconnected in order to allow for the read/write head to continue moving. The function generator was swept to multiple frequencies to see if the input frequency changed the power consumption. When active, the floppy drive pulls 400 mA regardless of the frequency of the input. Each floppy drive that is added to the system will require an additional 400 mA to the power budget.

2.3 Stepper Motor Movement

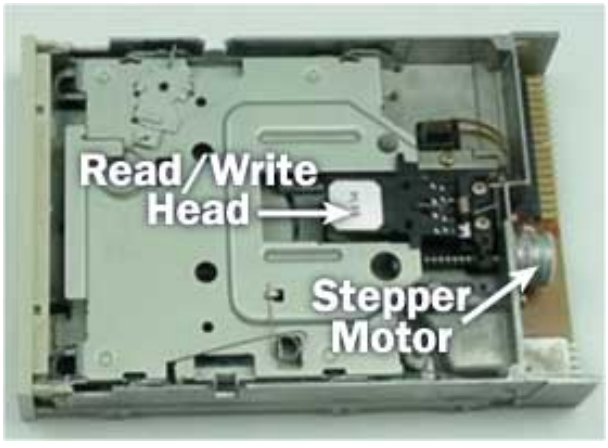


Figure 2.6: Floppy Drive With Top Off

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 step pin (pin 20). The drive enable pin (pin 12) is grounded, and the 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 (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.

One interesting result from this experiment was that each floppy drive has a different behavior when the read/write head reached its limit. Some floppy drives will try to move the head further, which results in the head to continuously tick. Other floppy drives will not make any noise and will stay in place.

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 connected 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. It is not quite clear why the floppy drive was able to run at higher speeds with the software. It is possible that the cable used in the experiment with the function generator may have caused the signals to deteriorate once they reached the floppy drive pins.

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 using the function generator, it is also important to take a look at the actual audio produced from the floppy drive. The experiment is to understand whether or not the note being played from the floppy drive has the same frequency as what is being pushed through the floppy.

Because of scheduling conflicts, an analysis of the recordings were not done. The test to be done will be discussed instead.

The first test is to be conducted in an anechoic chamber so that any stray sounds is absorbed by the room. The recording can be done all at once, however, after the recording, the audio file needs to be clipped appropriately so each note is analyzed separately. The test plays the same note at different octaves. If the note played is C, then the first note to be played would be C-1, followed by C0, C1, etc. The last note played would be C9. A Fast Fourier Transform of each note will reveal whether or not the audio file's peak frequency is the same as the notes actual frequency. This relationship between the note and the frequency can be found in Figure 4.1 and Table 4.1.

From observations, depending on the direction that the read/write head is going, it makes a different sound. This means that if a note is being held for a long time, there will actually be an audible difference as the head goes back and forth. What is not obvious, however, is how different the two are. The second test is to figure out how different the sounds are. The test is similar to the previous one, however, special care must be taken so that the initial direction of the head is known. Once the recordings are finished, the audio files need to be clipped to separate the recordings of each direction.

Chapter 3

Hardware

The Arduino provides a lot of flexibility with the shields that interface with it. The shield used in this project provided the necessary power and connections to the floppy drive. The parts and components used can be found in the Bill of Materials found in Appendix C.1.

3.1 Power

As discussed in the previous chapter on floppy drive power characteristics, since each floppy drive draws a maximum of 400 mA, the total power consumption required will simply be the number of floppy drives multiplied by 400 mA. For 8 floppy drives, the total power consumption from the floppy drive system would be around 3200 mA, or 3.2 A. This means the power supply would need to be at least 4 A to take into consideration of the Arduino power consumption. This extra 800 mA would be plenty of enough to power the Arduino and run very extensive code.

The power is supplied from a 5V 4A power supply. This is directly connected into a power terminal. The original power input that comes with the Arduino cannot supply enough current because the regulator limits the current output. Instead, another power terminal needs to be soldered onto the shield. The grounds should be connected to the Arduino and the 5V should be connected to 5V on the Arduino.

3.2 Cables

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

To build the wires, pre-crimped female terminal wires were used. For the power cord to the floppy drives, a red and black wire to connected to a 1 x 2 pin housing.

The cables for the pins on the floppy drive consist of a 2 x 10 housing on the floppy drive side. 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 2.4. Pins 11 and 12 are connected to black pre-crimped wires. Pin 18 is connected to a blue pre-crimped wire and pin 20 is connected to a purple pre-crimped wire.

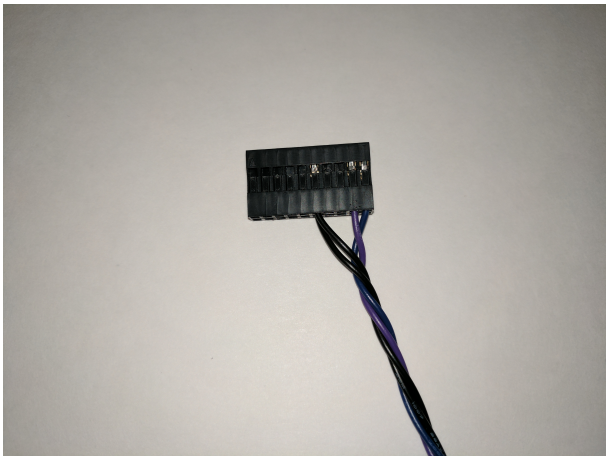


Figure 3.1: Data Cable on Floppy Side

On the other end of the data cable is a 2 x 2 pin housing. The order is significant and must be kept track of because the software needs to know which pins are direction and step. 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. If one wanted to flip the pins, they would need to change the software to match it.

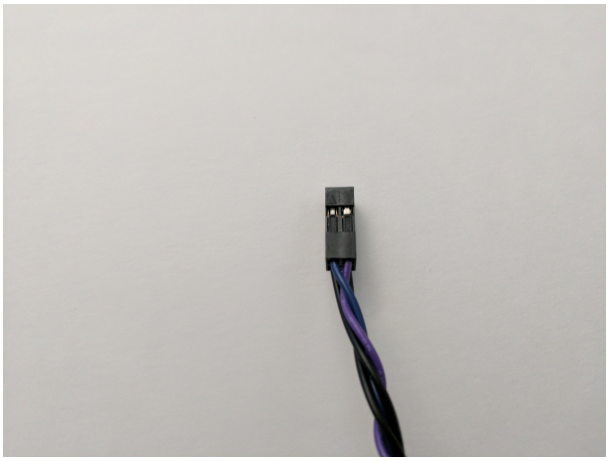


Figure 3.2: Data Cable on the Arduino Side

3.3 Schematic

To incorporate the necessary circuitry to power the floppy drives and organize the data pins, an Arduino shield makes it extremely flexible. The shield purchased can be found in Table C.1. There are many ways of setting the shield up, however, this is the way that was done.
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 3.3.

