Floppy Drive Orchestra

University of Colorado Boulder

Independent Study

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.

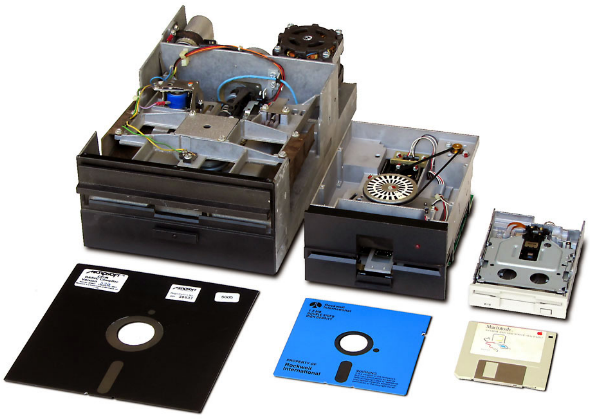


Figure 2.1: Floppy Drive of All Sizes

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.

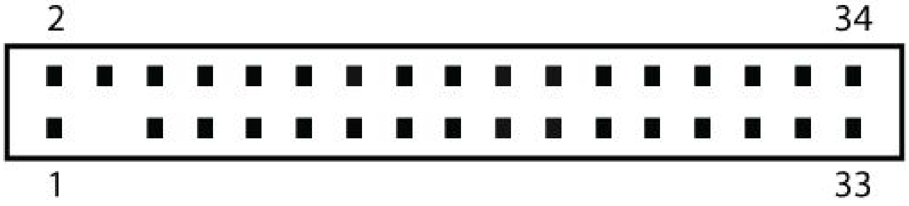
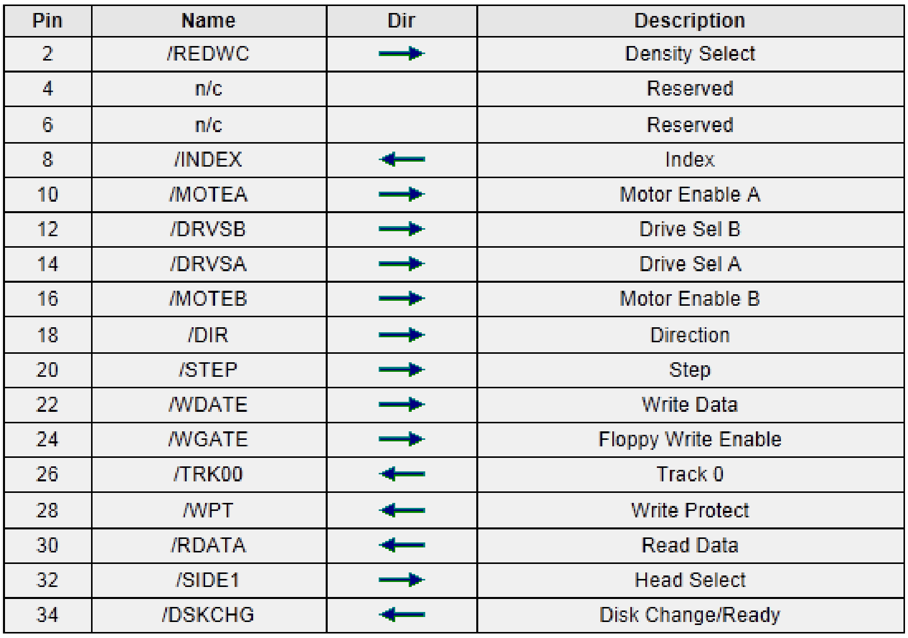


Figure 2.2: Floppy Drive Pinout



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.

2.2 Power

The floppy drive takes a mini Molex cable as seen in Figure 2.4. 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.

