Project Lab 2

Final Report

Dirk Thieme
R11636727
Texas Tech University

December 2022

Abstract

This paper describes the final state of the Automated Recorder Project. While the requirements of the project will be discussed more in further sections, the general goal is to automate the playing of a recorder using a Raspberry Pi, which will be able to play any song as given to it through a flash drive. The device can also be controlled using an electric piano, where one can play the recorder using the keys. This project is being developed by Dirk Thieme, Mason Hadley, Jake Kiedasch, and Mariano Arce.

Table of Contents

List of Figures

1 Introduction

The original premise of the Automated Recorder Project is to play a grade school recorder using a Raspberry Pi. The recorder should be able to play eight notes, and would be able to play a select number of songs chosen by the instructor of the course and a local area elementary school teacher. The recorder apparatus should be able to play in front of an elementary school class.

While the premise of the project was enough of a challenge, this project aims to go above and beyond the original scope by allowing the recorder to play all 27 notes within its capability and by allowing the user to dynamically chose the desired song to play with an MP3 file. To allow the recorder to play all notes, certain holes must only be covered halfway, a technique known as leaking and pinching [1]. To be able to play any MP3 file provided, the software will be able to process the song on demand and convert the sound wave into a collection of frequencies which correspond to notes. These notes will be automatically converted into notes that the recorder can play. The recorder will also be able to use an electric piano as a human interface to play songs manually.

The brain of the apparatus is the Raspberry Pi 4B (furthermore Pi), a single-board computer which executes the code and controls the hardware. The Pi is powered using a quad core 64-bit ARM-Cortex A72 running at 1.5GHz [2] which brings enough power to the table in order to complete the challenging digital signal processing required. The GPIO pins are also utilized to be able to control the hardware which actuates the recorder.

The recorder is physically controlled through the use of six solenoids, three servo motors, and three compounded fans. Five of the solenoids cover and uncover the main holes, and

three of the servos perform the pinching of the holes which require multiple positions. The one other solenoid opens and closes the main labium, which stops the recorder from playing to play rests. The solenoids require 12V power, while the servos and Pi require 5V power. The Pi only outputs 5V at 16mA on the GPIO pins [2], therefore a circuit has been designed in order to be able to actuate the solenoids while not burning out the GPIO pins. To power the entire setup, a 12V, 150W power supply will be able to accept AC wall power, which it can then send to a custom made power distribution PCB, which incorporates a 12V-5V buck converter and the various high power outputs, along with the solenoid circuits. The piano used to play the device is the AKAI MPK Mini MK3, a MIDI controller built-in to a 25 key keyboard, which allows the user to play songs. The piano can sweep between 10 octaves, however this device only uses about 2 [3]. The integrated USB port is plugged into the Pi and is integrated into the Python code so it can use the recorder.

While the structure of the software is rather simple, the implementation is exceedingly complex, and requires immense amounts of research and assistance to complete. The software is controlled by the user using a GUI written using the TKinter library, where there are three tabs. The first tab is a list of songs which are built-in to the device, and one clicks on one and it plays. The second tab allows the user to ingest an MP3 file, then convert it into a WAV file. The WAV file bytestream is parsed into its proper values, then its time-domain signal is broken up into bins which are individually processed using a Fast Fourier Transform into their frequency-domain counterpart. The transform is not performed on the Pi, as the CPU would take to long, so it is sent over the Internet to a server which uses its actual full-size CPU to calculate the answer.The magnitudes of each bin are then calculated, which reveal the peak frequency of the bin, corresponding to the note at that time period. These notes

are then converted into their recorder playable counterparts, which will then be played for the correct amount of time. The third tab allows the user to play the piano.

2 Body

The final state of this project is comprised of the the Software and the Hardware. The Softwre is a Python codebase which consists of the GUI and the three methods of interaction, the Custom Songs, the Hardcoded Songs, and the Piano Input. The Hardware consists of the recorder itself, the control devices, the air production, the piano input, and the Pi which runs the whole thing.

2.1 Software

2.1.1 Flow of Control Chart

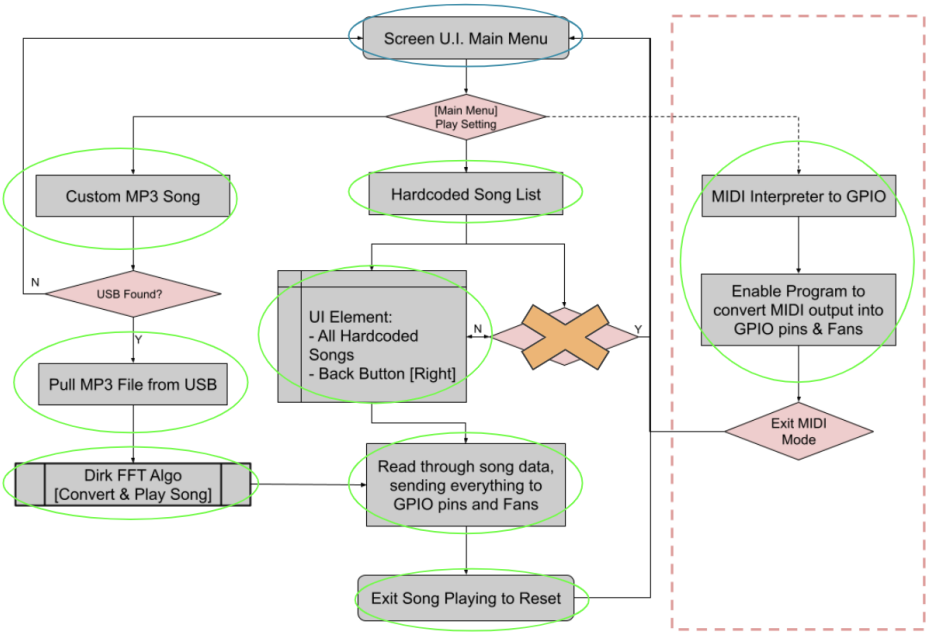

Figure I: Main Software Flowchart

As shown in Figure I, this flowchart is the path that the software takes to complete its tasks. The screen is a basic 7-inch touchscreen, and the code just automatically opens up upon startup of the Pi.

