Project Lab 2

Midterm Report

Dirk Thieme
R11636727
Texas Tech University

October 2022

Abstract

This paper describes the current progress 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. 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 [5]. 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 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 [4] 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 [4], 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.

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 not currently finalized, however in its current state it ingests an MP3 file then converts 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 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.

2 Body

The current state of this project is comprised of the following components, the Software, a Python codebase which consists of the MP3 ingestion, the WAV parsing, the Fast Fourier Transform, and the musical outputs, and the Hardware, which consists of the recorder itself, the control devices, the air production, and the Pi which runs the whole thing.

2.1 Software

2.1.1 Flow of Control Chart

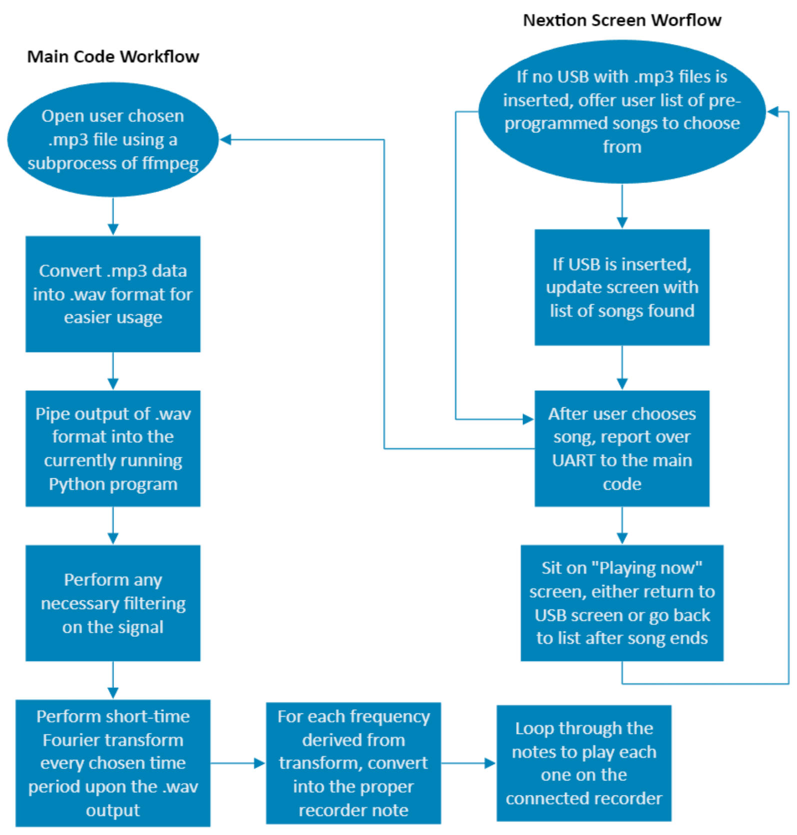
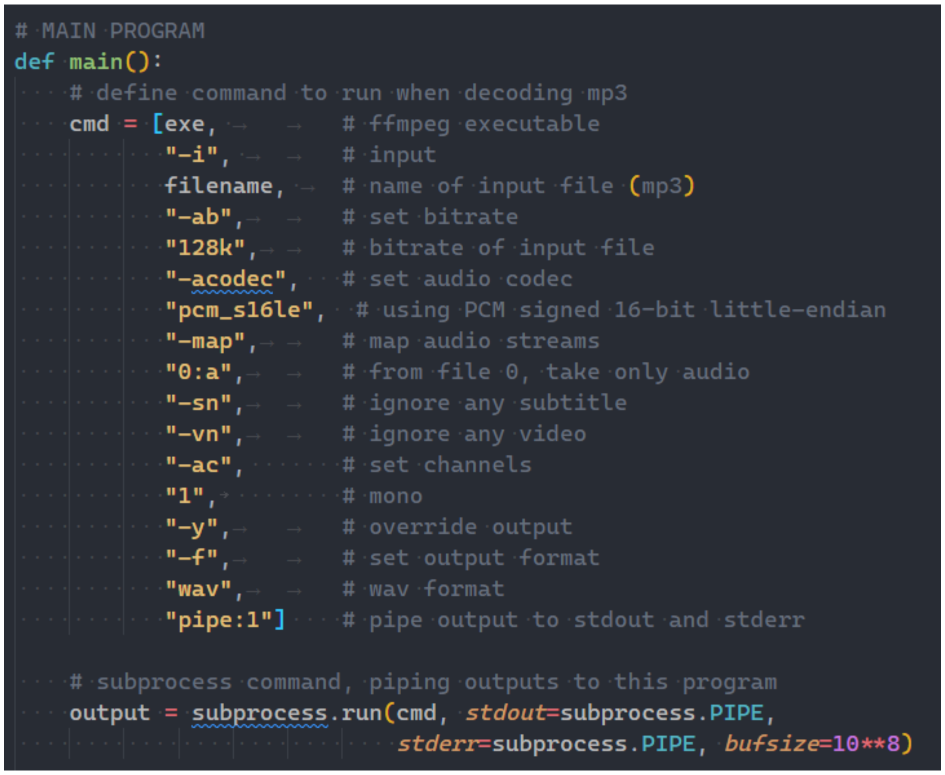