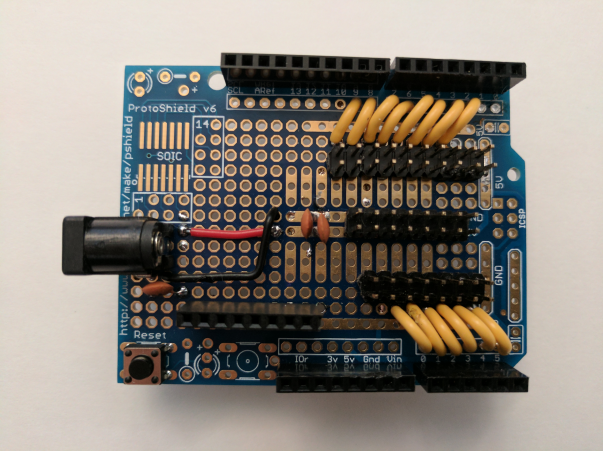


Figure 3.3: Top of the Shield

On the left of Figure 3.3 is the power terminal. This is where the 5V power supply would be plugged into.

In the center of the shield, there is a row of connections that are 5V and ground. Male headers are soldered to the center so they can be connected to the cable that power the floppy drives.

For the floppy drive pins, two rows of male headers are placed along the analog pins and the digital pins. The inner row of headers are all connected to ground. These are the headers which the drive select pins should be connected to. The other row of headers are the step and direction pin.

In Figure 3.4 and 3.5, the yellow wires are connecting the direction and step pins to the Arduino pins.

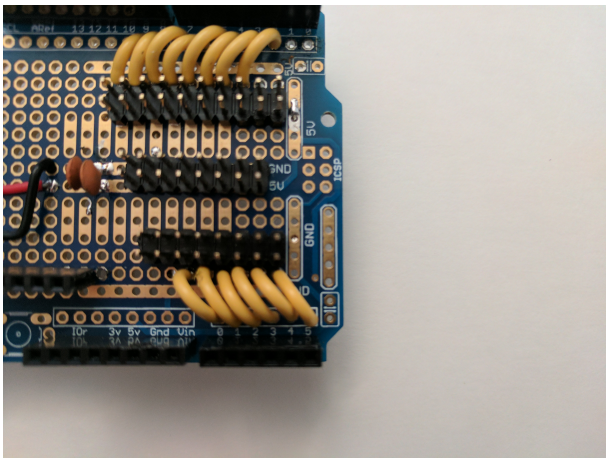


Figure 3.4: Top of the Shield

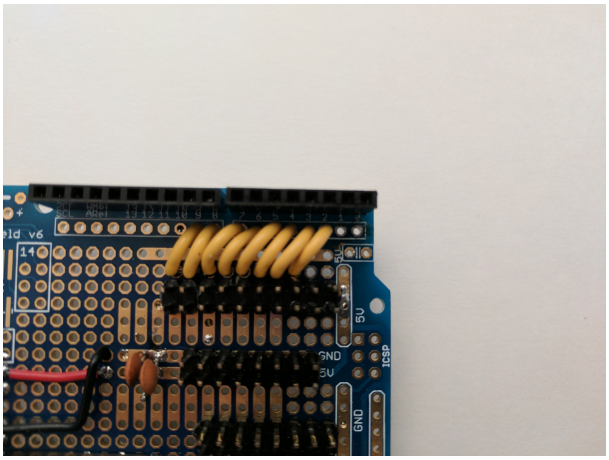


Figure 3.5: Top of the Shield

The schematic of the board is shown below.

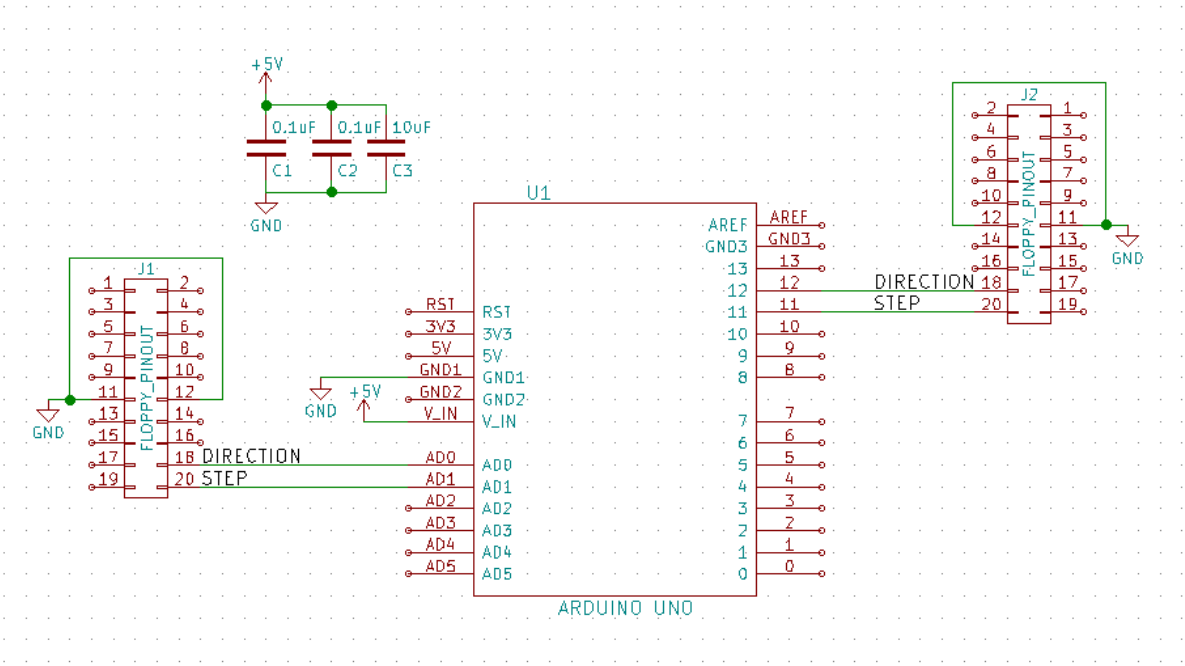


Figure 3.6: Schematic

The current iteration of the board has all step pins on the odd numbered pins while the direction pins are on the even numbered pins. This can be changed by changing the wiring, as long as it is clear which pin on the Arduino is talking to the direction and step pins.

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. The MIDI file is the main reason why the floppy drive orchestra is possible. Without it, the only way to replicate notes is by manually playing them.

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. It only uses the lower 7 bits, so the note values 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.

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

In Table 4.1, the actual frequency of the MIDI note number is displayed in hertz. Because the floppy drive maximum bandwidth is about 400 Hz, notes 0 to 67, or at least half of the range, can be played.

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 reads the data from a serial port. In order to know how to design the software to accept and understand the MIDI data, the messages need to be understood. MIDI files are separated into two parts: a header chunk and a track chunk. The header chunk gives information about the song, such as the number of tracks it has, the beats per quarter note of the song, and other information. The track chunk contains the notes of the song.

4.2.1 Header Chunk

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)

The first 4 bytes of any chunk is an ASCII representation of what type of chunk it is. This is always MThd for the header chunk. The next 4 bytes is the length of the data section. The data section is usually always 6 bytes. The next 16 bits i 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

After the format, The next 16 bits are the number of tracks. The last 16 bits are the time divisions. The value dictates the resolution of the MIDI file. In other words, it gives the relation between a MIDI tick in a song and time.