2.1.2 GUI Code

As the touchscreen needs to be used to keep the interface compact and efficient, the GUI must be easy for fingers to touch and control, otherwise the interface would be unsatisfactory to use. The code itself is broken up into a separate class which makes it easy to edit and update accordingly. TKinter is a GUI library which breaks up the interface into different parts, the root, which contains the main window, the notebook, which is what contains the individual frames. The frames contain each interactive element, like a folder contains a file. The root is configured to popup in full screen and to accept the notebook. The notebook is configured to have three tabs which each have a single frame containing the three main aspects of the code. This is shown in Figure II.

```
# extra constants
self.filename = "None chosen ... "
self.noteObject = Notes()
self.exitFlag = False
self.midiin = rtmidi.RtMidiIn()

# window setup
self.title("Automatic Recorder")
self.attributes("-fullscreen", True)

# configure rows and columns for auto-scaling
self.grid_rowconfigure(0, weight=1)
self.grid_columnconfigure(0, weight=1)

# setup notebook
self.book = ttk.Notebook(self)
self.book.grid(row=0, column=0, sticky="nsew")

# make frames
self.hardFrame = ttk.Frame(self.book)
self.hardFrame.pack(fill='both', expand=True)
self.realFrame = ttk.Frame(self.book)
self.realFrame.pack(fill='both', expand=True)
self.pianoFrame = ttk.Frame(self.book)
self.pianoFrame.pack(fill='both', expand=True)

# hard frame config
ttk.Label(self.hardFrame, text="Hardcoded Song List", font=("", 20)).place(relx=0.5, rely=0.15, anchor="center")
songListItems = tk.Variable(value=list([songList[i].name for i in range(len(songList))]))
songListBox = tk.Listbox(self.hardFrame, listvariable=songListItems, height=len(songList), width=30, font=("", 20), selectmode=tk.SIN
songListBox.place(relx=0.5, rely=0.45, anchor='center')
tk.Button(self.hardFrame, text="Click to Play Selected Song",
                        width=75,
                        height=4,
                        command=lambda: songList[songListBox.curselection()[0]].SongPlayback(self.noteObject)).place(relx=0.5, rely=
```

Figure II: GUI Setup Code

2.1.3 Notes and Fan Control

A class is setup which is used to control the GPIO pin outputs, including the PWM signals that control the fan and the servos, and the standard on/off signals which control

the solenoids. At every startup of the code, the GPIO pins are initialized and turned off to make sure the solenoids are inactive when waiting for a song to play. Figure III displays this code. When any song or piano is active, the fan is started for a few seconds before the

```
def GPIOInit(self):
    GPIO.setmode(GPIO.BOARD)
    GPIO.setwarnings(False)

    GPIO.setup(Notes.SOL_1, GPIO.OUT, initial=GPIO.LOW)
    GPIO.setup(Notes.SOL_2, GPIO.OUT, initial=GPIO.LOW)
    GPIO.setup(Notes.SOL_3, GPIO.OUT, initial=GPIO.LOW)
    GPIO.setup(Notes.SOL_4, GPIO.OUT, initial=GPIO.LOW)
    GPIO.setup(Notes.SOL_5, GPIO.OUT, initial=GPIO.LOW)
    GPIO.setup(Notes.SOL_6, GPIO.OUT, initial=GPIO.LOW)

    self.fan = HardwarePWM(0, 500)
    self.servo0 = Servo(Notes.SERVO_0)
    self.servo6 = Servo(Notes.SERVO_6)
    self.servo7 = Servo(Notes.SERVO_7)



def GPIOClean(self):
    self.fan.stop()
    self.servo0.detach()
    GPIO.output(Notes.SOL_1, Notes.SOL_OPEN)
    GPIO.output(Notes.SOL_2, Notes.SOL_OPEN)
    GPIO.output(Notes.SOL_3, Notes.SOL_OPEN)
    GPIO.output(Notes.SOL_4, Notes.SOL_OPEN)
    GPIO.output(Notes.SOL_5, Notes.SOL_OPEN)
    GPIO.output(Notes.SOL_6, Notes.SOL_OPEN)
    self.servo6.detach()
    self.servo7.detach()
    print("cleaned")
```

Figure III: GUI Setup Code

actual input begins playing, as the fan needs time to be able to push air at a usable rate. When a song is playing, the notes are played using a function which actuates all necessary solenoids and rotates servos, while altering the speed of the fan as required. This function

is showed in Figure IV.

```
def playNote(self, note, length):
    GPIO.output(Notes.SOL_6, Notes.SOL_CLOSED)
    timer = time.time()
    while time.time() - timer ≤ length:
        if note is None or note == "-":
            GPIO.output(Notes.SOL_6, Notes.SOL_OPEN)
            continue

        self.fan.change_duty_cycle(Notes.PHYS_NOTE_DICT[note][0])
        self.servo0.value = Notes.PHYS_NOTE_DICT[note][1]
        GPIO.output(Notes.SOL_1, Notes.PHYS_NOTE_DICT[note][2])
        GPIO.output(Notes.SOL_2, Notes.PHYS_NOTE_DICT[note][3])
        GPIO.output(Notes.SOL_3, Notes.PHYS_NOTE_DICT[note][4])
        GPIO.output(Notes.SOL_4, Notes.PHYS_NOTE_DICT[note][5])
        GPIO.output(Notes.SOL_5, Notes.PHYS_NOTE_DICT[note][6])
        self.servo6.value = Notes.PHYS_NOTE_DICT[note][7]
        self.servo7.value = Notes.PHYS_NOTE_DICT[note][8]
```

Figure IV: GPIO Playback Code

2.1.4 Hardcoded Song Playback

As the main goal of the project is to be able to play a given list of songs, it was found that it was easier and more reliable to simply hardcode the songs that were needed. This is contained within the HardcodedSong class, shown in Figure V. Figure VI shows an example of how the class input is setup. The name of the song is first, then the tempo, then a list of the notes, then finally the list of timings. To play the song, the method SongPlayback is called on the instance. This loops through each of the notes and plays them for the allotted time, using the note playing class mentioned above.