Main Code Workflow
Open user chosen .mp3 file using a subprocess of ffmpeg
Convert .mp3 data into .wav format for easier usage
Pipe output of .wav format into the currently running Python program
Perform any necessary filtering on the signal
Perform short-time Fourier transform every chosen time period upon the .wav output
For each frequency derived from transform, convert into the proper recorder note
Loop through the notes to play each one on the connected recorder
Nextion Screen Worflow
If no USB with .mp3 files is inserted, offer user list of pre-programmed songs to choose from
If USB is inserted, update screen with list of songs found
After user chooses song, report over UART to the main code
Sit on "Playing now" screen, either return to USB screen or go back to list after song ends

Figure I: Main Software Flowchart

As shown in Figure I, this flowchart is the initial path which the code takes. The Nextion workflow has not yet been implemented, as it is unnecessary in determining if the main important code is functional, which itself cannot be yet determined without the construction of the apparatus. As stated previously, the code currently begins at the top if the "Main Code Workflow" section, however in the future it will begin at the "Nextion Screen Workflow". The Nextion screen is a touch screen with a built in microcontroller, which will be used to take user input and send it over to the Pi for actual use.

2.1.2 MP3 Ingestion

The bulk of the MP3 ingestion is performed by the software **ffmpeg**, originally developed by Fabrice Bellard and Bobby Bingham on December 20, 2000 [3]. Shown in Figure II, certain



```python
# 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 II: ffmpeg Command and Output Implementation

command flags must be used upon executing the program in order to ensure proper output

[3]. 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. 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.

2.1.3 WAV Parsing and Processing

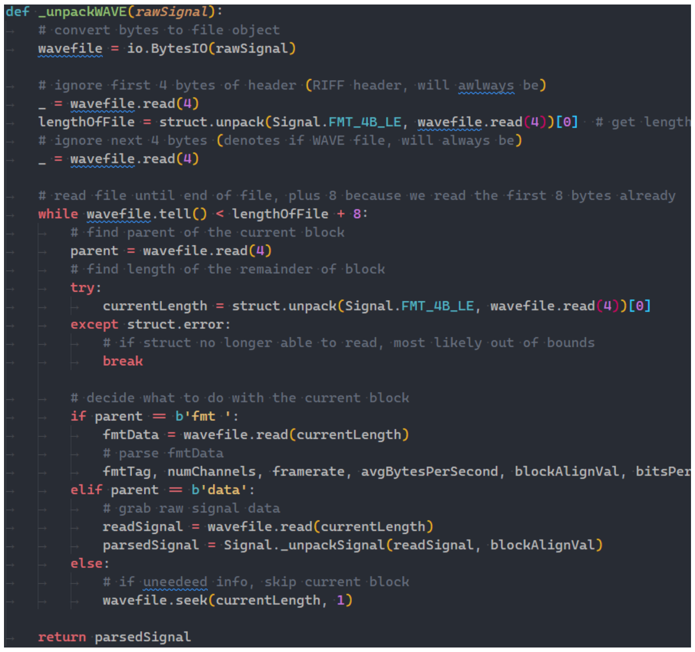
Now that a WAV "file" has been obtained, work must be done in order to ascertain the correct values. The codec which was used to process the song is known as PCM, and the actual values that were output were unsigned 16-bit little-endian chunks. If one simply tried to read the bytes as they were, nothing 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 [6]. 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 [7]. 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 III. After this is



```python
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 III: 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.

