Wireless Audio Transmitter

(WAT)

By

Julian Bell

A capstone project submitted for
Graduation with University Honors

June 1, 2020

University Honors
University of California, Riverside

APPROVED

[illegible]

Dr. Roman Chomko
Department of Electrical and Computer Engineering

Dr. Richard Cardullo, Howard H Hays Jr. Chair, University Honors

Abstract

My Honors Capstone project is a Wireless Audio Transmitter (WAT). It is made up of two modules and a Wi-Fi server. On one module, analog audio data is sampled and digitized by an analog to digital converter (ADC). This discretized data is transmitted wirelessly to the second module via Wi-Fi and a Wi-Fi server. Once the data is received by the other module, playback occurs on both modules when the digital data is made into an analog audio waveform by a digital to analog converter (DAC). Bluetooth is used to sync up playback between both modules to create a surround sound effect. The user can also apply digital audio filters to their music to change the way it sounds.

This was a senior design project done by four people, and I was responsible for the ADC and digital audio filters. I tested, designed, and implemented the ADC and verified that it could successfully sample analog audio perfectly without distortion. I also designed the digital audio filters by calculating filter coefficients for a second order Z transform filter. The user has the option to use a high pass or low pass filter on the sampled music data.

Acknowledgments:

I would like to acknowledge my design teammates Duane Buchholz, Ethan Kim and Robert Jimenez for working on their own subsystems and helping to integrate the project together. Duane Buccholz tested and implemented the DAC and designed the Bluetooth synchronization. Ethan Kim tested and implemented the Wi-Fi communication and the Wi-Fi server. Robert Jimenez tested Bluetooth functionality and designed a boost converter for powering the project. Without them, I would not have been able to complete this design project.

I would like to acknowledge my faculty mentor Professor Roman Chomko. He helped guide our team through the design process. He provided useful information when we ran into testing issues. He also made sure that we were continuing to progress each and every week. This was helpful as we did not fall behind during the process. I would like to thank the honors program for the opportunities and advantages it has provided me throughout my undergraduate education. Finally, I would like to thank Professor Tofigh Heidarzadeh for teaching me the technical writing style and helping to improve my technical writing ability.

Table of Contents:

1. Introduction:

My senior design project is a wireless audio transmitter, or WAT for short. Fundamentally, this is a device which takes in analog audio waveforms from a 3.5 mm AUX input and uses an analog to digital converter to discretize the data as a 16 bit digital number. Once the data is discretized, it is transmitted to another module via a Wi-Fi server. Once the data is received by the other module, Bluetooth is used to sync up playback between both modules to create a surround sound effect. The data is played back by passing through an analog to digital converter where it is played out of a speaker connected by a 3.5 mm AUX cord. The first module also allows a user to apply digital audio effects to a 10 second recorded audio sample. The user can select a low pass filter, which will make the audio sound low pitch, or a high pass filter, which will make the audio sound high pitch.

This was not a solo effort, as that would be a monumental undertaking. I worked with Duane Buccholz, Ethan Kim, and Robert Jimenez. Their contributions will be discussed in the acknowledgments section. This capstone report will be focused on my contribution to this project. I worked primarily on two areas: analog to digital conversion and digital signal processing. Analog to digital conversion is the process of sampling analog audio data into discrete, or digital values. This way, the audio can be processed by a microprocessor and transmitted via Wi-Fi. Digital signal processing the process of taking the sampled music data and applying digital filters to it. These filters serve as audio effects that can change the way the music sounds. The rest of this report is written in a technical writing style, so everything is written in the third person.

Definitions and Acronyms:

WAT: Wireless Audio Transceiver (The overall project)

WAT1: Wireless Audio Transceiver 1 (The main module)

WAT2: Wireless Audio Transceiver 1 (The second module)

RPi3: Raspberry Pi 3 B+ (Each WAT module runs off of a Raspberry Pi 3 B+)

AUX: Auxiliary Port (A standard communications port that allows for auxillary inputs of analog audio data from a phone, microphone, etc.)

ADC: Analog to Digital Converter (Samples analog signals, in the case audio data over AUX, to digitize the music data.)

DAC: Digital to Analog Converter (Converts digital signals, in the case sampled audio data, back into a continuous signal that is then sent to a speaker.)

SNR: Signal to Noise Ratio (A metric for how “good” audio data is by comparing a desired signal in dB to the noise of the signal in dB. The DAC and ADC have an SNR greater than 40dB)

THD+n: Total Harmonic Distortion plus Noise (The sum of all harmonic distortion. The ADC and DAC have less than 60dB (0.1%) THD+n)

SPS: samples per second (ADC samples at 48,100 samples per second.)

I2C: Inter-Integrated Circuit (serial communication protocol)

Hz: Hertz (The unit of frequency, equivalent to one cycle per second.)

DSP: Digital Signal Processing (DSP to design and implement various filters)

HPF: High Pass Filter (filters out lower frequencies)

LPF: Low Pass Filter (filters out higher frequencies)

3. Experimentation Procedure:

Experiment 1A and 1B:

Julian Bell designed experiments centered around ADC testing, audio sampling, and audio filtering. The first test used the ADS1115 to verify its ability to sample music cleanly. This ADC was 16 bit, 860 SPS. This test was done by using a TRRS adapter which allows both audio channels to be wired to the ADC. The ADC communicated with an RPi3 via I2C. A Python 3 script was created to record 10 seconds of audio data and to convert it from a numpy array to a .wav file. The expected results were for audio to play cleanly, but instead there was only static white noise. This was a flawed test, as there was no way to analyze where the issue was when the only thing known is that music doesn't play properly. The issue could be the sampling rate, chunk size, the SNR, the THD+n, or potentially anything else.

Julian Bell used a variable AC waveform generator to create a sinusoid with a constant amplitude, frequency, and phase shift. If the input was constant and predictable, he could properly analyze errors in the ADC output waveform. His input waveform had the following characteristics: amplitude of 1 V_{pp}, phase shift of 0 radians, and a range of frequencies. The frequencies include: 10 Hz, 100 Hz, 200 Hz, 400 Hz, 500 Hz, 1 kHz, 5 kHz, 10 kHz, and 20 kHz. Humans can only hear up to about 20 kHz, so there was no need to test past this point.

The ADS1115 had a variable sampling rate that goes up to 860 SPS. In order to increase this, Julian Bell overclocked the RPi3's I2C bus in order to increase its sampling rate. The first test analyzed the sampling rate due to the overclocked bus and the second test analyzed the digital reconstruction of different analog waveform frequencies. Due to the results which will be discussed in the next section, the ADS1115 was insufficient for sampling audio as the overclocked sampling was not high enough.

Experiment 2:

Julian Bell experimented with the SSS1629 DAC and ADC combo. The ADC was 16 bit, 48 kSPS and used USB and I2C for serial communication. It has a variable sampling rate that goes up to 48 kSPS. He performed a test that measures SNR, THD+n, and waveform reconstruction. He used the same amplitude and phase shift, but instead tested the following frequencies: 100 Hz, 1 kHz, 10 kHz, and 15 kHz.

Experiment 3:

Julian Bell also performed filtering tests. Since this added too much latency to the live playback and there wasn't enough time to fully implement them, he did filtering separately as a proof of concept. He tested two different second order Z-transform filters, with the intent of making a low pass and high pass filter. He recorded 10 seconds of audio data and played it back as it records. Then, the data was convoluted with the discrete time impulse response of the filter. He is using filter coefficients in the form:

$H(z) = \frac{c}{(1+az^{-1})(1+a_cz^{-1})}$, where ac is the complex conjugate of a. Through calculations, he found that good coefficient values were a = .9 +.1j and ac = .9 - .1j for a low pass filter, and a = 1.2 + .4j and ac = 1.2 - .4j for a high pass filter. He found the c value by substituting $z = e^{j\omega}$ at ω = pi, which yielded c = .02 for a low pass filter and c = .2 for a high pass filter.

4. Experimentation Results:

Experiment 1A:

Figure 4.2.1 shows the results from the first experiment which tested the sampling rate after overclocking the I2C at varying AC waveform frequencies for the ADS1115. The overclocked sampling rate of 2500 SPS is still insufficient, as the Nyquist rate for perfect audio sampling is 44.1 kSPS.