2.1.5 Custom Song Playback

To facilitate the playing of custom songs, a tab is there which consists of two buttons, a search for file button and a play button. The user first picks the MP3 file to play, then clicks

```
class HardcodeSong:
    def __init__(self, name, tempo, notes, rhythm):
        self.name = name #Name of the Song
        #converting tempo from [Beats per Minute] to [
        self.tempo = math.pow((tempo/60), -1)
        #Lists of Notes (To be Converted to GPIO) & No
        self.notes = notes
        self.rhythm = rhythm

    def SongPlayback(self, noteObj:Notes):
        print("Playing Hardcoded Song: " + self.name)
        noteObj.spoolFan()
        Note = ""
        Value = 0
        for i in range(len(self.notes)):
            # Pop a note and rhythm off the two lists
            Note = self.notes[i]
            Value = (self.rhythm[i])*self.tempo
            # Play/Print True note (Find GPIO Pin Inpu
            # print(Note)
            # Wait = (Multiply Rhythm by tempo: Note L
            noteObj.playNote(Note, Value)

        noteObj.GPIOClean()
```

Figure V: Harcoded Song Class

the play song button to initiate the process. The bulk of the MP3 ingestion is performed by the software **ffmpeg**, originally developed by Fabrice Bellard and Bobby Bingham on December 20, 2000 [4]. Shown in Figure VII, certain command flags must be used upon executing the program in order to ensure proper output [4]. Some of the commands are less important, such as "-sn", which ignores subtitles or "-vn", which ignores any video streams (although this does implicate the software could feasibly process video data). Other commands are a necessity for proper output, such as "-f" and "wav", which sets the output format to a WAV file, and "-acodec" and "pcm_s16le", which sets the output codec correctly.

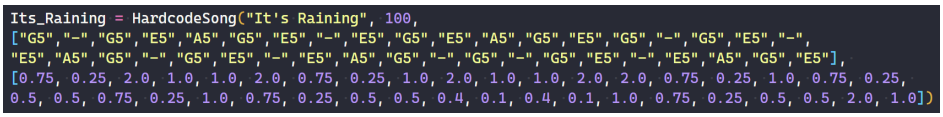

Figure VI: Harcoded Song Example

```
# MAIN PROGRAM
def main():
    # define command to run when decoding mp3
    cmd = [exe,          # ffmpeg executable
           "-i",         # input
           filename,     # name of input file (mp3)
           "-ab",        # set bitrate
           "128k",       # bitrate of input file
           "-acodec",    # set audio codec
           "pcm_s16le",  # using PCM signed 16-bit little-endian
           "-map",       # map audio streams
           "0:a",        # from file 0, take only audio
           "-sn",        # ignore any subtitle
           "-vn",        # ignore any video
           "-ac",        # set channels
           "1",          # mono
           "-y",         # override output
           "-f",         # set output format
           "wav",        # wav format
           "pipe:1"]     # pipe output to stdout and stderr

    # subprocess command, piping outputs to this program
    output = subprocess.run(cmd, stdout=subprocess.PIPE,
                            stderr=subprocess.PIPE, bufsize=10**8)
```

Figure VII: ffmpeg Command and Output Implementation

The most important flag there is "pipe:1" which outputs the WAV bytestream to the stdout, instead of dumping the output to a file. All of the output is contained in the variable conveniently named "output." Output consists of two properties, stderr and stdout. As stated above, stdout contains the bytestream of the output, but for debugging purposes stderr is useful as it contains the output of the command.

2.1.6 WAV Parsing and Processing

Now that a WAV "file" has been obtained, work must be done in order to ascertain the correct values. The codec which was used to process the song is known as PCM, and the actual values that were output were unsigned 16-bit little-endian chunks. If one simply tried to read the bytes as they were, nothing but gibberish would be produced. This is because the bytes are little-endian, meaning the lower 8-bits are placed ahead of the upper 8-bits, effectively reversing the number in memory. As an example, take the decimal number $2,500$. Its representation in a 16-bit hexadecimal is $0x09C4$. Storing this value into contiguous memory locations would result in the following:

$$0xC4 \rightarrow 0x09 \rightarrow ...$$

Attempting to get this value by only reading from left to right would return the number $0xC409$, which is the decimal number $50,185$, the incorrect value.

To combat this, the Python package struct is used, as the programmer can tell the struct unpacking object what format to retrieve the bytestream as. This is done using format strings, in this case the string "<L". The < symbol tells the object the bytes will be little-endian, and the L tells the object to pack them into a 4-byte number [5]. This initial reading of the file allows the program to find the necessary sections of the file to read from.

WAV files consist of a standardized structure [6]. They begin with a 44-76 byte header, depending on the WAV file creator. The program parses this header to either find the "fmt" section or the "data" section. The "fmt" section contains useful file information, such as number of channels, sample rate, and codec type. The "data" section contains, quite simply,

all the actual song data. The program parses the "fmt" section for the information it needs to later process the "data" section. This is shown in below in Figure VIII. After this is

```
def _unpackWAVE(rawSignal):
    # convert bytes to file object
    wavefile = io.BytesIO(rawSignal)

    # ignore first 4 bytes of header (RIFF header, will awlways be)
    _ = wavefile.read(4)
    lengthOfFile = struct.unpack(Signal.FMT_4B_LE, wavefile.read(4))[0]  # get length
    # ignore next 4 bytes (denotes if WAVE file, will always be)
    _ = wavefile.read(4)

    # read file until end of file, plus 8 because we read the first 8 bytes already
    while wavefile.tell() < lengthOfFile + 8:
        # find parent of the current block
        parent = wavefile.read(4)
        # find length of the remainder of block
        try:
            currentLength = struct.unpack(Signal.FMT_4B_LE, wavefile.read(4))[0]
        except struct.error:
            # if struct no longer able to read, most likely out of bounds
            break

        # decide what to do with the current block
        if parent == b'fmt ':
            fmtData = wavefile.read(currentLength)
            # parse fmtData
            fmtTag, numChannels, framerate, avgBytesPerSecond, blockAlignVal, bitsPer
        elif parent == b'data':
            # grab raw signal data
            readSignal = wavefile.read(currentLength)
            parsedSignal = Signal._unpackSignal(readSignal, blockAlignVal)
        else:
            # if uneedeed info, skip current block
            wavefile.seek(currentLength, 1)

    return parsedSignal
```

Figure VIII: WAV Processing Code

complete, the variable "parsedSignal" is returned to the main program, which contains the actual waveform, and is then ready for further processing.