2.1.4 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

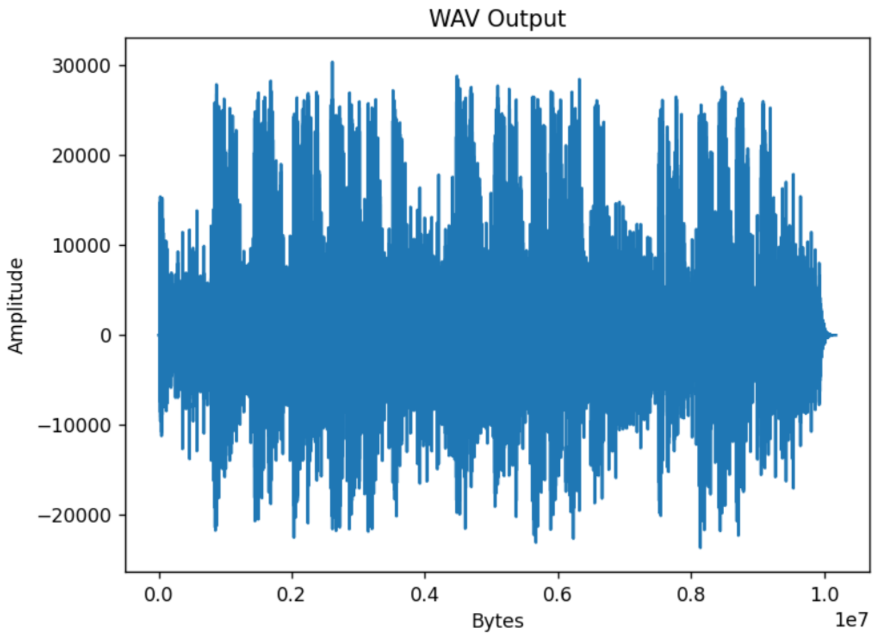


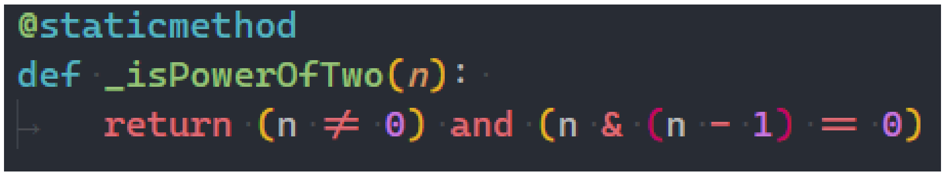
Figure IV: An Example Waveform of a Song

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 Friedrich Gauss [2]. 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 [1]. The standard Discrete Fourier Transform is defined by the equation [2]:

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

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.

A prerequisite of this operation is that the provided signal must be a power of 2, which is where the time savings come from [2]. In order to this, the length of the signal is checked using the code in Figure V, 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

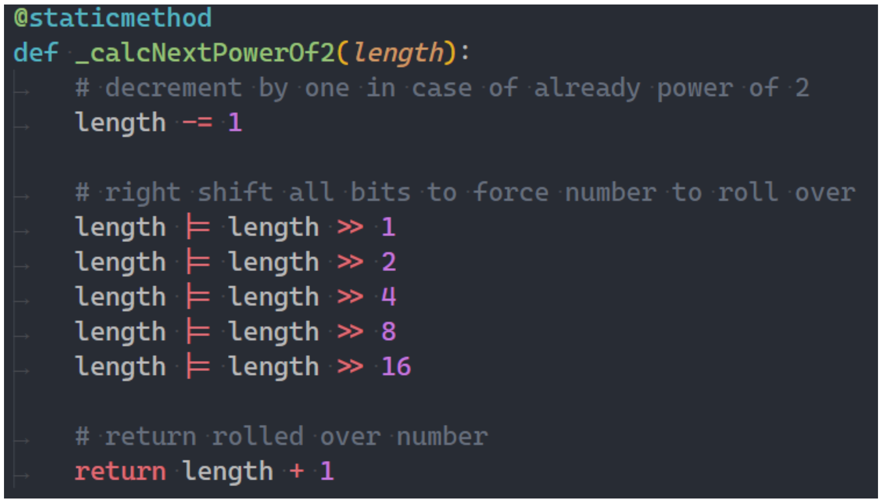


```python
@staticmethod
def _isPowerOfTwo(n):
    return (n != 0) and (n & (n - 1) == 0)
```

Figure V: 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 VI. Once the signal is appropriately padded, the actual math can begin. The main algorithm consists of five steps:



```python
@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 VI: Function to Find the Next Highest Power of Two

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.

2.1.5 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. To do this, simply take the real and imaginary

References

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

[2] “Fast Fourier transform.” (Oct. 19, 2022), [Online]. Available: `https://en.wikipedia.org/wiki/Fast_Fourier_transform`.

[3] “ffmpeg Documentation.” (Oct. 19, 2022), [Online]. Available: `https://ffmpeg.org/ffmpeg.html`.

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

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

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

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