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. This value is sometimes accurate, but it is commonly recognized in the community as not usually accurate. The last section is the data. This contains the delta times and events which make up the notes that are to be played.

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 contains are the MIDI messages that contain the song content. Each message comes in 3 bytes. The first byte is a status byte that contains a command and channel. There are several commands, but for the floppy drive orchestra, there are only two significant commands: note on and note off. The channel bits are the MIDI channels that the message is being sent to.

The last two bytes are note and velocity The velocity represents the loudness of the note, where zero represents no sound. Since the floppy drive produces a constant sound, the velocity bit can be ignored.

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.

In the case that the Arduino is acting as the player, the delta time message is a very complicated process. Each tick is of a variable length. If the value is greater than or equal to 0x80, the top bit is cut off, and it is concatenated to the next delta message. This continues until the next value is not greater than or equal to 0x80.

4.3 Extra Sources

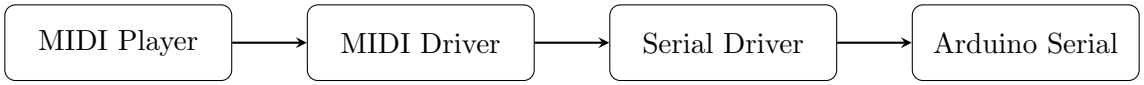
There are a lot more details to the MIDI files and how it is formated. Here are some more references that provide more detail.

Topic	URL
MIDI Messages	[4]
MIDI file outline	[2]
MIDI file format Standard	[3]
MIDI Tutorial	[6]

Chapter 5

Software

In order to run the software, there needs to be information to be sent from a computer to the Arduino. The data flow path of the driver is as following.



The MIDI player handles all the MIDI file format and essentially plays the music. The MIDI driver takes the information from the MIDI player and converts it into serial data. The serial data is sent over USB to the Arduino.

The floppy drive orchestra currently only works on Linux. The limiting factor of the orchestra is the inability to find the proper music player that will work with MIDI to Serial drivers. If the correct player can be found, this system will be able to work on Windows and Mac.

5.1 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 to 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.

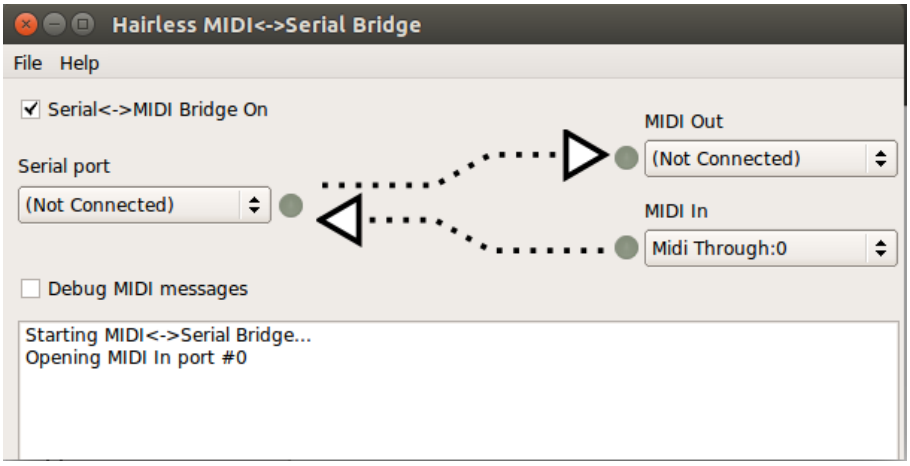


Figure 5.1: Hairless MIDISerial

Another alternative in the case that Hairless MIDI does not work is called ttymidi, found at [9]. Since ttymidi does not have a GUI, it is much more difficult to work with it. It also has some bugs in the code, however, will work on most Linux platforms. Further information on how to install and run ttymidi will be explained later on.

5.2 Virtual Keyboard

There is a program called VMPK or Virtual MIDI Piano Keyboard that sends MIDI file based on the note played [10]. 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:

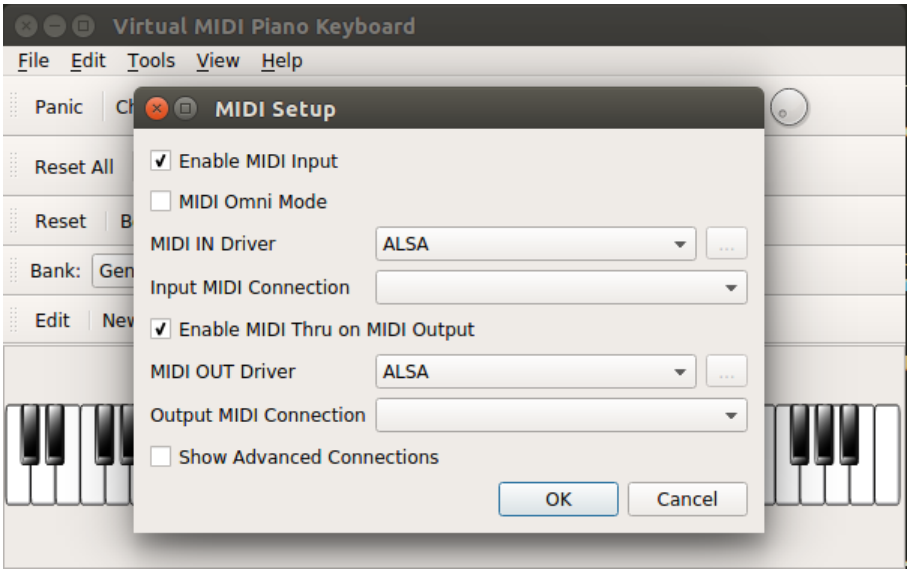


Figure 5.2: Hairess MIDI Setup

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.

An alternative keyboard is called vkeybd. It can be installed with apt-get.

```
1    $ sudo apt-get install vkeybd
```

The keyboard is similar to VMPK but has a simple GUI.

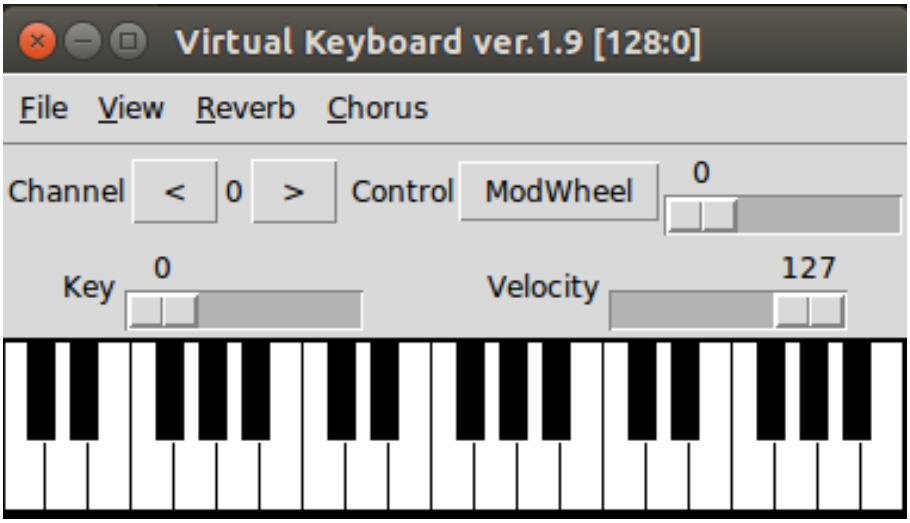


Figure 5.3: Virtual Keyboard

It can be run by using the command:

```
1    $ vkeybd
```

In order to interface it with Hairless MIDI, the MIDI IN will have a drop down option for vkeybd, similar to VMPK.