2.1.7 The Fast Fourier Transform

While the program now has a long list of numbers which correspond to the waveform, shown below, the waveform itself is not a very useful signal for note processing. These numbers are just the amplitude of the sound at that point in time, but to find the note at that point in time, the Fourier transform of the signal must be taken. The algorithm that has been implemented for live music processing is the Cooley-Tukey Fast Fourier Transform. This algorithm was popularized in 1965 by James Cooley of IBM and John Tukey of Princeton, although the original implementation was found to have been discovered in 1805 by Carl

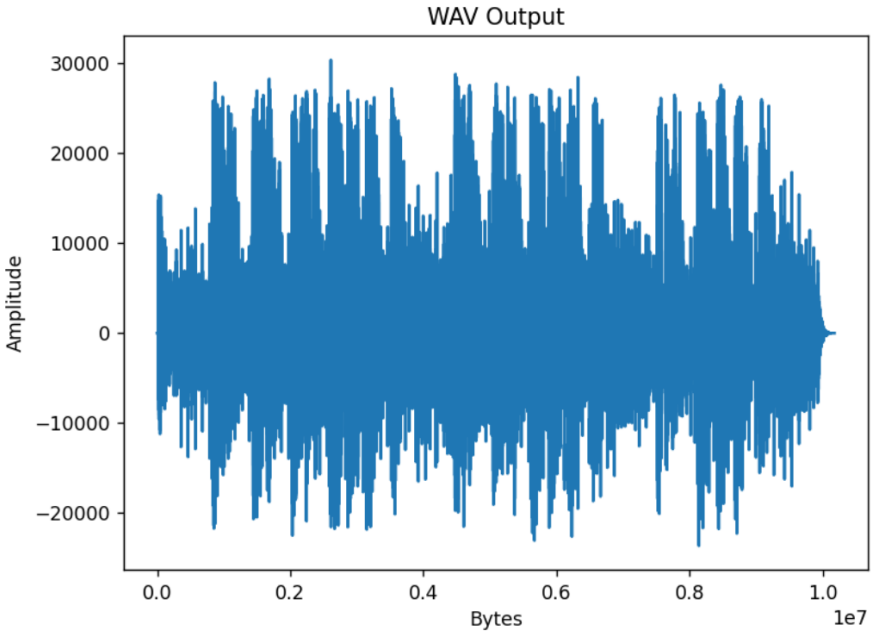

Figure IX: An Example Waveform of a Song

Friedrich Gauss [7]. The main premise is to recursively divide the Fourier transform into smaller and smaller chunks, which are then added back together at the end to produce the output [8]. The standard Discrete Fourier Transform is defined by the equation [7]:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi kn/N} \quad k = 0 \,..\, N-1 \tag{1}$$

which is an $O(n^2)$ operation. The radix-2 algorithm brings this down to only $O(n \log n)$ through the recursion and division of the problem, leading to massive time and memory savings down the line. In order to save even more time for the sake of presentation, the original bytestream is sent over the network to a server which will then perform the FFT algorithm on the far more powerful Intel Xeon processors. The code is placed in a Docker container, which is a virtual machine inside of the server that can be enabled on input of the bytestream.

A prerequisite of this operation is that the provided signal must be a power of 2, which is where the time savings come from [7]. In order to this, the length of the signal is checked using the code in Figure X, which compares the bits of the number and the number minus one. If the number is a power of 2, the result of ANDing them together would be 0, otherwise it would be any other number. Because the sample rate of the songs is 44.1kHz, and the

```
@staticmethod
def _isPowerOfTwo(n):
    return (n ≠ 0) and (n & (n - 1) == 0)
```

Figure X: Function to Calculate if Number is a Power of 2

windows of time are 100ms long, the number is not a power of 2, therefore the next highest power of 2 must be found and the signal must be extended to the new number.

To find the next power of 2, a function which forces all the bits of the number to roll over until they are all 1's, then adds one to the number is used to calculate the next number to pad to. This is shown in Figure XI. Once the signal is appropriately padded, the actual

```
@staticmethod
def _calcNextPowerOf2(length):
    # decrement by one in case of already power of 2
    length -= 1

    # right shift all bits to force number to roll over
    length |= length >> 1
    length |= length >> 2
    length |= length >> 4
    length |= length >> 8
    length |= length >> 16

    # return rolled over number
    return length + 1
```

Figure XI: Function to Find the Next Highest Power of Two

math can begin. The main algorithm consists of five steps:

1. Find the length of the current input list. If the length is one, return the list as is, otherwise continue.

2. Using the length of the list, calculate ω using the formula $\omega = e^{j2\pi/N}$ where N is the length.

3. Break the current list into its odd and even elements.

4. Pass the two new lists recursively into function.

5. When the lists return, recombine the odd and even lists into a single list by raising ω to the current index of the loop and doing some other math.

After these steps are completed the Fast Fourier Transform of the current list is complete.

2.1.8 FFT Post-Processing

Once the waveform is converted from a time domain signal into a frequency domain signal, further work must be done to transform the current values into usable frequencies. To do this, the highest magnitudes of the bins of values must be calculated. First, the second half of the signal is discarded, as the second half if just the complex conjugate of the first half [7]. Then, the absolute value at each point is taken, as this is a real signal, the imaginary and negative components are irrelevant. Finally, the actual magnitude is calculated. To do this, loop through the remainder of the list, and at each even and odd point treat the even as the real component and the odd as the imaginary, then combine them using the following formula:

$$Magnitude = \sqrt{Re^2 + Im^2} \tag{2}$$

These magnitudes are then inserted into their own list, where the largest value corresponds to the highest peak in the frequency spectrum, which is almost always the frequency at that point in time. An example of the peaks is shown below in Figure XII. To find the actual