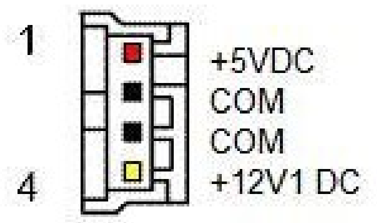


Figure 2.4: Floppy Drive Power Pinout

The power consumption was measured by using jumper cables to connect the floppy drives to a power supply, through an ampmeter. Pin 12 is grounded, to enable drive B. When the drive is not running, the ampmeter 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

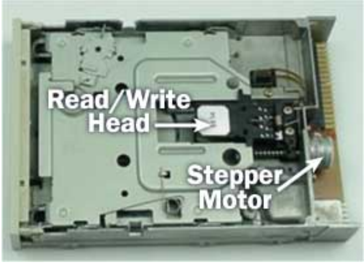


Figure 2.5: Floppy Drive With Top Off

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 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 swepted from 1 Hz up until the read/write head is no longer moving. The drive was able to handle up to 400 Hz, but afterwords, 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.

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.

3.2 Arduino Wiring

3.3 Mount

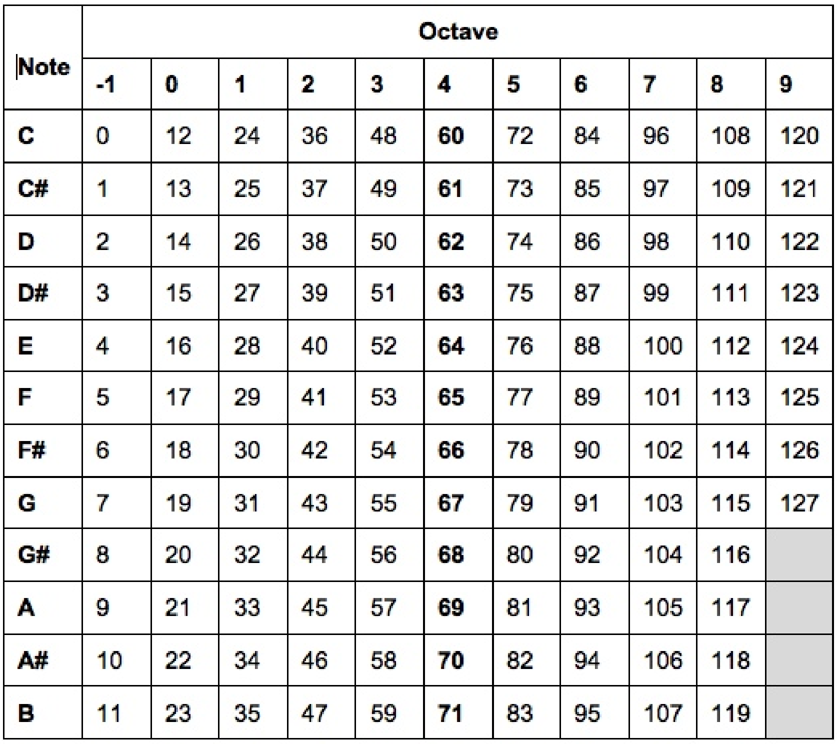
Chapter 4

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 send data to several instruments.

4.1 MIDI Notes

The MIDI interface has notes that are mapped to specific piano keys at their respected frequencies.



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 Messages

Table 4.2: 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

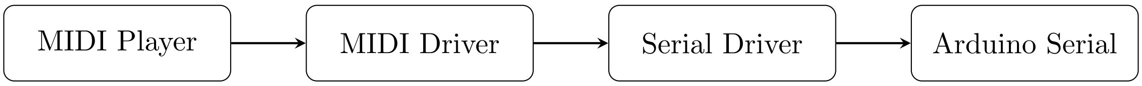
Chapter 5

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.