If using ttymidi, refer to the instructions found in section 5.3.2.

5.3 MIDI Player

The MIDI player is just like any audio player, with 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. The MIDI player used for this example is known as pmidi.

As mentioned before, in order to get the orchestra working with a Windows or Mac operating system, a MIDI player that can interface with Hairless MIDI must be found.

5.3.1 pmidi and Hairless MIDI

In Linux, one must use ALSA, or the 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 [8], and aconnect, which is pre-installed 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. The -l flag tells pmidi to list out all ports.

```
$ 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. Ensure that Hairless MIDI has Midi Through selected under MIDI In. This connects the Midi Through port to the serial port that the Arduino is connected to.

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.

5.3.2 pmidi and ttymidi

In order to use ttymidi, first download from [9]. 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
```

Once these steps have been taking, the next two commands will make the library and install it.

```
$ make
$ make install
```

In order to start ttymidi, ensure that the Arduino is plugged into the computer. It can be found in the dev directory and is usually named ttyACM0.

```
$ ls /dev
```

Once you have figured out which device it is, the next line initializes the drivers. The -s flag says which serial device to connect to. The -b flag sets the baud rate. This should be the same baud rate set in the Arduino code. This command occasionally causes the Arduino to get stuck, so it is necessary to reset the Arduino.

```
$ ttymidi -s /dev/ttyACM0 -b 115200
```

Now that ttymidi is configured, the next steps depends on if the driver is being connected to pmidi or a virtual keyboard.

pmidi

The next command will list out all the ports that pmidi can talk to. The -o flag asks for output ports.

```
1    $ aconnect -o
```

The result of the command should list out all the available ALSA connections on the device. Since ttymidi was initialized, the client and port number for ttymidi will be shown. These numbers will be used in conjunction with the next line in order to play music.

```
1    $ pmidi -p [client]:[port] [music file]
```

The last command will play the music through the specified port. The current bugs with ttymidi seems that if there are too many channels in a song, then ttymidi will crash. It may be necessary to rerun ttymidi with the same parameters.

Virtual Keyboard

The next command will list out all the input ports. The virtual keyboard should be found in this list. The -i flag asks for input ports.

```
1    $ aconnect -i
```

From this list, the client number and port number pertaining to the virtual keyboard should be found. The next command will look for the output ports that will redirect the notes from the virtual keyboard to the serial driver.

```
1    $ aconnect -o
```

The result of the command should list out all the available ALSA connections on the device. Since ttymidi was initialized, the client and port number for ttymidi will be shown. These numbers will be used in conjunction with the next line in order to connect the keyboard and driver.

```
1    $ aconnect [piano client]:[piano port] [ttymidi client]:[ttymidi port]
```

The command tells aconnect to connect the first port to the second. This connection causes all the commands from the first parameter to be sent to the second parameter.

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
5  #define NUMDRIVES 8 //Number of Drives being used
6
7  #define MAXSTEPS 150 //Maximum number of steps the head can go
8  #define RESOLUTION 100 //us resolution of timer
9
10 volatile int stepPins[NUMDRIVES + 1] = { A1, A3, A5, 1, 3, 5, 7, 9, 11}; ...
       //ODD PINS
11 volatile int dirPins[NUMDRIVES + 1]  = { A0, A2, A4, 2, 4, 6, 8, 10, 12}; ...
       //EVEN PINS
12
13 //drive head position
14 volatile int headPos[NUMDRIVES + 1];
15
16 //period counter to match note period
17 volatile int periodCounter[NUMDRIVES + 1];
18
19 //the note period
20 volatile int notePeriod[NUMDRIVES + 1];
21
22 //State of each drive
23 volatile boolean driveState[NUMDRIVES + 1]; //1 or 0
24
25 //Direction of each drive
26 volatile boolean driveDir[NUMDRIVES + 1]; // 1 -> forward, 0 -> backwards
27
28 //MIDI bytes
29 byte midiStatus, midiChannel, midiCommand, midiNote, midiVelocity;
30
31 //Freq = 1/(RESOLUTION * Tick Count)
32 static int noteLUT[127];
```

There are a couple of preset defines. NUMDRIVES is simply the number of drives the Arduino is connected to. 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 4 byte parameters are the MIDI messages that will be received from serial.

noteLUT is a lookup table that will be be filled later on. This lookup table contains the number of ticks required to generate a MIDI note.

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 every time the timer interrupts.

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  void setup(){
2
3    //Init parameters
4    int i,j;
5    for(i = 0; i < NUMDRIVES; i++){
6      driveDir[i] = 1; //Set initially at 1 to reset all drives
7      driveState[i] = 0;
8
9      periodCounter[i] = 0;
10     notePeriod[i] = 0;
11
12     headPos[i] = 0;
13
14   }
```

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]. This table will be used by the software to know how many times a drive should tick in order to produce the correct frequency output at the step pin.

```
1   //Midi note setup found on wikipedia page on tuning standards
2   double midi[127];
3   int a = 440; // a is 440 hz...
4   for (i = 0; i < 128; ++i)
5   {
6     midi[i] = a*pow(2, ((double)(i-69)/12));
7   }
8
9   //1/(resolution * noteLUT) = midi -> midi*resolution = 1/noteLUT
10  for(i = 0; i < 127; i++){
11    noteLUT[i] = 1/(2*midi[i]*RESOLUTION*.000001);
12  }
```

The Arduino then forces all the drives to reset and moves the read/write head back to the starting position, which is closest to the pins.

```
1   //Pin Setup
2   for(i = 0; i < NUMDRIVES; i++){
3     pinMode(stepPins[i], OUTPUT);
4     pinMode(dirPins[i], OUTPUT);
5   }
```

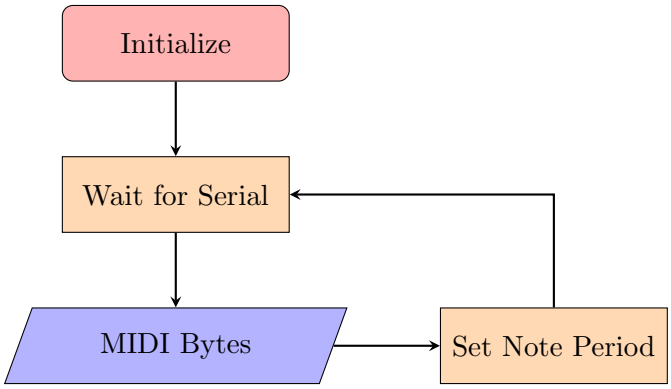
```
1   //Drive Reset
2   for(i = 0; i < 80; i++){
3     for(j = 0; j < NUMDRIVES; j++){
4       digitalWrite(dirPins[j], driveDir[j]);
5       digitalWrite(stepPins[j], 0);
6       digitalWrite(stepPins[j], 1);
7     }
8
9     delay(50);
10  }
```

Finally, timer1 is initialized with the appropriate service function and the serial communication is set to the default rate of 115200. This value can be changed to be lower, however, this may cause quick songs to not get their notes across fast enough.

```
1   Timer1.initialize(RESOLUTION);
2   Timer1.attachInterrupt(count);
3
4
5   //Serial for MIDI to Serial Drivers
6   Serial.begin(115200);
```

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.