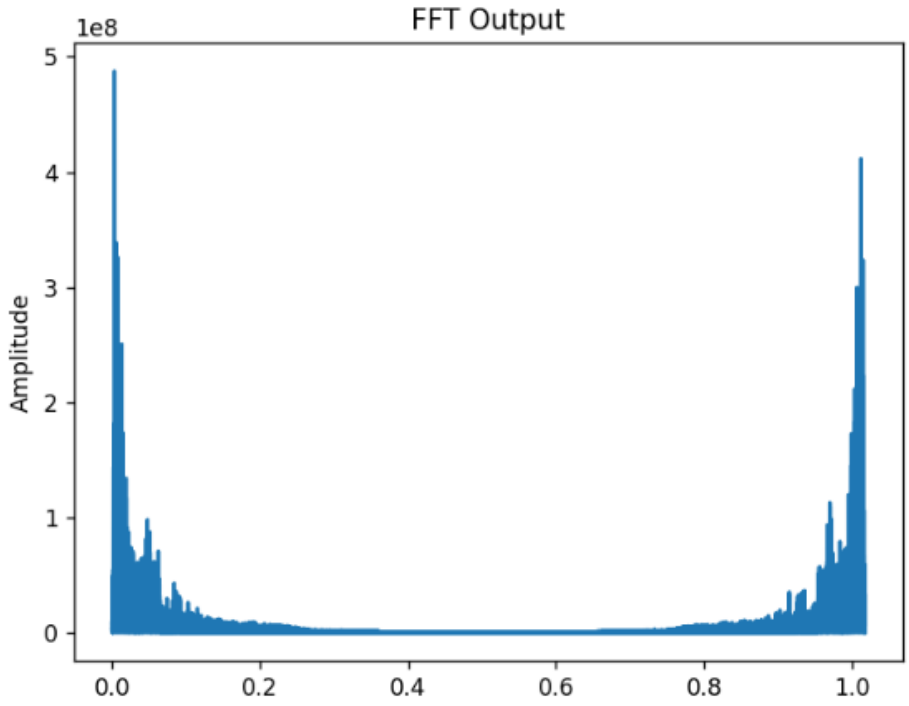

Figure XII: Example of the FFT Output

frequency associated with the peak, the equation as shown below can be used:

$$Frequency = \frac{(index\ of\ largest\ peak) * (sample\ rate)}{(length\ of\ bin)} \tag{3}$$

To actually play a song, the frequencies are placed into a list, where repeated frequencies are counted and divided by 10 to calculate the time required to play that note. To find the note, a ± 5 percent range is calculated around the frequency, which is used as a key for a dictionary with the frequencies to notes built in.

Once those notes are found, those notes are rounded up and down into the actual playable

notes on the recorder. This is done by comparing the note to a list of notes which the reorder can physically play. The note is then played for the length of time calculated before using the note playing method in the note class.

2.1.9 Piano Input

To play the piano, another tab is included which consists of two buttons, a play piano button and a stop playing button. The stop playing button is required as the function, shown in Figure XIII, is permanently stuck in a loop which looks for input until a certain flag is marked true. To make sure the program is not locked up upon hitting the start playing button, the initial piano playing process is actually being run in a seperate thread, which is done by using the Python library threading, a library that leverages the CPU's multiple cores for concurrent parallel execution. The exitLoop function is also executed in a separate

```
def playPiano(self):
    self.midiin.openPort(1)
    self.noteObject.spoolFan()

    while not self.exitFlag:
        m = self.midiin.getMessage(250) # some timeout in ms
        if m:
            self.noteObject.playPianoNote(m)

    self.exitFlag = False
    self.noteObject.GPIOClean()


def exitLoop(self):
        self.exitFlag = True
```

Figure XIII: Piano Playing and Exit Functions

thread as to not lock up the main thread which keeps the main thread free for other program usage.

2.2 Hardware

2.2.1 The Recorder

As the main goal of the project is to play a recorder automatically, the most important facet of the hardware is the recorder itself. The YRS-24B is the instrument used. This recorder is fairly standard, and is tuned in C. Figure XIV depicts the recorder. This specific

Figure XIV: The Yamaha YRS-24B Recorder

recorder can play notes from C5 to D7, which is a range of 27 total notes. Playing is accomplished by blowing into the mouthpiece and covering and uncovering the holes as required by each note and octave. This recorder also has a rotating end piece, which makes it easier to articulate with rigid mechanical components, and is used to tune the instrument.

2.2.2 Airflow

The recorder is a woodwind instrument [1], and being so it requires airflow to play. The proper airflow requirement is around 0.1 to 1 liter per second [9], therefore while the speed of the air is less important, generating the correct amount airflow is necessary for good sound. To produce a balance between too much and too little airflow, a single blower fans are mounted in series in order to control the flow. This is shown in Figure XV. The fan is

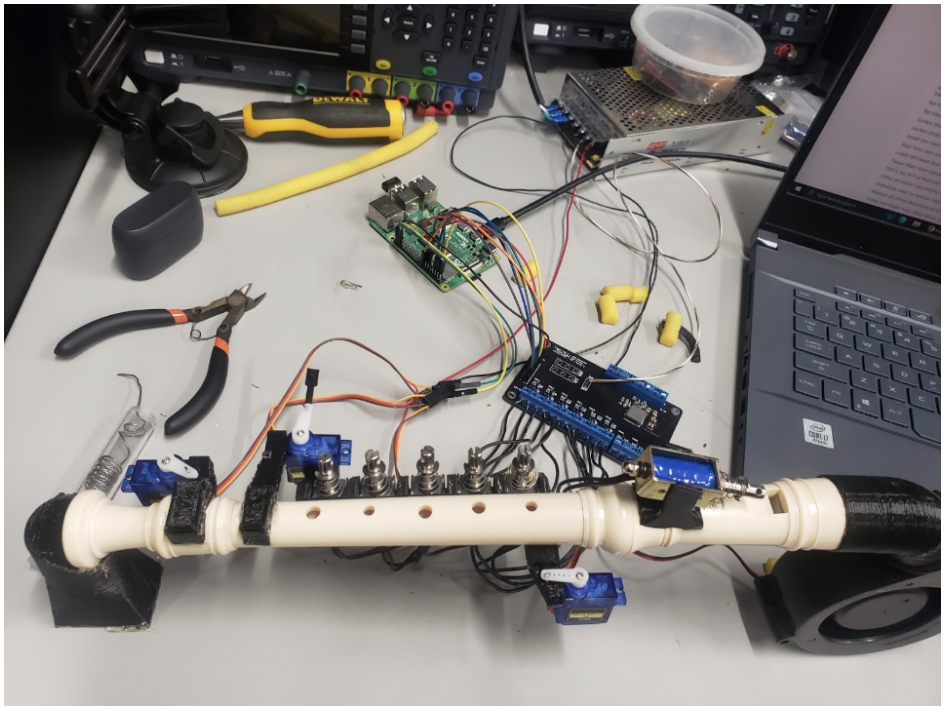

Figure XV: Recorder Fan Setup

a Sparkfun 12V blower, a DC fan that can output 16CFM at 3500 RPM[10]. This blower motor provides the perfect amount of airflow and is simple to increase or decrease at will. Controlling the air is done through a hardware PWM port with the Raspberry Pi.