Figure 4.2.1:

Frequency of Sinusoid (Hz)	Number of Samples	Elapsed Time (s)	Average Samples Per Second (SPS)
10	1000	0.3981	2512
100	1000	0.4002	2498
200	1000	0.3993	2504
400	1000	0.3995	2502
500	1000	0.4002	2498
1000	1000	0.4006	2496
5000	1000	0.3990	2506
10000	1000	0.3983	2510
20000	1000	0.3986	2508

Experiment 1B:

Figures 4.2.2, through 4.2.4 show the results from testing the digital waveform reconstruction at varying analog frequencies for the ADS1115. As the input sinusoid's frequency went up, the reconstructed sinusoid's period went down. This was the result of undersampling. The ADS1115 had failed feasibility tests, and was insufficient for sampling audio data.

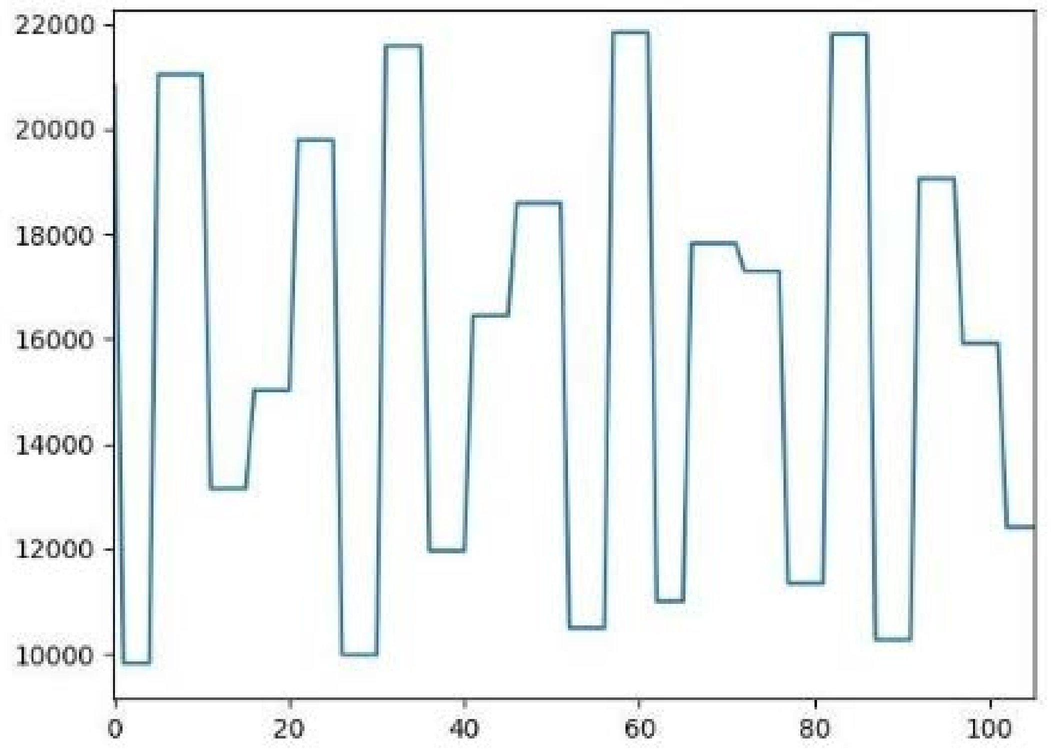


Figure 4.2.2: 200 Hz input sinusoid reconstruction (raw data) for ADS1115

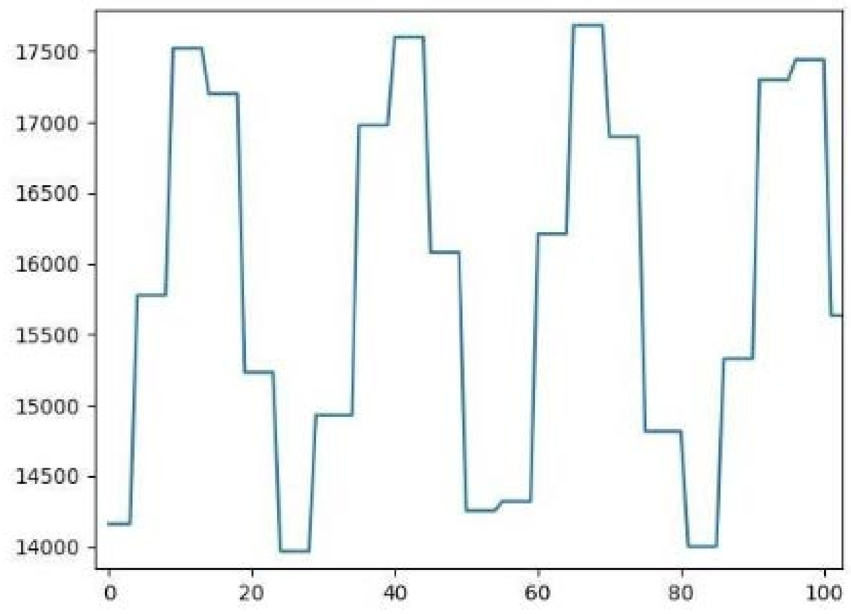


Figure 4.2.3: 400 Hz sinusoid reconstruction (raw data) for ADS1115:

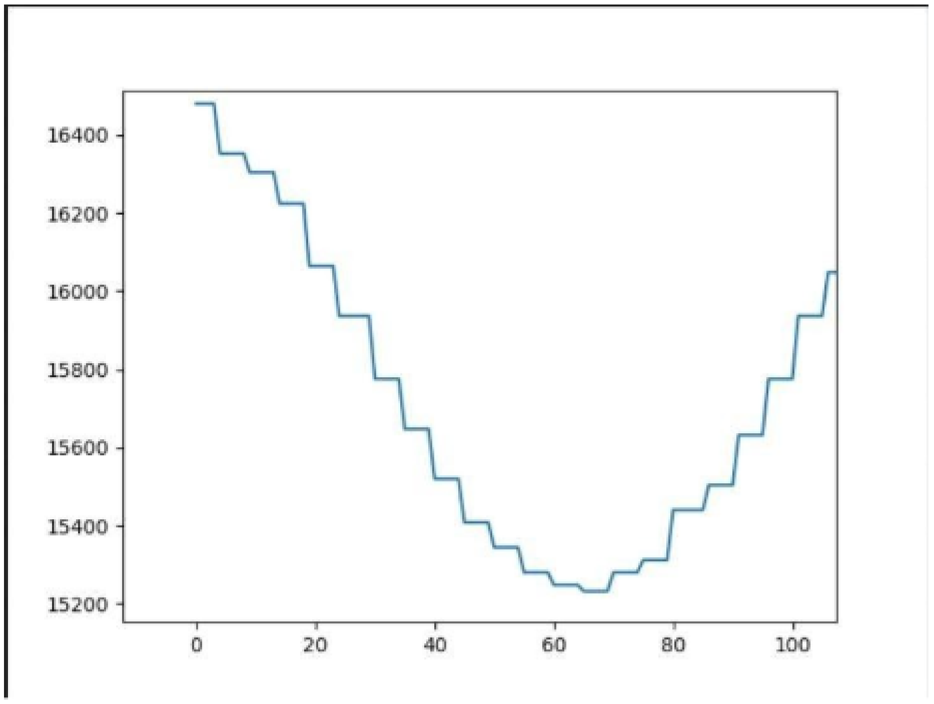


Figure 4.2.4: 1 kHz sinusoid reconstruction (raw data) for ADS1115:

Experiment 2:

Figures 4.2.5 through 4.2.8 show the FFT spectrum graphs for a constant sinusoidal analog input to the SSS1629 ADC, where blue is the left channel, and orange is the right channel. The peak of the FFT spectrum corresponds to the dominant frequency of the waveform and the rest of the graph is minor noise detected. The SSS1629 had a variable sampling rate that went up to 48 kSPS. This ADC was sufficient in reconstructing the full audible range of frequencies. There was interference that is noticeable at the 60 Hz band. This is because the 3.5mm AUX cord used was broken and picked up electromagnetic interference from the wall power. Purchasing a new cord fixed the problem.