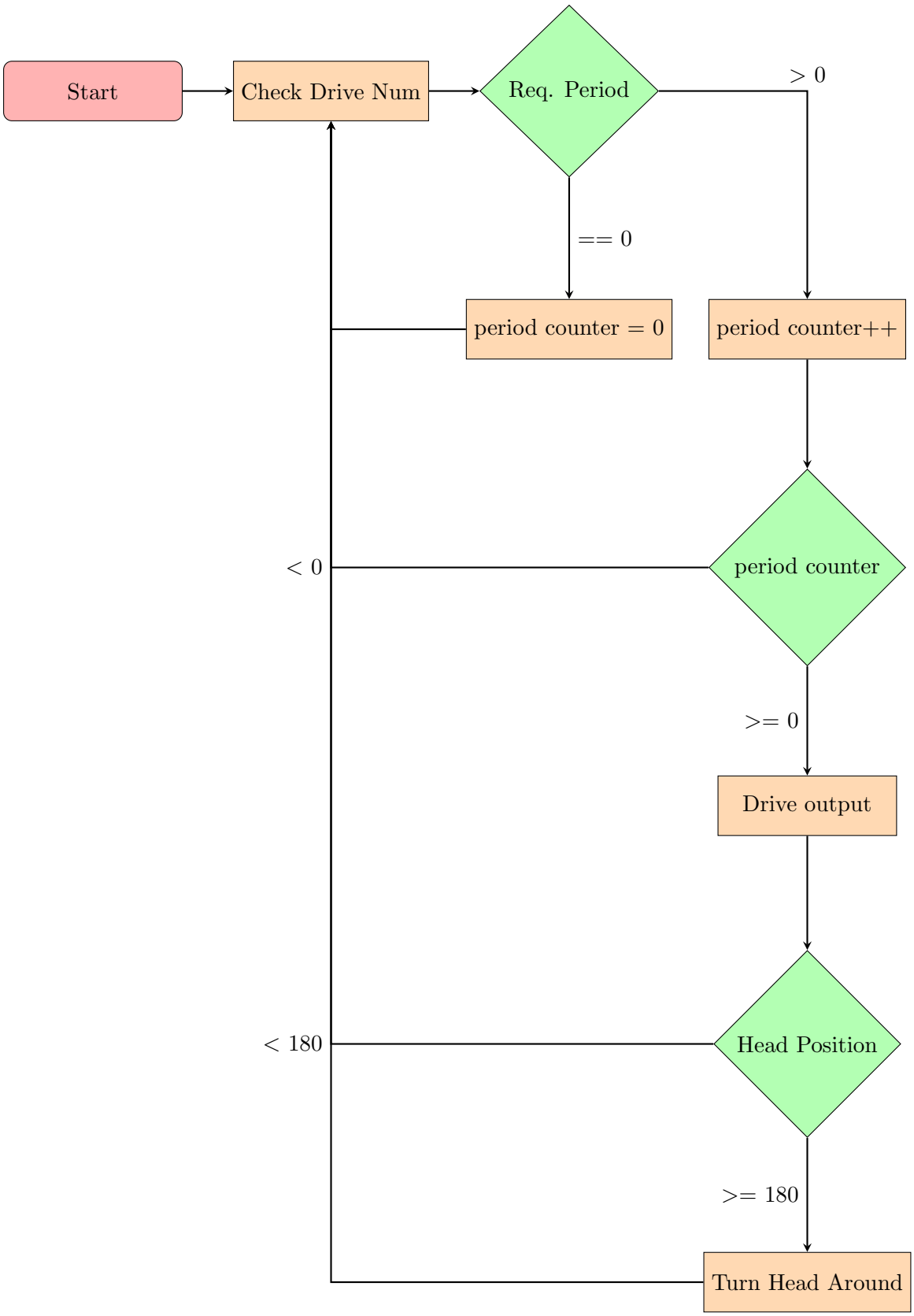


This loop is represented in the loop function of the Arduino. This code is continuously waiting for serial data, and then it is parsed. The parsing follows the MIDI file format as discussed in Section 4.2. The loop looks for note on, note off, zero velocity, and a channel mode message that occurs when the song ends.

```
1
2  void loop(){
3
4    //Only looking for 3 byte command
5    if(Serial.available() == 3){
6      midiStatus = Serial.read(); //MIDI Status
7      midiNote = Serial.read(); //MIDI Note
8      midiVelocity = Serial.read(); //MIDI Velocity
9
10     //Parse out the channel and the command
11     midiChannel = midiStatus & B00001111;
12     midiCommand = midiStatus & B11110000;
13
14     //Ensure we are not going over the number of drives we have
15     if(midiChannel < NUMDRIVES){
16     //Note On, set notePeriod to the midi note period
17       if(midiCommand == 0x90 && midiVelocity != 0){
18         notePeriod[midiChannel] = noteLUT[midiNote];
19       }
20       //Note off, Channel Off, velocity = 0
21       else if(midiCommand == 0x80 || (midiCommand == 0xB0 && midiNote == ...
             120) || (midiCommand == 0x90 && midiVelocity == 0)){
22         notePeriod[midiChannel] = 0;
23       }
24     }
25   }
26
27 }
```

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 assigned to that drive. Otherwise, it increments the period counter. If the period counter matches the notePeriod, it flips the step pin. This action of flipping will cause the read/write head to move every other time. It then resets the period counter so that it will start counting again.

The head position is also check to ensure that the read/write head does not get stuck at the ends of the track. If this variable has reached the maximum value, it will revert the direction by changing the value on the direction pin.

```
1
2  void count(){
3    int i;
4    //For each drive
5    for(i = 0; i < NUMDRIVES; i++){
6      //If the desired drive is suppose to be ticking
7      if(notePeriod[i] > 0){
8        //tick the drive
9        periodCounter[i]++;
10
11       //If the drive has reached the desired period
12       if(periodCounter[i] ≥ notePeriod[i]){
13
14         //Flip the drive
15         driveState[i] ^= 1;
16         digitalWrite(stepPins[i], driveState[i]);
17
18         //reset the counter
19         periodCounter[i] = 0;
20
21         headPos[i]++;
22         //If the drive is at the maximum step, reset its direction
23         if(headPos[i] ≥ MAXSTEPS){
24           headPos[i] = 0;
25           driveDir[i] ^= 1;
26           digitalWrite(dirPins[i], driveDir[i]);
27         }
28       }
29     }
30     else{
31       //Set Counter to 0
32       periodCounter[i] = 0;
33
34     }
35   }
36 }
```

Chapter 7

Future Work

The floppy drive orchestra has room for a lot of improvement. As it is right now, the floppy drive orchestra successfully plays MIDI files that have been altered. There was an attempt to play any music and it demonstrated that even though the high notes cannot be played, the song can still be made out. The improvements to the orchestra fall under three main sections: transitioning it to an Arduino Mega, turning the Arduino into a MIDI player, and adjusting the bandwidth.