2.2.3 Solenoids and Servos

While the correct amount of air now properly flows through the recorder, only a single note would play if the holes are not used. To do this, a series of five solenoids and three servos is used. The five solenoids are meant to actuate the holes facing upwards, where a fully open or fully closed state is acceptable. For the holes that require variably closed and open states, the three servos are positioned under the recorder to close and open them as needed. A 3D model of the system is shown below in Figure XVI.

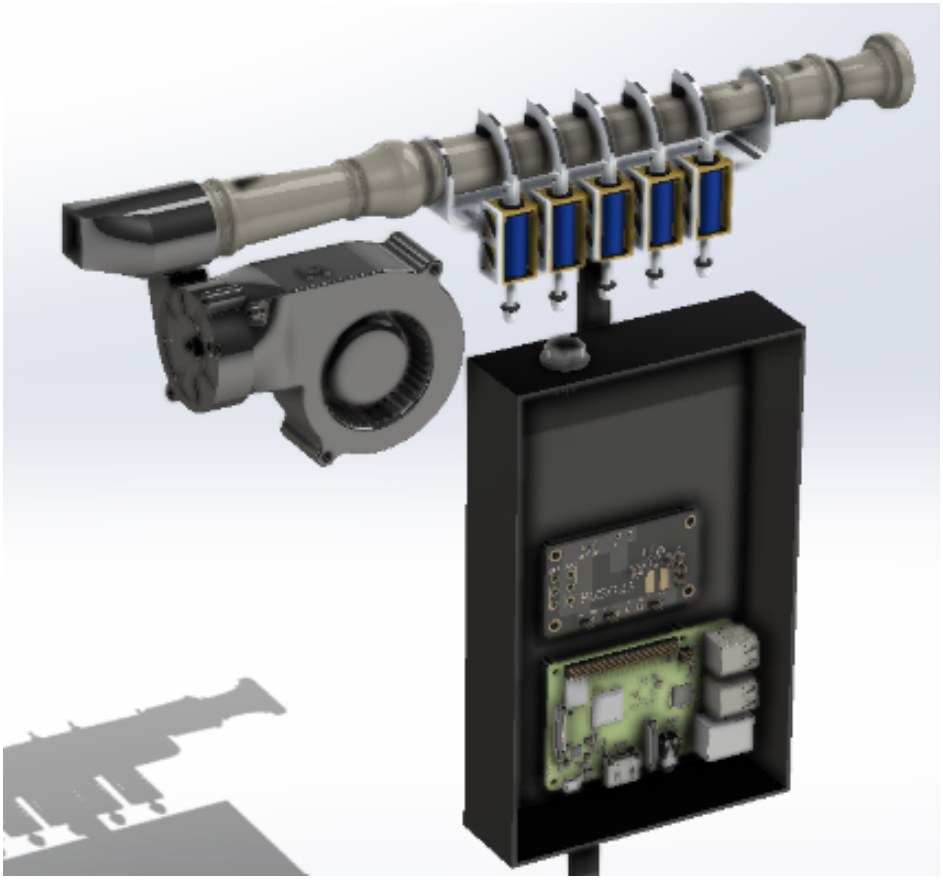

Figure XVI: 3D Recorder Model

The solenoids are rated for 12V, 1A at around 500g of actuation force. In reality, that is closer to around 600mA, but everything designed is rated for 1A. As stated previously,

the Pi can only output 5V at 16mA, so a circuit has been designed in order to drive them. The power supply, which will be discussed later, provides both 12V and 5V, so a MOSFET is activated with the 5V GPIO signal, which allows 12V power to run the solenoids. The solenoids are inductive loads, so when they are deactivated, a spike of current flows back into the circuit. To protect the main circuit, a flyback diode is used to stop the current. This is shown in Figure XVII. For the flyback diode, a Schottky diode is used for its fast