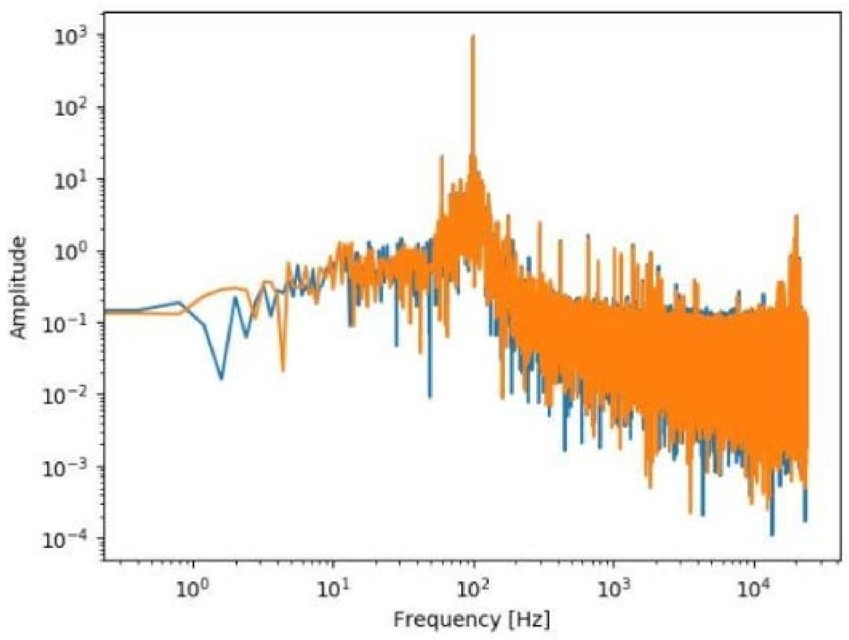


Figure 4.2.5: 100 Hz sinusoid reconstruction (FFT data) for SSS1629:

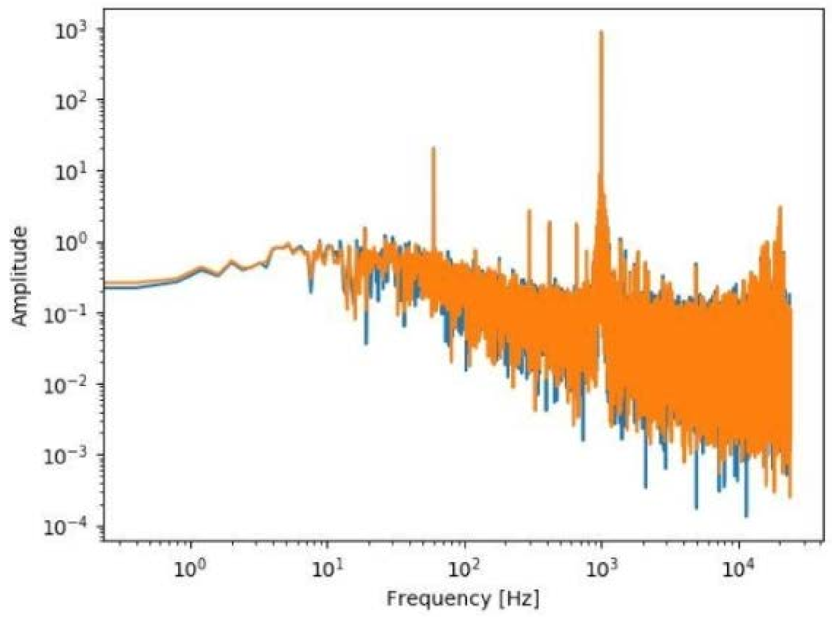


Figure 4.2.6: 1 kHz sinusoid reconstruction (FFT data) for SSS1629:

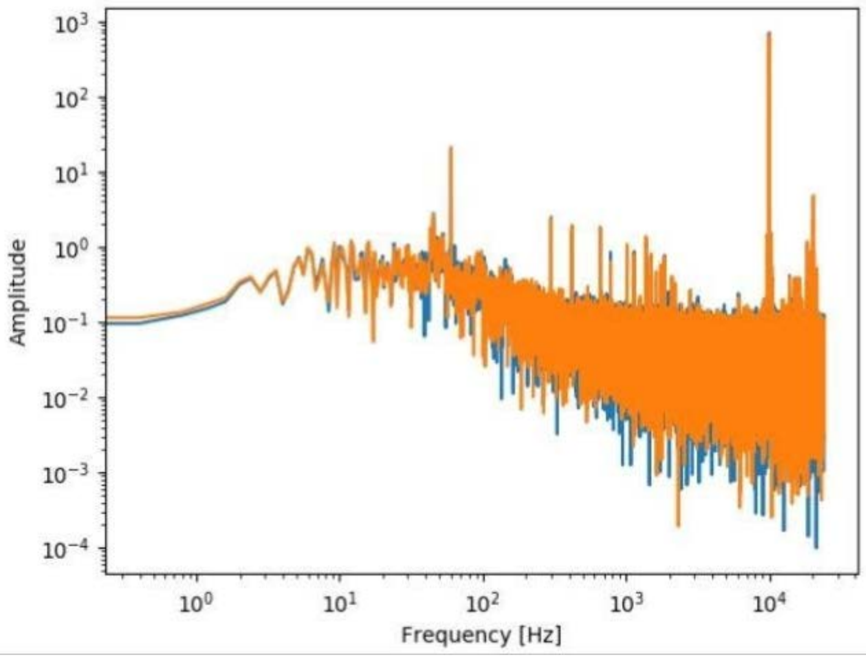


Figure 4.2.7: 10 kHz sinusoid reconstruction (FFT data) for SSS1629:

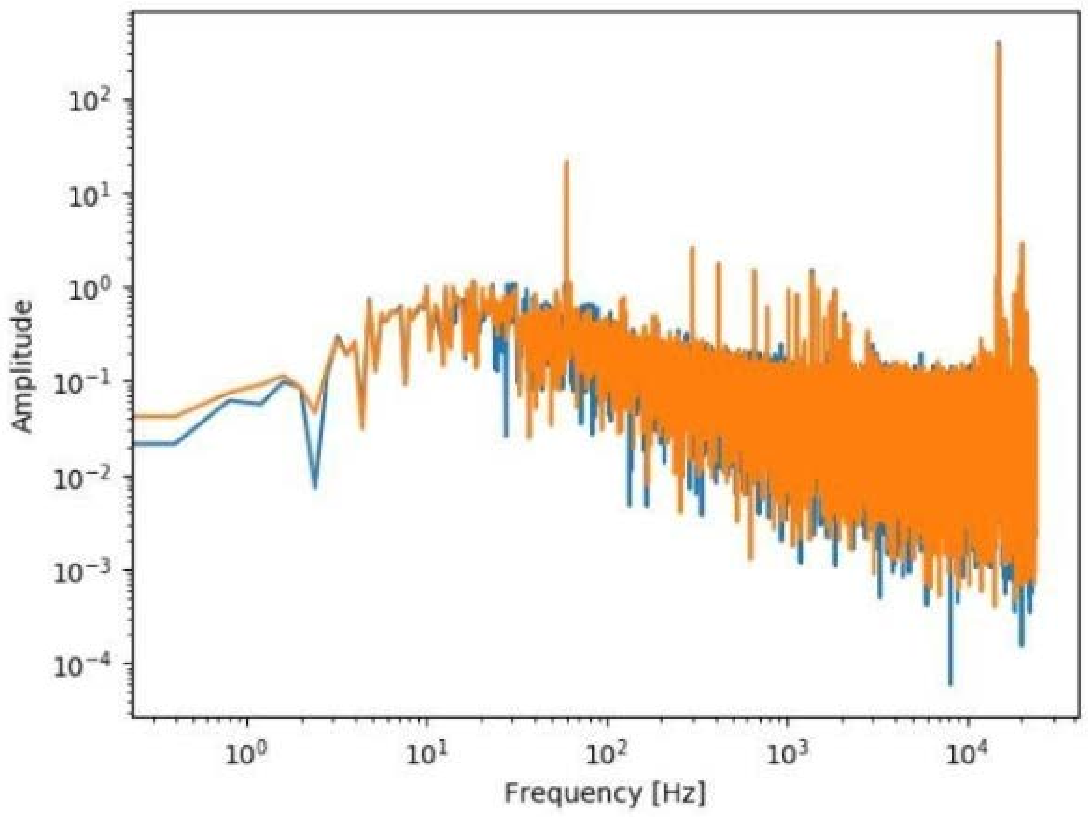


Figure 4.2.8: 15 kHz sinusoid reconstruction (FFT data) for SSS1629:

Experiment 3:

These are the FFT spectrum results from the audio filtration test, where blue is the left channel and orange is the right channel. Figure 4.2.9 shows the unfiltered 10 second audio sample and Figure 4.2.10 shows that same audio sample after it has passed through a low pass filter. Figure 4.2.11 and 4.2.12 show the FFT spectrum of a different unfiltered 10 second audio sample and the audio after passing through a high pass filter.