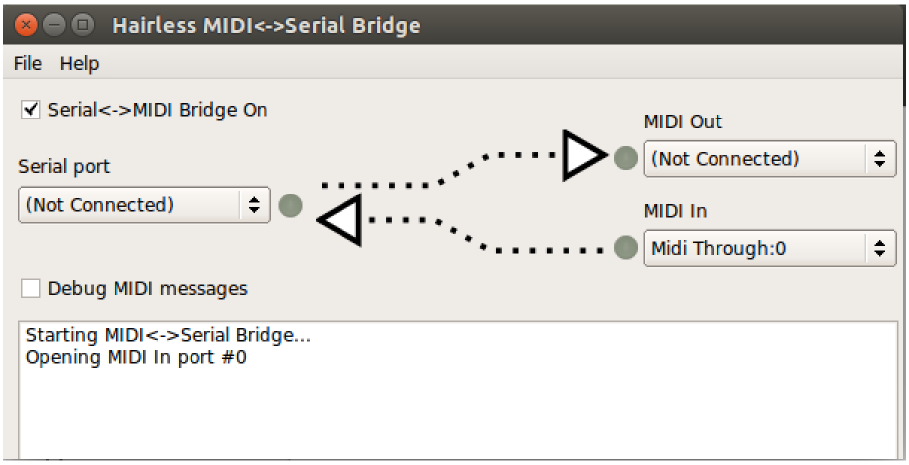
5.1.1 MIDI Player

The MIDI player takes in a MIDI file and play the music.

5.1.2 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 `http://projectgus.github.io/hairless-midiserial/`.

This project has platforms for all three operating systems and has a GUI that is easy to interface with as evident in 5.1. The Serial¡-¿MIDI Bridge On turns the driver on or off, being helpful with programming 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 should be located.



```
Starting MIDI<->Serial Bridge...
Opening MIDI In port #0
```

Figure 5.1: Hairless MIDISerial

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.

```

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

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

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

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

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


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

void setup()  {
```

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

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.

```
    periodCounter[i] = 0;
    notePeriod[i] = 0;

    headPos[i] = 0;

  }

  //Midi note setup found http://subsynth.sourceforge.net/midinote2freq.html
  double midi[127];
  int a = 440; // a is 440 hz...
  for (i = 0; i < 128; ++i)
  {
    midi[i] = a*pow(2, ((double)(i-69)/12));
  }
```

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: `https://en.wikipedia.org/wiki/MIDI_tuning_standard`.

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

  //Pin Setup
  for(i = 0; i < NUMDRIVES; i++){
```

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

```
    pinMode(dirPins[i], OUTPUT);
  }

  //Drive Reset
```

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

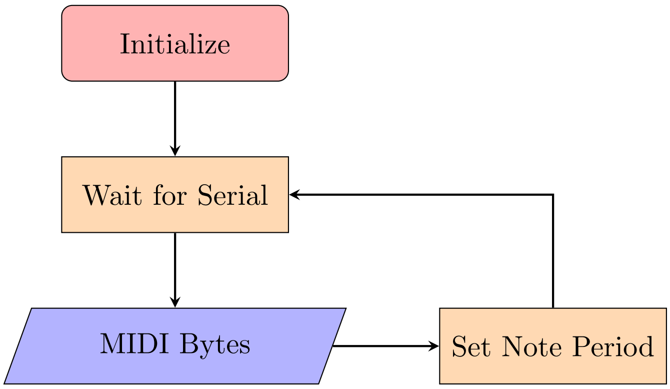
```
    for(j = 0; j < NUMDRIVES; j++){
      digitalWrite(dirPins[j], driveDir[j]);
      digitalWrite(stepPins[j], 0);
      digitalWrite(stepPins[j], 1);
    }

    delay(50);
  }

  //Set Drive Pins to forward direction
  for(i = 0; i < NUMDRIVES; i++){
    driveDir[i] = 0;
    digitalWrite(dirPins[i], driveDir[i]);
  }

  delay(1000);

  Timer1.initialize(RESOLUTION); //1200 microseconds * 1 = 800 Hz, but 400 Hz output.
  Timer1.attachInterrupt(count);

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

}
```

Finally, timer1 is initialized with the appropriate service function.