7.1 Arduino Mega

The first step is to transition the project over to an Arduino Mega. The Mega has a significant amount of digital pins that can be used to drive 16 floppy drives. Since the MIDI file format allows for 16 channels, it would be very useful to have 16 drives available so that any song can be played.

However, another system to make up for the lack of floppy drives is to build a system that dynamically allocates tasks to each floppy drive. Instead of directing the channels to one floppy drive at a time, the software would keep track of which drives were available. When a note on or off message for a channel is received, the software will look through all the drives to see if a drive is already servicing that channel. If so, the software will pass the message to that drive. Otherwise, it will pick a free drive and assign a channel to that drive. This method can reduce the number of drives necessary except for when a song utilizes all of the channels.

7.2 Arduino as the MIDI Player

The second step is to create a MIDI player on the Arduino itself by adding a microSD card and parsing through the MIDI files. This step is the most intensive step because the Arduino is replacing the MIDI player on the computer. This step eliminates the need of a computer, making the orchestra a plug and play.

The challenges with this step is to understand the MIDI file format and parse it through properly. Because of the different formats that MIDI has, it would require a very complex state machine be able to parse through it. Another issue is that depending on the format of the MIDI file, the different tracks are spread throughout the file. This means that the entire file must be parsed through before the entire song can be played.

The microSD card used can be found in Table C.1. The library is installed by default in Arduino under the header SD. There is a specific way the pins need to be connected.

Table 7.1: MicroSD Pinout to Arduino Mega

Pin Name	MicroSD Pin	Arduino Mega Pin
Clock	Clk	52
Chip Select	CS	53
MOSI	DO	51
MISO	DI	50

Once these are connect, it is greatly helpful to go through the SD header examples. An attempt to parse through a MIDI file is done in Appendix B.

In addition to using an SD card, the Arduino Timer3 library will be needed in order to properly act as a MIDI player. It is possible that more than Timer3 will be needed because different MIDI files may require more than one tempo. If that is the case, then each track will have to have its own timer to maintain the correct beat.

7.3 Addressing the Bandwidth

Because the floppy drive has a limited bandwidth, it would be difficult to be able to play any MIDI file that has not been altered. One method discussed would be to check the note and simply play the note, just an octave lower. This method is very simple, however, may cause the song to sound more dull.

Appendix A

A.1 Arduino Uno Code

```
1  // floppy - Independent Study Spring 2017 with Professor Shalom Ruben
2
3  #include <TimerOne.h>
4
5  #define NUMDRIVES 8 //Number of Drives being used
6
7  #define MAXSTEPS 150 //Maximum number of steps the head can go
8  #define RESOLUTION 100 //us resolution of timer
9
10 volatile int stepPins[NUMDRIVES + 1] = { A1, A3, A5, 1, 3, 5, 7, 9, 11}; ...
       //ODD PINS
11 volatile int dirPins[NUMDRIVES + 1]  = { A0, A2, A4, 2, 4, 6, 8, 10, 12}; ...
       //EVEN PINS
12
13 //drive head position
14 volatile int headPos[NUMDRIVES + 1];
15
16 //period counter to match note period
17 volatile int periodCounter[NUMDRIVES + 1];
18
19 //the note period
20 volatile int notePeriod[NUMDRIVES + 1];
21
22 //State of each drive
23 volatile boolean driveState[NUMDRIVES + 1]; //1 or 0
24
25 //Direction of each drive
26 volatile boolean driveDir[NUMDRIVES + 1]; // 1 -> forward, 0 -> backwards
27
28 //MIDI bytes
29 byte midiStatus, midiChannel, midiCommand, midiNote, midiVelocity;
30
31 //Freq = 1/(RESOLUTION * Tick Count)
32 static int noteLUT[127];
33
34 void setup(){
35
36   //Init parameters
37   int i,j;
38   for(i = 0; i < NUMDRIVES; i++){
39     driveDir[i] = 1; //Set initially at 1 to reset all drives
40     driveState[i] = 0;
41
42     periodCounter[i] = 0;
43     notePeriod[i] = 0;
44
45     headPos[i] = 0;
```

```
46
47    }
48
49    //Midi note setup found on wikipedia page on tuning standards
50    double midi[127];
51    int a = 440; // a is 440 hz...
52    for (i = 0; i < 128; ++i)
53    {
54      midi[i] = a*pow(2, ((double)(i-69)/12));
55    }
56
57    //1/(resolution * noteLUT) = midi -> midi*resolution = 1/noteLUT
58    for(i = 0; i < 127; i++){
59      noteLUT[i] = 1/(2*midi[i]*RESOLUTION*.000001);
60    }
61
62    //Pin Setup
63    for(i = 0; i < NUMDRIVES; i++){
64      pinMode(stepPins[i], OUTPUT);
65      pinMode(dirPins[i], OUTPUT);
66    }
67
68    //Drive Reset
69    for(i = 0; i < 80; i++){
70      for(j = 0; j < NUMDRIVES; j++){
71        digitalWrite(dirPins[j], driveDir[j]);
72        digitalWrite(stepPins[j], 0);
73        digitalWrite(stepPins[j], 1);
74      }
75
76      delay(50);
77    }
78
79    //Set Drive Pins to forward direction
80    for(i = 0; i < NUMDRIVES; i++){
81      driveDir[i] = 0;
82      digitalWrite(dirPins[i], driveDir[i]);
83    }
84
85    delay(1000);
86
87    //Set timer1 interrupt and initialize
88    Timer1.initialize(RESOLUTION);
89    Timer1.attachInterrupt(count);
90
91
92    //Serial for MIDI to Serial Drivers
93    Serial.begin(115200);
94  }
95
96  void loop(){
97
98    //Only looking for 3 byte command
99    if(Serial.available() == 3){
100     midiStatus = Serial.read(); //MIDI Status
101     midiNote = Serial.read(); //MIDI Note
102     midiVelocity = Serial.read(); //MIDI Velocity
103
104     //Parse out the channel and the command
105     midiChannel = midiStatus & B00001111;
106     midiCommand = midiStatus & B11110000;
107
108     //Ensure we are not going over the number of drives we have
```

```
109       if(midiChannel < NUMDRIVES){
110       //Note On, set notePeriod to the midi note period
111         if(midiCommand == 0x90 && midiVelocity != 0){
112           notePeriod[midiChannel] = noteLUT[midiNote];
113         }
114         //Note off, Channel Off, velocity = 0
115         else if(midiCommand == 0x80 || (midiCommand == 0xB0 && midiNote == ...
              120) || (midiCommand == 0x90 && midiVelocity == 0)){
116           notePeriod[midiChannel] = 0;
117         }
118       }
119     }
120
121   }
122
123   void count(){
124     int i;
125     //For each drive
126     for(i = 0; i < NUMDRIVES; i++){
127       //If the desired drive is suppose to be ticking
128       if(notePeriod[i] > 0){
129         //tick the drive
130         periodCounter[i]++;
131
132         //If the drive has reached the desired period
133         if(periodCounter[i] ≥ notePeriod[i]){
134
135           //Flip the drive
136           driveState[i] ^= 1;
137           digitalWrite(stepPins[i], driveState[i]);
138
139           //reset the counter
140           periodCounter[i] = 0;
141
142           headPos[i]++;
143           //If the drive is at the maximum step, reset its direction
144           if(headPos[i] ≥ MAXSTEPS){
145             headPos[i] = 0;
146             driveDir[i] ^= 1;
147             digitalWrite(dirPins[i], driveDir[i]);
148           }
149         }
150       }
151       else{
152         //Set Counter to 0
153         periodCounter[i] = 0;
154
155       }
156     }
157   }
```