Figures 4.2.13 and 4.2.1.15 show the zero-pole plot of the poles and zeros of the Z-transform of the low and high pass filters. The HPF had poles outside of the unit circle, and the LPF had poles within the unit circle, which was expected. Figured 4.2.14 and 4.2.16 show the FFT filter response of the magnitudes of both the HPF and LPF. The LPF's magnitude decreased as the frequency went up, and the HPF's magnitude increased as the frequency went up, which was also expected.

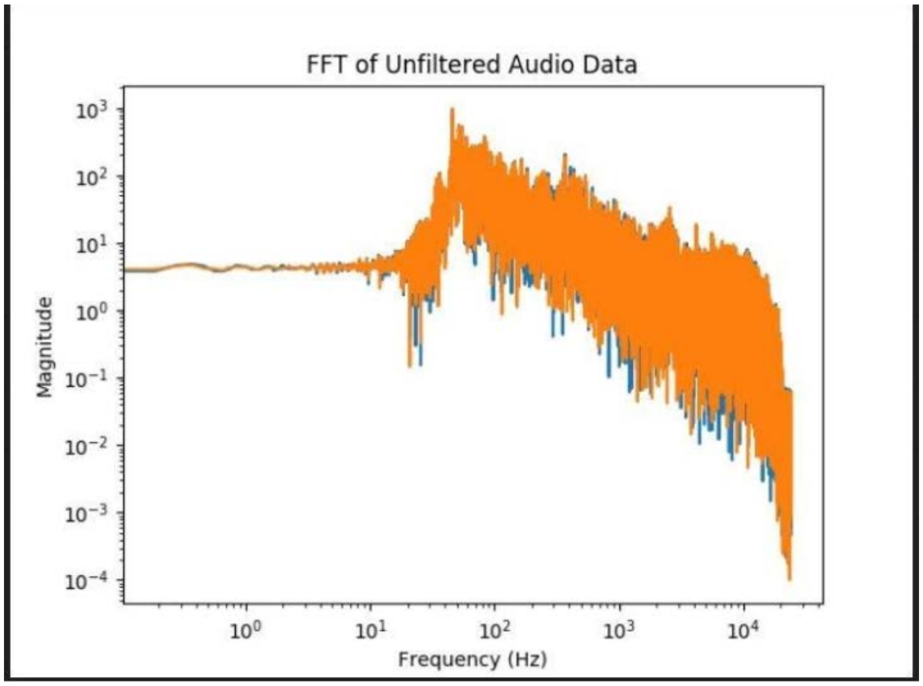