```
  if(Serial.available() == 3){
    midiStatus = Serial.read(); //MIDI Status
```

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.

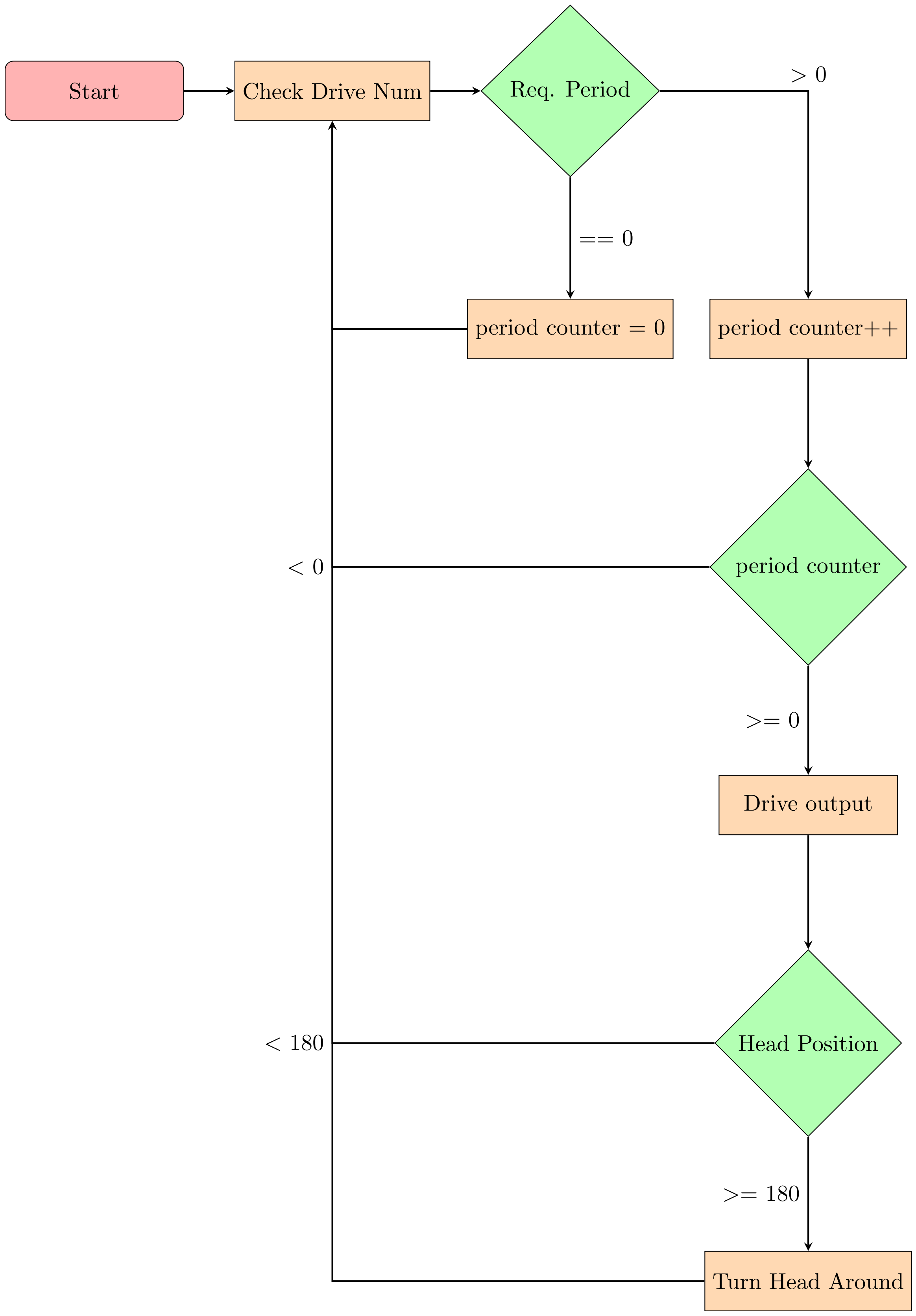


This loop is represented in the loop function of the arduino. This code is continously waiting for serial data, and then it is parsed.