Appendix B

B.1 Arduino Mega Code

```
// floppy - Independent Study Spring 2017 with Professor Shalom Ruben

#include <TimerOne.h>
#include <TimerThree.h>

#include <SD.h>

#define NUMDRIVES 7 //Number of Drives being used

#define MAXSTEPS 150 //Maximum number of steps the head can go
#define RESOLUTION 100 //us resolution of timer

//(125 cycles) * (128 prescaler) / (16MHz clock speed) = 1ms
//(1 * 128) / 16MHz

volatile int stepPins[NUMDRIVES + 1] = {
  A0, A2, A4, 9, A5, A3, A1}; //ODD PINS
volatile int dirPins[NUMDRIVES + 1]  = {
  A1, A3, A5, 8, A4, A2, A0}; //EVEN PINS

volatile int flag = 1;

//drive head position
volatile int headPos[NUMDRIVES + 1];

//period counter to match note period
volatile int periodCounter[NUMDRIVES + 1];

//the note period
volatile int notePeriod[NUMDRIVES + 1];

//State of each drive
volatile boolean driveState[NUMDRIVES + 1]; //1 or 0

//Direction of each drive
volatile boolean driveDir[NUMDRIVES + 1]; // 1 -> forward, 0 -> backwards

//MIDI bytes
byte midiStatus, midiChannel, midiCommand, midiNote, midiVelocity;

//Freq = 1/(RESOLUTION * Tick Count)
static int noteLUT[127];

volatile int tick = 0;

volatile int wait = 0;