Figure 4.2.9: Unfiltered

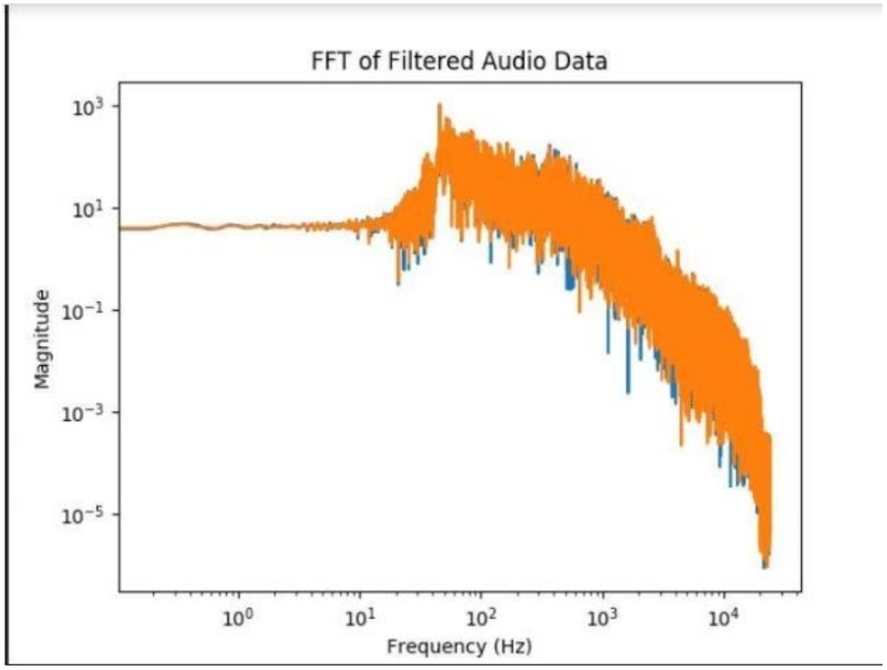


Figure 4.2.10: Low Pass Filter

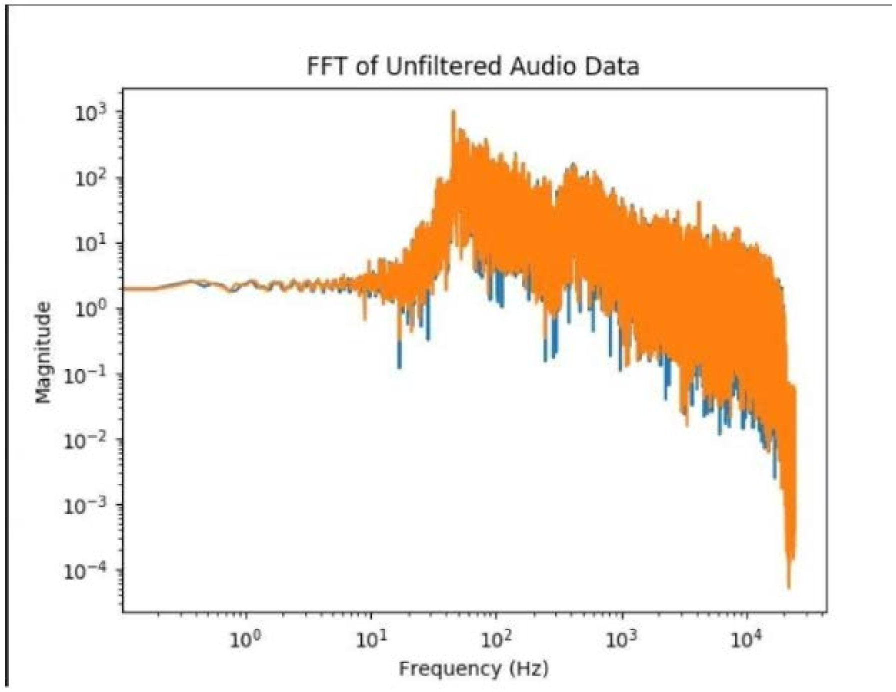


Figure 4.2.11: Unfiltered

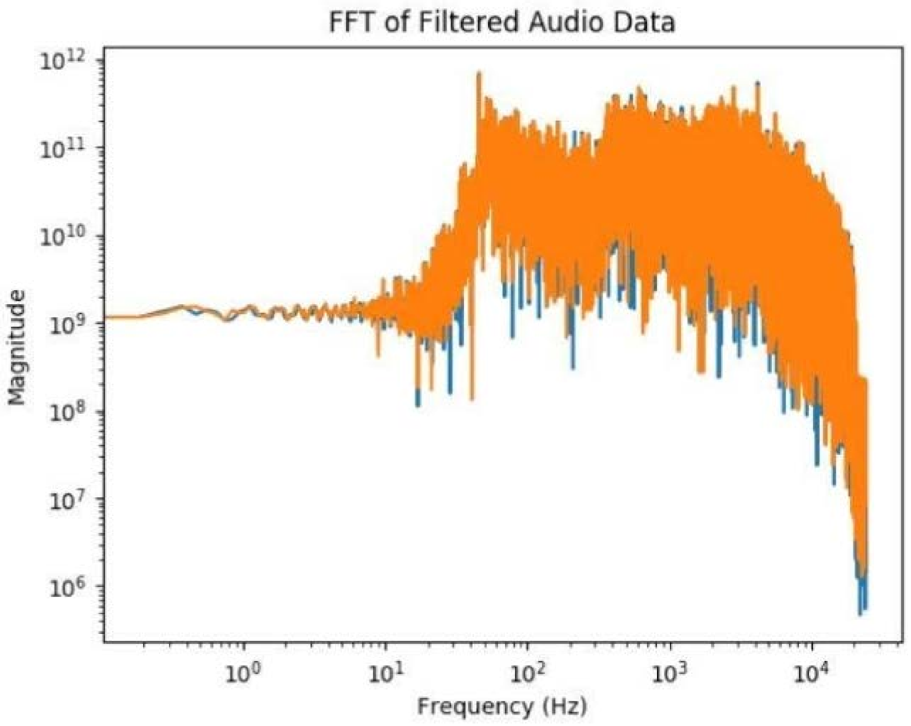


Figure 4.2.12: High Pass Filter:

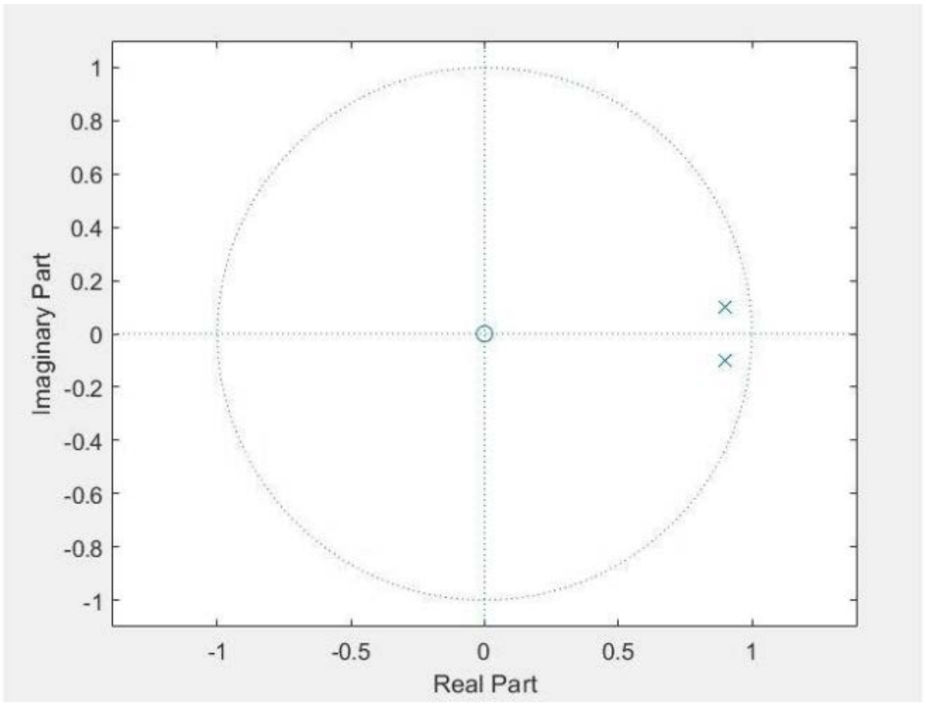


Figure 4.2.13: Poles (x) and Zeros (o) LPF:

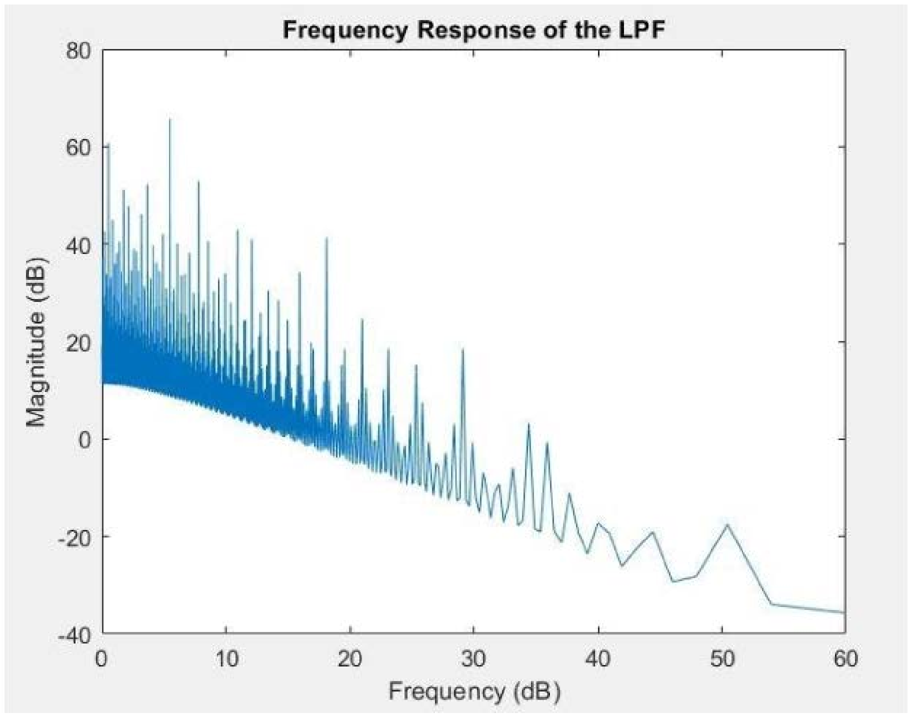


Figure 4.2.14: LPF

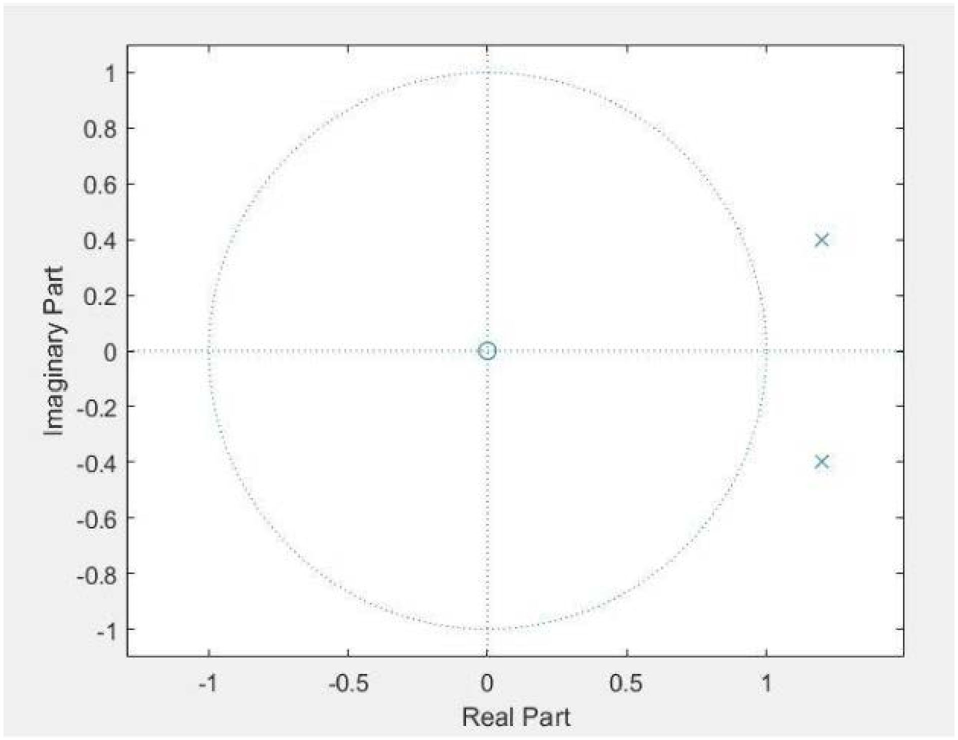


Figure 4.2.15: Poles (x) and Zeros (o) HPF:

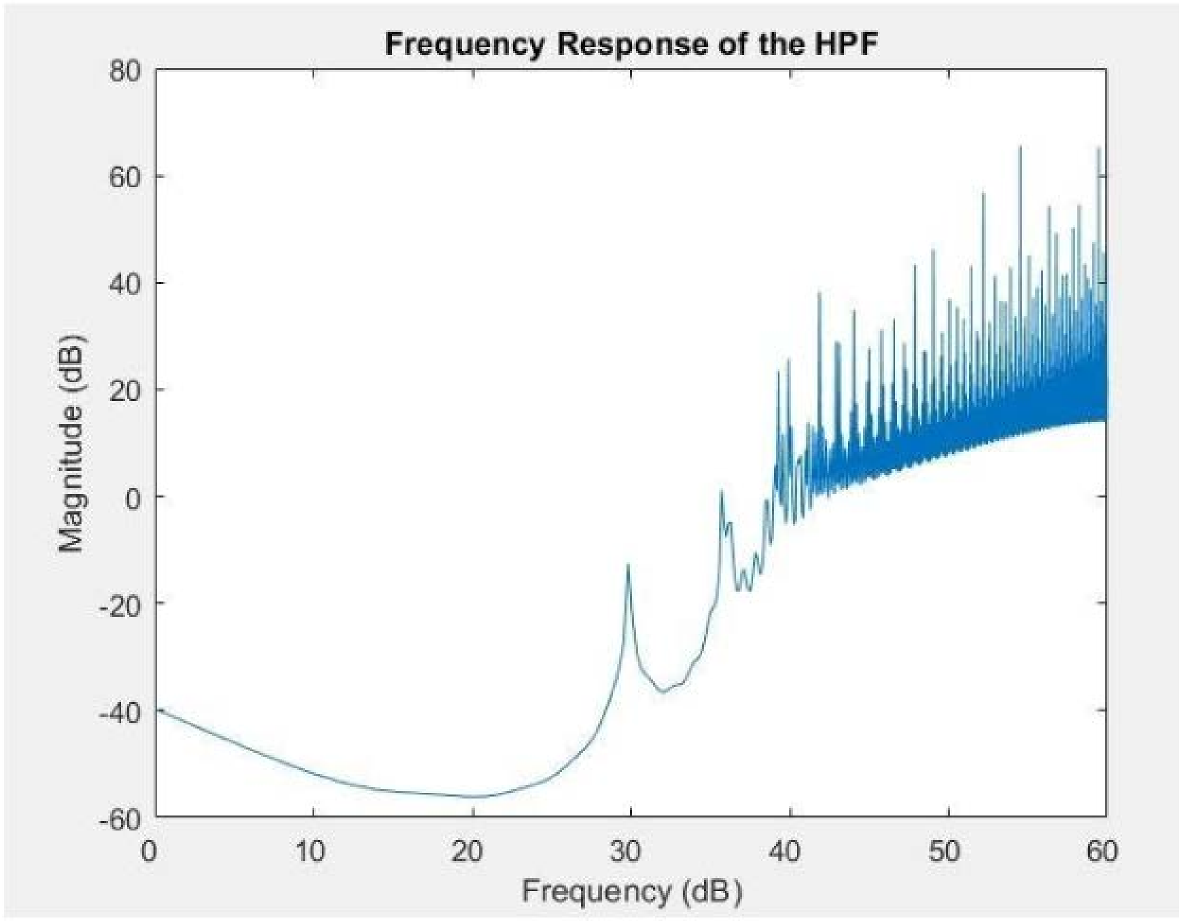


Figure 4.2.16: HPF

5. High Level Design:

Figure 5.1.1 shows the system block diagram of the WAT module. Julian Bell was responsible for the ADC and DSP blocks. He performed feasibility tests on the ADC measuring its sampling rate, SNR, THD+n, and effective waveform reconstruction. He also sampled audio data in half second chunks and stored them into .wav files. He also designed a filter script which can record audio and apply a high or low pass filter. The filters were calculated and designed in Matlab, then imported to Python 3.

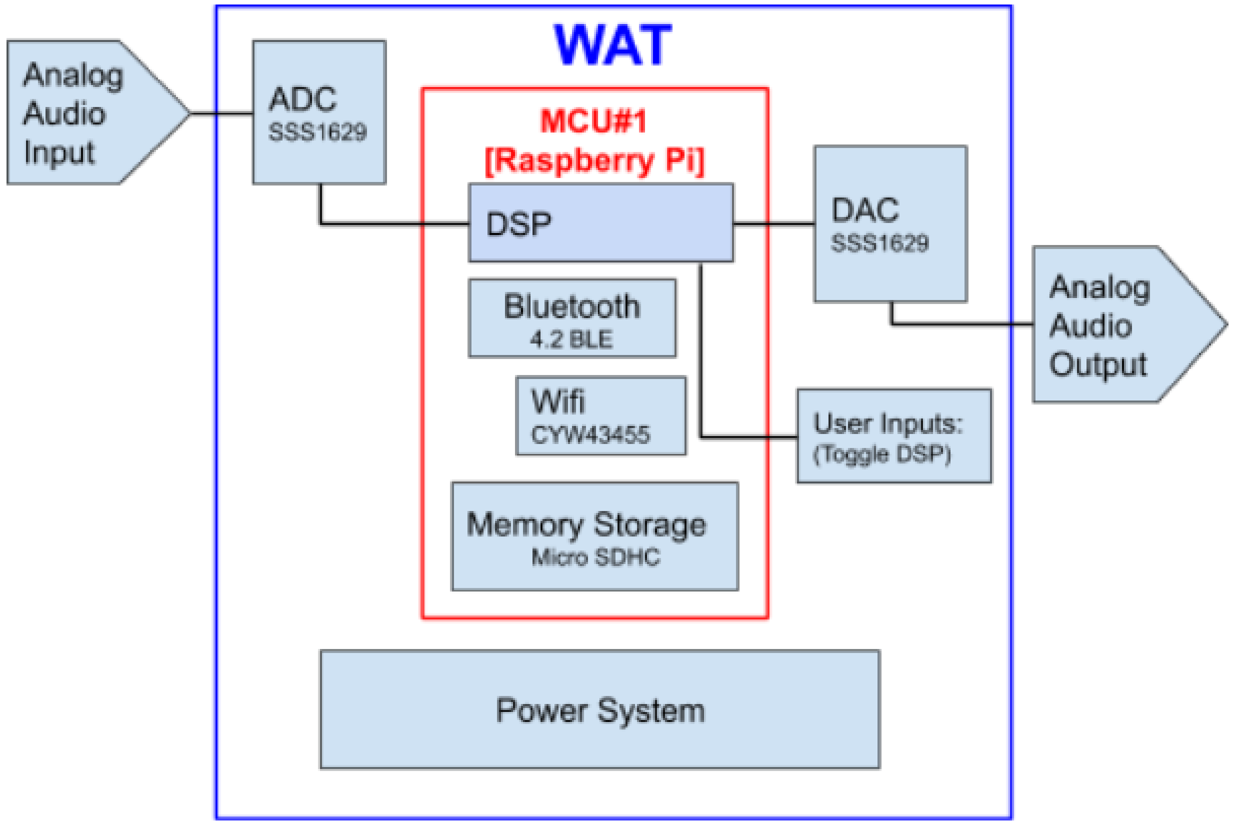


Figure 5.1.1: Final System Block Diagram

Figure 5.1.2 shows the high level interaction between WAT modules and the Wi-Fi server. Digitized data is transmitted to a Wi-Fi server via Wi-Fi and then it is transmitted from the server to another module. Once the data has been transmitted, the modules communicate between each other via Bluetooth to sync up audio playback.