```

    if(midiChannel < NUMDRIVES){
      if(midiCommand == 0x90){
        notePeriod[midiChannel] = noteLUT[midiNote];
      }
      else if(midiCommand == 0x80 || (midiCommand == 0xB0 && midiNote == 120)){
        notePeriod[midiChannel] = 0;
      }
    }
  }
}

void count(){
  int i;
  //For each drive
  for(i = 0; i < NUMDRIVES; i++){
    //If the desired drive is suppose to be ticking
    if(notePeriod[i] > 0){
      //tick the drive
      periodCounter[i]++;
```

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.


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

6.6 Floppy Drive Driver

6.7 SD Card(?)

Chapter 7

Bill of Materials

Table 7.1: MIDI Message

BOM			
ITEM	QUANTITY	COST	SOURCE
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

Appendix A

A.1 Full Arduino Code

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

#include <TimerOne.h>
#include <MIDI.h>

#define NUMDRIVES 8 //Number of Drives being used

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

volatile int stepPins[NUMDRIVES] = {3, 5, 7, 9, A5, A3, A1}; //ODD PINS
volatile int dirPins[NUMDRIVES]  = {2, 4, 6, 8, A4, A2, A0} ; //EVEN PINS

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

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

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

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

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

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


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

void setup()  {

  //Init parameters
  int i,j;
  for(i = 0; i < NUMDRIVES; i++){
    driveDir[i] = 1; //Set initially at 1 to reset all drives
    driveState[i] = 0;

    periodCounter[i] = 0;
    notePeriod[i] = 0;

    headPos[i] = 0;

  }

  //Midi note setup found http://subsynth.sourceforge.net/midinote2freq.html
  double midi[127];
  int a = 440; // a is 440 hz...
  for (i = 0; i < 128; ++i)
  {
    midi[i] = a*pow(2, ((double)(i-69)/12));
  }

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

  //Pin Setup
  for(i = 0; i < NUMDRIVES; i++){
    pinMode(stepPins[i], OUTPUT);
    pinMode(dirPins[i], OUTPUT);
  }

  //Drive Reset
  for(i = 0; i < 80; i++){
    for(j = 0; j < NUMDRIVES; j++){
      digitalWrite(dirPins[j], driveDir[j]);
      digitalWrite(stepPins[j], 0);
      digitalWrite(stepPins[j], 1);
    }

    delay(50);
  }

  //Set Drive Pins to forward direction
  for(i = 0; i < NUMDRIVES; i++){
    driveDir[i] = 0;
    digitalWrite(dirPins[i], driveDir[i]);
  }

  delay(1000);

  Timer1.initialize(RESOLUTION); //1200 microseconds * 1 = 800 Hz, but 400 Hz output.
  Timer1.attachInterrupt(count);

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

}

void loop()  {

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

    midiChannel = midiStatus & B00001111;
    midiCommand = midiStatus & B11110000;

    if(midiChannel < NUMDRIVES){
      if(midiCommand == 0x90){
        notePeriod[midiChannel] = noteLUT[midiNote];
      }
      else if(midiCommand == 0x80 || (midiCommand == 0xB0 && midiNote == 120)){
        notePeriod[midiChannel] = 0;
      }
    }
  }
}

void count(){
  int i;
  //For each drive
  for(i = 0; i < NUMDRIVES; i++){
    //If the desired drive is suppose to be ticking
    if(notePeriod[i] > 0){
      //tick the drive
      periodCounter[i]++;

      //If the drive has reached the desired period
      if(periodCounter[i] ≥ notePeriod[i]){

        //Flip the drive
        driveState[i] ^= 1;
        digitalWrite(stepPins[i], driveState[i]);

        //reset the counter
        periodCounter[i] = 0; //IT WAS == AND I COULDN'T FIGURE OUT WHY IT ...
            WASN'T WORKING

        headPos[i]++;
        //If the drive is at the maximum step, reset its direction
        if(headPos[i] ≥ MAXSTEPS){
          headPos[i] = 0;
          driveDir[i] ^= 1;
          digitalWrite(dirPins[i], driveDir[i]);
        }


      }
    }
    else{
      //Set Counter to 0
      periodCounter[i] = 0;

    }
  }

}
```