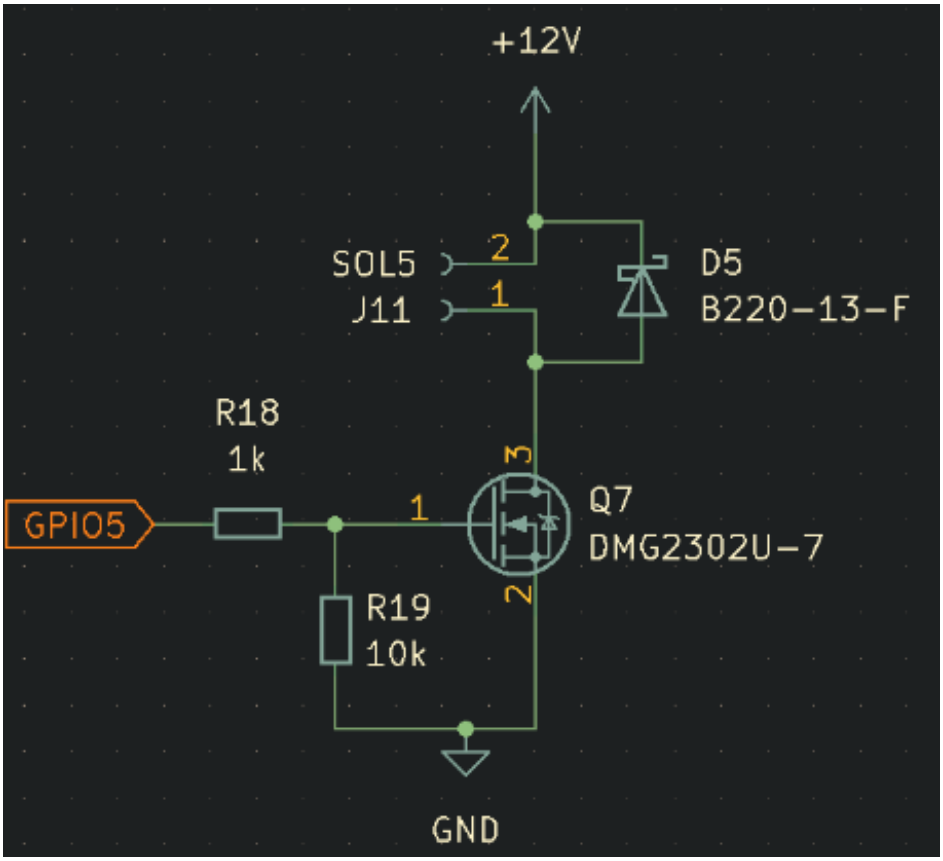

Figure XVII: Solenoid Circuit Diagram

switching abilities and high reverse breakdown voltage. When the GPIO pin is high, 12V flows through the circuit, which then can enter the solenoid. The servos are connected to dedicated 5V outputs, and the positions are controlled with the dedicated PWM connection.

2.2.4 Power Supply and Driver PCB

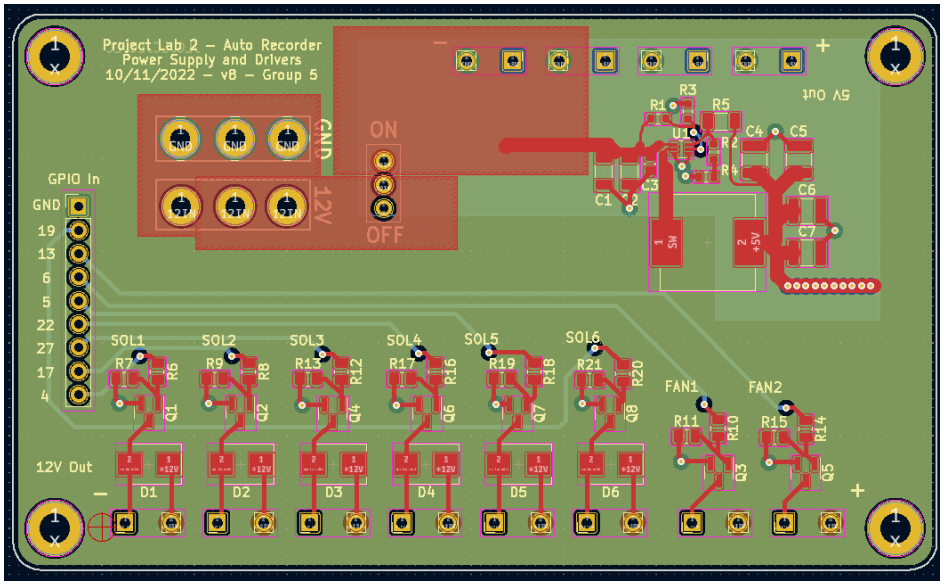

Figure XVIII: Main PCB

The PCB has been designed to provide power and outputs for all connected hardware. It is capable of providing a maximum of 12V at 7A and 5V at 6A. Input power is provided from a 12V, 15A AC to DC power supply, and is connected to the board using a GX-16 6 pin connector. A switch connects this input to the rest of the board, which then sends the power to the rest of the board.

The 5V is generated with a TPS566247DRLR, a 5V, 6A switch mode power supply. This voltage is output to four connectors, three for the servos and one for the Pi. The rest of the raw 12V power is routed to the six solenoid drivers and two fan drivers, which is the same circuit as the solenoid driver but lacking a flyback diode. The board is a four layer design, with a stackup consisting of signal-ground-12V-signal, measuring 1.6mm thick. The board is 100x60mm, with four M3 bolt holes for mounting to the apparatus.

2.2.5 Computational Power

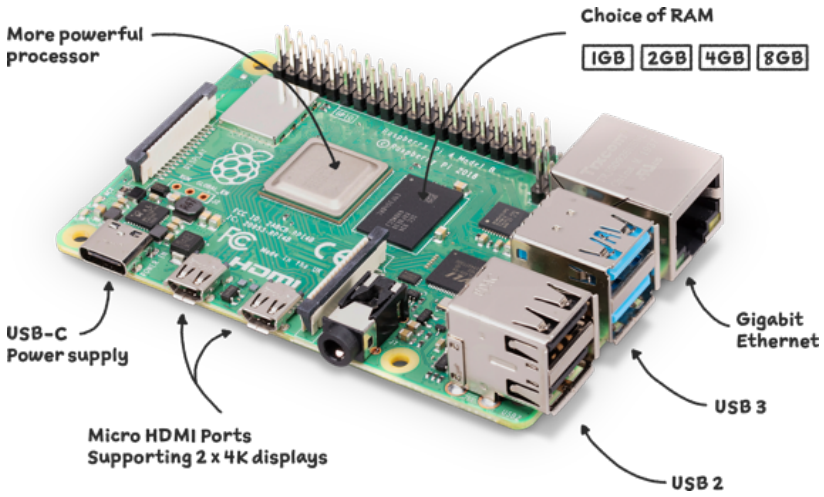

Figure XIX: The Raspberry Pi Model 4B

The Pi used is a 4GB model with a quad core ARM processor. To aid the speed of the FFT execution for songs, the process will be sent to another server over the internet and then multithreaded using Python's builtin threading library. GPIO pins 4, 17, 27, 22, 5, 6, 13, 19 are connected to the various solenoid and servo outputs. To provide the least overhead possible, instead of installing a full-fledged operating system, a headless version of Raspbian is used in combination with SSH for access.

3 Conclusion

As of writing this paper, the project is complete and works well. The speed of the played songs is a little bit slower than the typical way one might sing the songs, that is not an issue of the device itself, its just the rhythm of the song as was programmed. The final touch before presenting to the elementary school we will be at is putting all the parts inside of a case to neatly tie everything up. The project went smoothly, and although there were a few

changes in between the midterm paper and now, there were no significant pivots to other technologies.