```

```
48 File song;
49 byte input[100];
50
51 void setup()  {
52
53   //Init parameters
54   int i,j;
55   for(i = 0; i < NUMDRIVES + 1; i++){
56     driveDir[i] = 1; //Set initially at 1 to reset all drives
57     driveState[i] = 0;
58
59     periodCounter[i] = 0;
60     notePeriod[i] = 0;
61
62     headPos[i] = 0;
63
64   }
65
66   //Midi note setup found on wikipedia page on tuning standards
67   double midi[127];
68   int a = 440; // a is 440 hz...
69   for (i = 0; i < 128; ++i)
70   {
71     midi[i] = a*pow(2, ((double)(i-69)/12));
72   }
73
74   //1/(resolution * noteLUT) = midi -> midi*resolution = 1/noteLUT
75   for(i = 0; i < 127; i++){
76     noteLUT[i] = 1/(2*midi[i]*RESOLUTION*.000001);
77   }
78
79   //Pin Setup
80   for(i = 0; i < NUMDRIVES + 1; i++){
81     pinMode(stepPins[i], OUTPUT);
82     pinMode(dirPins[i], OUTPUT);
83   }
84
85   //Drive Reset
86   for(i = 0; i < 80; i++){
87     for(j = 0; j < NUMDRIVES + 1; j++){
88       digitalWrite(dirPins[j], driveDir[j]);
89       digitalWrite(stepPins[j], 0);
90       digitalWrite(stepPins[j], 1);
91     }
92
93     delay(50);
94   }
95
96   //Set Drive Pins to forward direction
97   for(i = 0; i < NUMDRIVES + 1; i++){
98     driveDir[i] = 0;
99     digitalWrite(dirPins[i], driveDir[i]);
100  }
101
102  delay(1000);
103
104  //Serial for MIDI to Serial Drivers
105  Serial.begin(115200);
106
107  pinMode(53, OUTPUT);
108  SD.begin(53);
109
110  song = SD.open("getLucky.mid");
```

```
111
112   if(song){
113     Serial.print("File Found");
114   }
115
116   int header = 1;
117
118   int trackNum;
119   int fileType;
120   int timeSig;
121   int keySig;
122
123   uint32_t PPQN=0; //Parts Per Quarter Note
124   double BPM =0; //Beats Per Minute
125   double usPerTick=0; //usPerTick
126
127   int length = 0;
128
129   if(header == 1){
130     //Header
131     song.read(input, 4);
132     Serial.print(input[0], HEX);
133     Serial.print(" ");
134     Serial.print(input[1], HEX);
135     Serial.print(" ");
136     Serial.print(input[2], HEX);
137     Serial.print(" ");
138     Serial.print(input[3], HEX);
139     Serial.print("\n");
140
141     //Length of Data
142     song.read(input, 4);
143     Serial.print(input[0], HEX);
144     Serial.print(" ");
145     Serial.print(input[1], HEX);
146     Serial.print(" ");
147     Serial.print(input[2], HEX);
148     Serial.print(" ");
149     Serial.print(input[3], HEX);
150     Serial.print("\n");
151
152     //MIDI Format
153     song.read(input, 2);
154     Serial.print(input[0], HEX);
155     Serial.print(" ");
156     Serial.print(input[1], HEX);
157     Serial.print("\n");
158
159     fileType = input[1];
160
161     //Number of Tracks
162     song.read(input, 2);
163     Serial.print(input[0], HEX);
164     Serial.print(" ");
165     Serial.print(input[1], HEX);
166     Serial.print("\n");
167
168     trackNum = input[1];
169
170     //Time Divison or PPQN
171     song.read(input, 2);
172     Serial.print(input[0], HEX);
173     Serial.print(" ");
```

```
174     Serial.print(input[1], HEX);
175     Serial.print("\n");
176 
177     PPQN = ((input[0]) << 8) + input[1];
178 
179 header:
180     //Track Header
181     song.read(input, 4);
182     Serial.print(input[0], HEX);
183     Serial.print(" ");
184     Serial.print(input[1], HEX);
185     Serial.print(" ");
186     Serial.print(input[2], HEX);
187     Serial.print(" ");
188     Serial.print(input[3], HEX);
189     Serial.print("\n");
190 
191     //Track Length
192     song.read(input, 4);
193     Serial.print(input[0], HEX);
194     Serial.print(" ");
195     Serial.print(input[1], HEX);
196     Serial.print(" ");
197     Serial.print(input[2], HEX);
198     Serial.print(" ");
199     Serial.print(input[3], HEX);
200     Serial.print(" ");
201     Serial.print("\n");
202 
203     song.read(input, 1);
204     Serial.print(input[0], HEX);
205     Serial.print("\n");
206 
207     while(song.peek() == 0xFF){
208       song.read(input, 1);
209       Serial.print(input[0], HEX);
210       Serial.print(" ");
211       song.read(input, 1);
212       Serial.print(input[0], HEX);
213       Serial.print(" ");
214 
215       if(input[0] == 0x51){
216 
217         song.read(input, 1);
218 
219         Serial.print(input[0], HEX);
220         Serial.print(" ");
221 
222         length = input[0];
223 
224         song.read(input, length);
225 
226         usPerTick = ((long)input[0] << 16) + ((long)input[1] << 8) + input[2];
227         BPM = 60000000/usPerTick;
228 
229 
230         for (int i = 0; i < length; i++){
231           Serial.print(" ");
232           Serial.print(input[i], HEX);
233         }
234         Serial.print("\n");
235       }
236       else if(input[0] == 0x59){
```

```
237         song.read(input, 1);
238
239         Serial.print(input[0], HEX);
240         Serial.print(" ");
241
242         length = input[0];
243
244         song.read(input, length);
245
246         for (int i = 0; i < length; i++){
247           Serial.print(" ");
248           Serial.print(input[i], HEX);
249         }
250         Serial.print("\n");
251       }
252       else if(input[0] == 0x2F){
253         song.read(input, 1); // 0x00
254         Serial.print(input[0], HEX);
255         Serial.print("\n");
256
257         goto header;
258
259
260       }
261       else{
262         song.read(input, 1);
263
264         Serial.print(input[0], HEX);
265         Serial.print(" ");
266
267         length = input[0];
268
269         song.read(input, length);
270         for (int i = 0; i < length; i++){
271           Serial.print(" ");
272           Serial.print(input[i], HEX);
273         }
274         Serial.print("\n");
275       }
276
277       song.read(input, 1); // 0x00
278       Serial.print(input[0], HEX);
279       Serial.print("\n");
280
281     }
282   }
283
284
285   header = 0;
286   BPM = 120;
287
288   PPQN = 192;
289   //This is the calculation needed to determine how much each tick lasts
290   //A good source was found here:
291   //http://stackoverflow.com/questions/24717050/midi-tick-to-millisecond
292   usPerTick = round(1000*60000/(BPM*PPQN));
293
294
295   //Set timer1 inter60000/(BPM*PPQN) rupt and initialize
296   Timer1.initialize(RESOLUTION);
297   Timer1.attachInterrupt(count);
298
299   Timer3.initialize(usPerTick);
```

```
300   Timer3.attachInterrupt(parse);
301
302
303 }
304
305 void loop(){
306
307   while(flag == 0);
308
309   song.read(input, 2);
310   midiStatus = input[0];
311   midiNote = input[1];
312
313   Serial.print(midiStatus , HEX);
314   Serial.print(" ");
315   Serial.print(midiNote, HEX);
316   Serial.print(" ");
317
318
319   midiChannel = midiStatus & B00001111;
320   midiCommand = midiStatus & B11110000;
321
322   //This is the message for the end of a track
323   //This part is where I left on because I realized
324   //that the midi file would need to be parsed through entirely
325   if(midiStatus == 0xFF && midiNote == 0x2F){
326     while(1);
327   }
328
329   if(midiCommand == 0xC0 || midiCommand == 0xD0){
330   }
331   else{
332
333
334     song.read(input, 1);
335     midiVelocity = input[0];
336
337     Serial.println(midiVelocity, HEX);
338
339   }
340
341   //Ensure we are not going over the number of drives we have
342   if(midiChannel < NUMDRIVES + 1){
343     //Note On, set notePeriod to the midi note period
344     if(midiCommand == 0x90 && midiVelocity != 0){
345       notePeriod[midiChannel] = noteLUT[midiNote];
346     }
347     //Note off, Channel Off, velocity = 0
348     else if(midiCommand == 0x80 || (midiCommand == 0xB0 && midiNote == 120) ...
          || midiVelocity == 0){
349       notePeriod[midiChannel] = 0;
350     }
351   }
352
353   int temp = 0;
354   wait = 0;
355   while(song.peek() ≥ 0x80){
356     song.read(input, 1);
357     Serial.print(input[0], HEX);
358     Serial.print(" ");
359     wait = input[0] & 0x7F;
360     wait = wait << 7;
361   }
```

```
362   song.read(input ,1);
363   Serial.println(input[0], HEX);
364
365   wait |= input[0];
366   flag = 0;
367
368 }
369
370 void count(){
371   int i;
372   //For each drive
373   for(i = 0; i < NUMDRIVES + 1; i++){
374     //If the desired drive is suppose to be ticking
375     if(notePeriod[i] > 0){
376       //tick the drive
377       periodCounter[i]++;
378
379       //If the drive has reached the desired period
380       if(periodCounter[i] ≥ notePeriod[i]){
381
382         //Flip the drive
383         driveState[i] ^= 1;
384         digitalWrite(stepPins[i], driveState[i]);
385
386         //reset the counter
387         periodCounter[i] = 0; //IT WAS == AND I COULDN'T FIGURE OUT WHY IT ...
                WASN'T WORKING
388
389         headPos[i]++;
390         //If the drive is at the maximum step, reset its direction
391         if(headPos[i] ≥ MAXSTEPS){
392           headPos[i] = 0;
393           driveDir[i] ^= 1;
394           digitalWrite(dirPins[i], driveDir[i]);
395         }
396       }
397     }
398     else{
399       //Set Counter to 0
400       periodCounter[i] = 0;
401
402     }
403   }
404 }
405
406 //ISR for parsing through the MIDI tracks
407 void parse(){
408   //If the flag is 0, we should be ticking
409   if(flag == 0){
410     tick++;
411
412     //If the wait time from the Δ time is reached
413     if(tick ≥ wait){
414       tick = 0;
415       //Raise the flag to let the main loop that it can get the next message
416       flag = 1;
417     }
418   }
419   else{
420     tick = 0;
421   }
422
423 }
```

Appendix C

C.1 Bill of Materials

Table C.1: Bills of Materials

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
MicroSD Breakout Board	1	$ 7.50	https://www.adafruit.com/product/254

References

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

[2] MIDI File Structure. http://www.ccarh.org/courses/253/handout/smf/

[3] Standard MIDI File Format. https://www.cs.cmu.edu/~music/cmsip/readings/Standard-MIDI-file-format-updated.pdf

[4] Summary of MIDI Messages. https://www.midi.org/specifications/item/table-1-summary-of-midi-message

[5] MIDI Tuning Standard. https://en.wikipedia.org/wiki/MIDI_tuning_standard

[6] MIDI Tutorial. http://www.music-software-development.com/midi-tutorial.html

[7] Altered MIDI Music. https://github.com/coon42/Floppy-Music--midis-/tree/master/old

[8] Pmidi Source. http://alsa.opensrc.org/Pmidi

[9] ttymidi Source. http://www.varal.org/ttymidi/

[10] VMPK Source. http://vmpk.sourceforge.net/