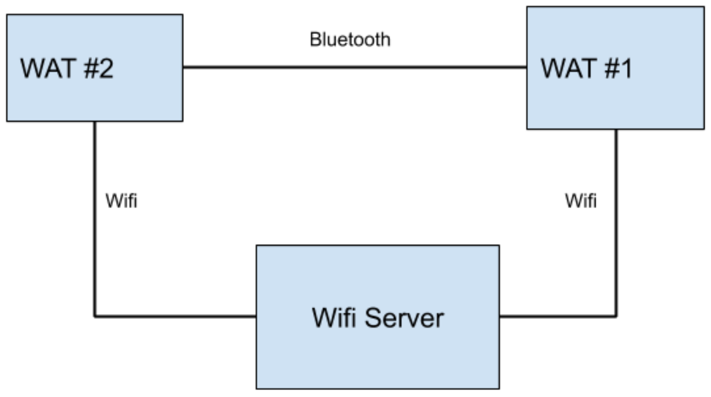


Figure 5.1.2: High Level Hardware Interaction

Figure 5.1.3 shows the high level Python software flowchart. Analog audio is sampled by a sampling script, then filtered by DSP, and transmitted via a Wi-Fi communication script to the other module. Once data is transmitted, bluetooth will sync up playback in the play script.

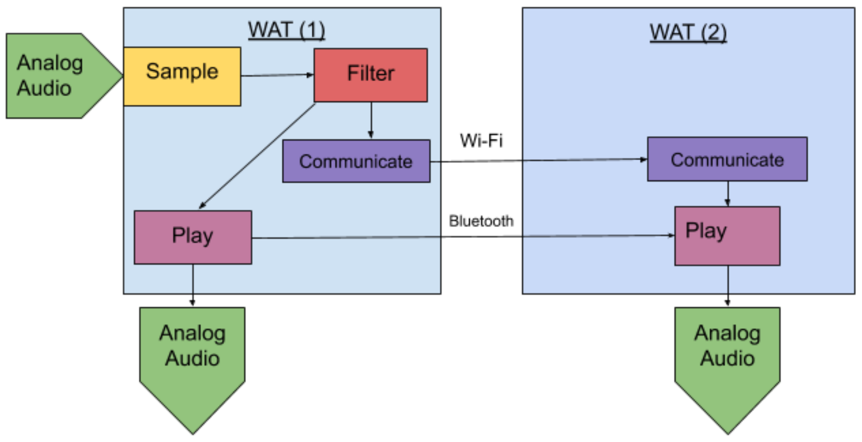


Figure 5.1.3: High Level Software Flowchart

6. Hardware Architecture:

Figure 6.1.1 shows the block diagram of the hardware architecture for the SSS1629 USB ADC/DAC combo. The WAT uses the 3.5 mm AUX MIC Line in as the input to the ADC. Figure 6.1.2 shows the pinout of the SSS1629 chip.

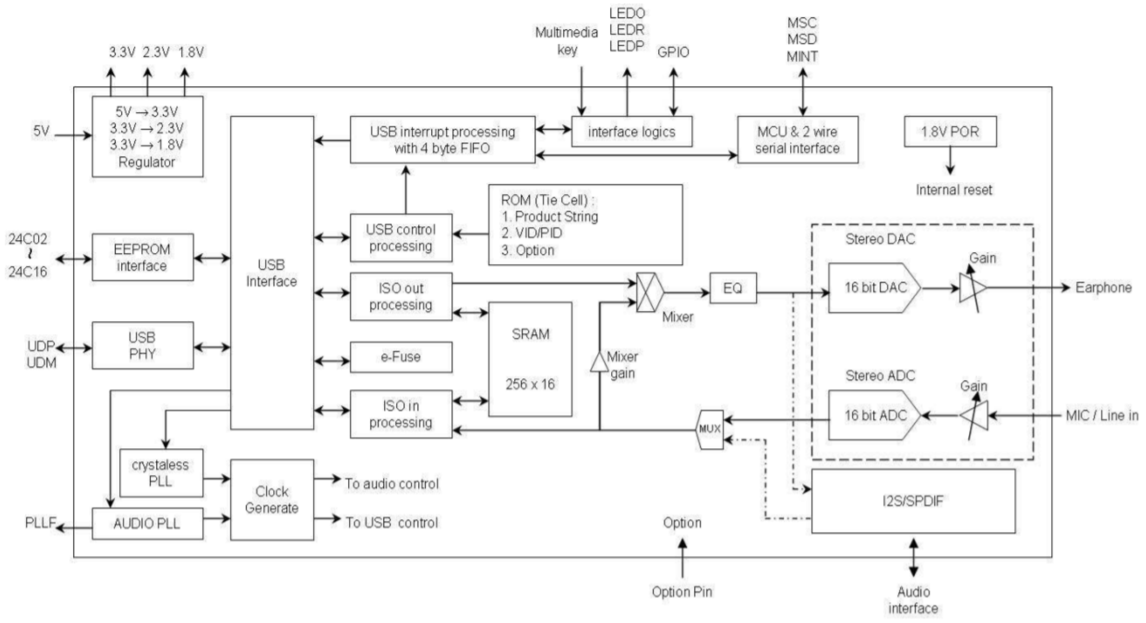


Figure 6.1.1: SSS1629 DAC and ADC Schematic:

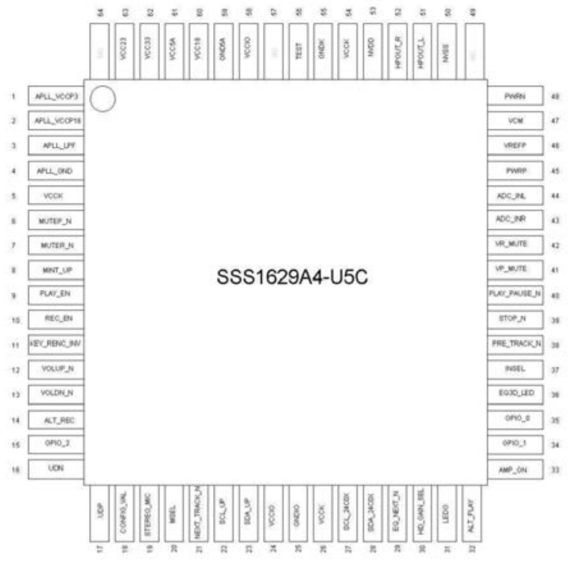


Figure 6.1.2: SSS1629 Chip Pinout

7. Software Architecture:

The flowchart in Figure 7.1.1 is for the sampling script. initializes arrays and objects, then samples half a second of data stored into a numpy array. It is then converted to a .wav file and stored in a file directory.