4 Appendix

4.1 GANTT Chart and Budget

Project Lab 1	Running Total			Total Estimate			Start Date:	9/2/22		
Direct Labor:							Today:	11/29/2022		
Catagory / Individual	*Rate/Hr*	*Hrs*	*Indv. Total*	*Rate/Hr*	*Hrs*	*Indv. Total*				
Mason	$18.00	119	$2,142.00	$18.00	130	$2,340.00	End Date:	12/3/22		
Dirk	$18.00	127	$2,286.00	$18.00	130	$2,340.00				
Jake	$18.00	121	$2,178.00	$18.00	130	$2,340.00				
Mariano	$18.00	105	$1,890.00	$18.00	130	$2,340.00				
DL Subtotal (DL)		*Subtotal:*	$8,496.00		*Subtotal:*	$9,360.00				
Labor Overhead	*rate:*	100%	$8,496.00	*rate:*	100%	$9,360.00				
Total Direct Labor (TDL)			$16,992.00			$18,720.00				
Contract Labor:										
Lab Assist	$40.00	0	$0.00	$40.00	3	$120.00				
Misc.	$18.00	2	$36.00	$18.00	5	$90.00				
Instructor	$200.00	2	$400.00	$200.00	3	$600.00				
Total Contract Labor (TCL)			$436.00			$810.00				
Direct Material Costs:										
Total DMC (TDM)			$386.13			$500.00				
Equipment Rental Costs:	Value:	Rental Rate:		Value:	Rental Rate:		Date Begin:	Today	End Date:	Total rental Days
Oscilloscope	$1,275.00	0.20%	$224.40	$1,275.00	0.20%	$234.60	9/2/22	11/29/2022	12/3/22	88
Power Supply	$966.00	0.20%	$170.02	$966.00	0.20%	$177.74	9/2/22	11/29/2022	12/3/22	88
Multimeter	$777.00	0.20%	$136.75	$777.00	0.20%	$142.97	9/2/22	11/29/2022	12/3/22	88
Waveform Generator	$1,051.00	0.20%	$184.98	$1,051.00	0.20%	$193.38	9/2/22	11/29/2022	12/3/22	88
							[table rent date]			
Total Rental Costs (TRM)			$716.14			$748.70				
Total TDL+TCL+TDM+TRM			$18,530.27			$20,778.70				
Business Overhead		55%	$10,191.65		55%	$11,428.28		Percentage Budget Expended		
Total Cost:		Current:	$28,721.92		Estimate:	$32,206.98		89.18%		

Figure XX: Current Budget

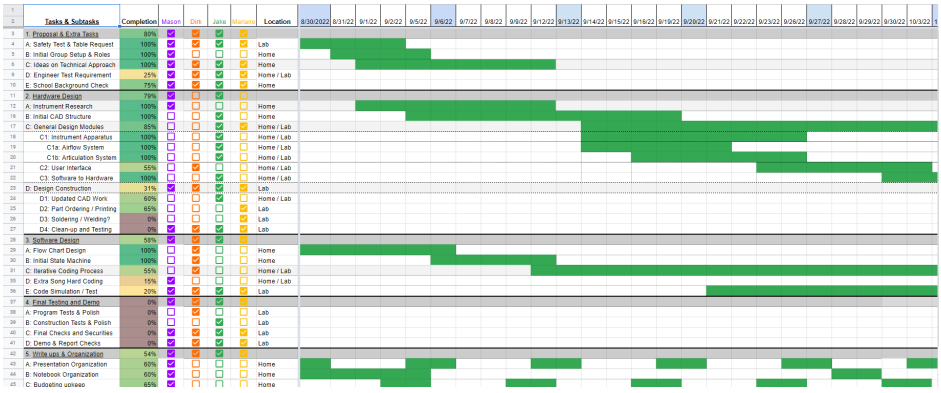

Figure XXI: Current GANTT Chart Progress

As shown in Figures XX and XXI,the budget and GANTT progressed perfectly on schedule and under budget. The final budget total was around 89%, which shows that the total amount was calculated correctly. The GANTT chart is completely satisfied as well.

4.2 Safety and Ethics

Special considerations must be thought of safety wise, as this project is meant to be presented to a classroom of elementary schoolers. Because of this, care has been taken to ensure AC power is never exposed anywhere a student might touch, and that the touchscreen access in general should only be relegated to the teacher of the classroom or the people of the lab group. It is also important to consider volume levels, as overblowing the recorder could cause a loud, sharp noise which could hurt a student. The maximum airflow of the fans has been found, and it is within an acceptable range as to not hurt any hearing. The ethical ramifications stem from the fact that human musicians are already paid very little for their talents, so developing an automated system to play an instrument might lead to less jobs. However, this machine will never be able to be as versatile and talented as a real person playing, so I think it is safe to say this shouldn't hurt any chances.

References

[1] "Recorder (musical instrument)." (Oct. 19, 2022), [Online]. Available: https://en.wikipedia.org/wiki/Recorder_(musical_instrument)#Acoustics.

[2] *Raspberry Pi 4 Model B*, Raspberry Pi (Trading) Ltd., Jun. 2019.

[3] "MPK Mini Play MK3." (Dec. 5, 2022), [Online]. Available: https://www.akaipro.com/mpk-mini-play.

[4] "ffmpeg Documentation." (Oct. 19, 2022), [Online]. Available: https://ffmpeg.org/ffmpeg.html.

[5] "struct — Interpret bytes as packed binary data." (Oct. 19, 2022), [Online]. Available: https://docs.python.org/3/library/struct.html.

[6] "What is a WAV file?" (Oct. 19, 2022), [Online]. Available: https://docs.fileformat.com/audio/wav/.

[7] "Fast Fourier transform." (Oct. 19, 2022), [Online]. Available: https://en.wikipedia.org/wiki/Fast_Fourier_transform.

[8] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," 1965.

[9] J. Wolfe. "Air speed and blowing pressure in woodwind and brass instruments: how important are they?" (Oct. 24, 2022), [Online]. Available: https://www.phys.unsw.edu.au/jw/air-speed.html.

[10] "SparkFun Blower - Squirrel Cage (12V)." (Dec. 5, 2022), [Online]. Available: https://www.amazon.com/SparkFun-Blower-Squirrel-Cage-12V/dp/B00LPUXVWS?source=ps-sl-shoppingads-lpcontext&ref_=fplfs&psc=1&smid=AM0JQO74J587C.