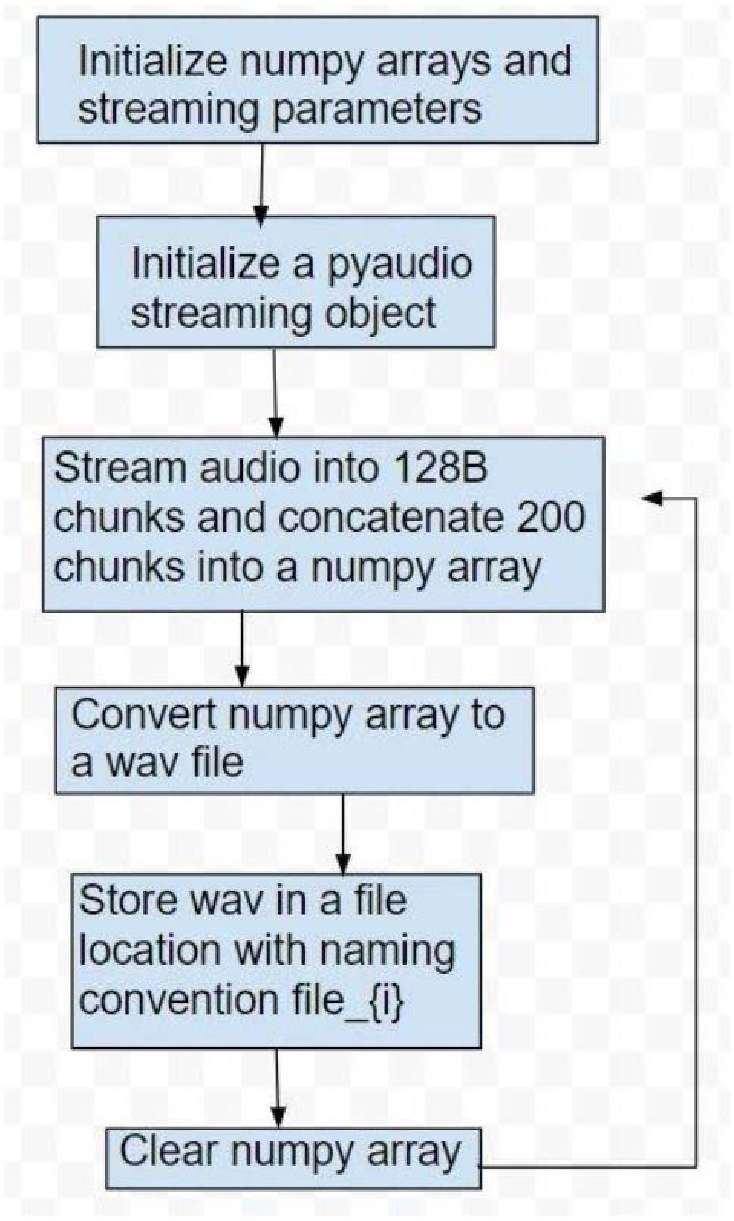

Initialize numpy arrays and streaming parameters
Initialize a pyaudio streaming object
Stream audio into 128B chunks and concatenate 200 chunks into a numpy array
Convert numpy array to a wav file
Store wav in a file location with naming convention file_{i}
Clear numpy array

Figure 7.1.1: Audio Sampling Script:

The script in Figure 7.1.2 records data and applies a user selected filter to it. The filters are created with filter coefficients and plays back a .wav file. It first plays back the unfiltered audio, then plays back the filtered audio data so the user can hear the difference.

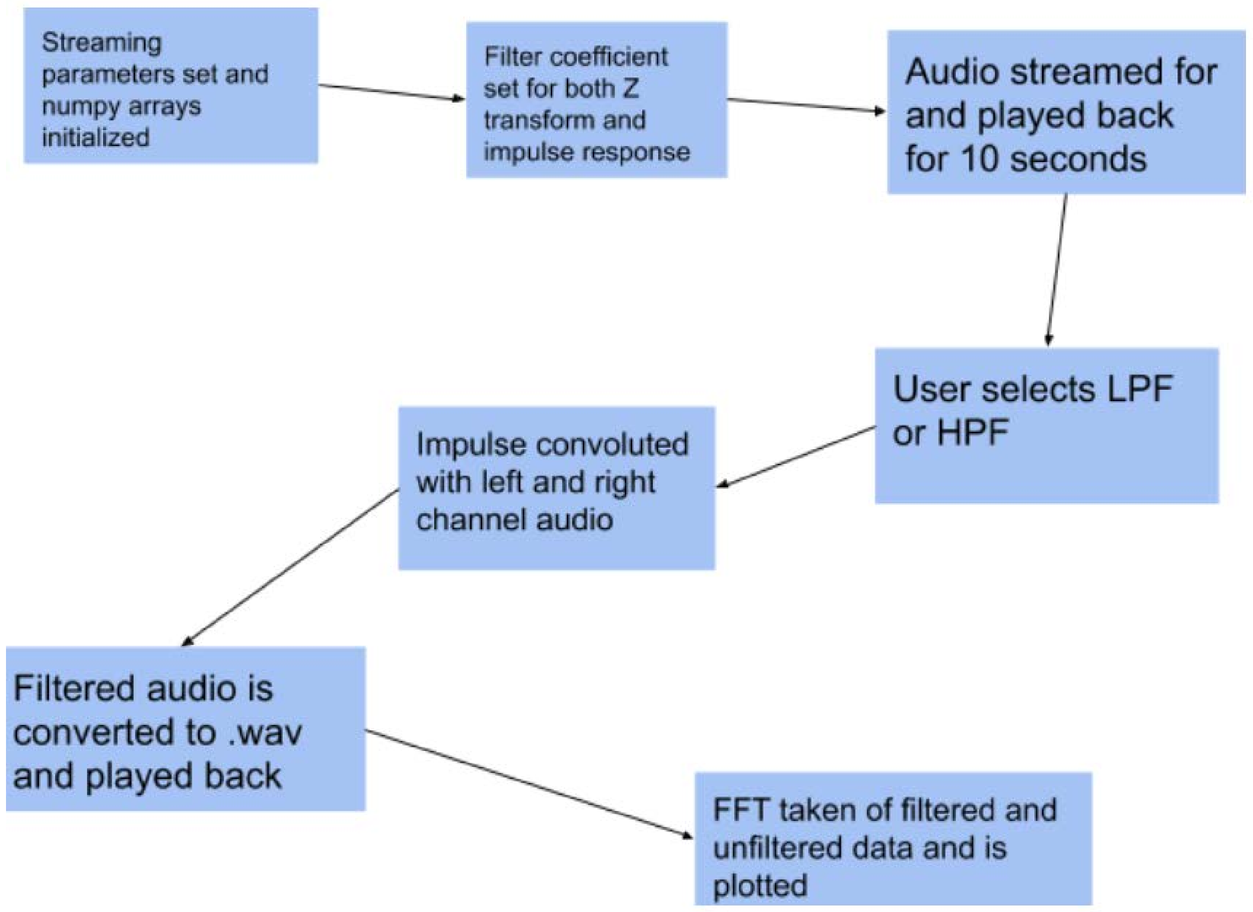
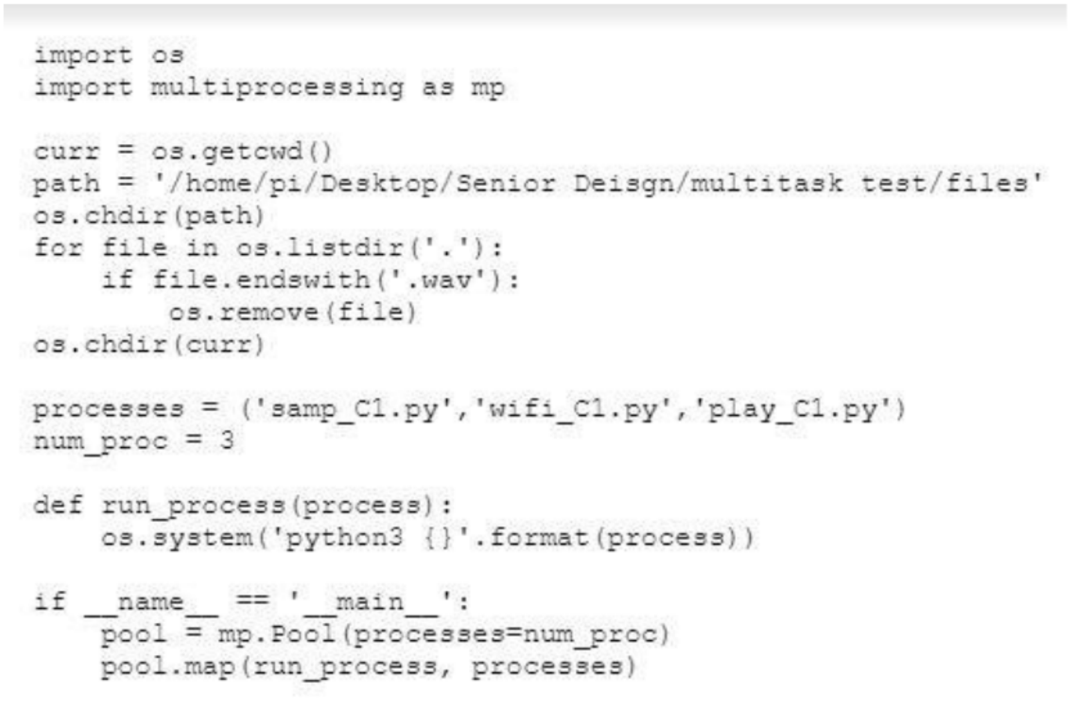

Streaming parameters set and numpy arrays initialized
Filter coefficient set for both Z transform and impulse response
Audio streamed for and played back for 10 seconds
User selects LPF or HPF
Impulse convoluted with left and right channel audio
Filtered audio is converted to .wav and played back
FFT taken of filtered and unfiltered data and is plotted

Figure 7.1.2 Audio Filtering Script:

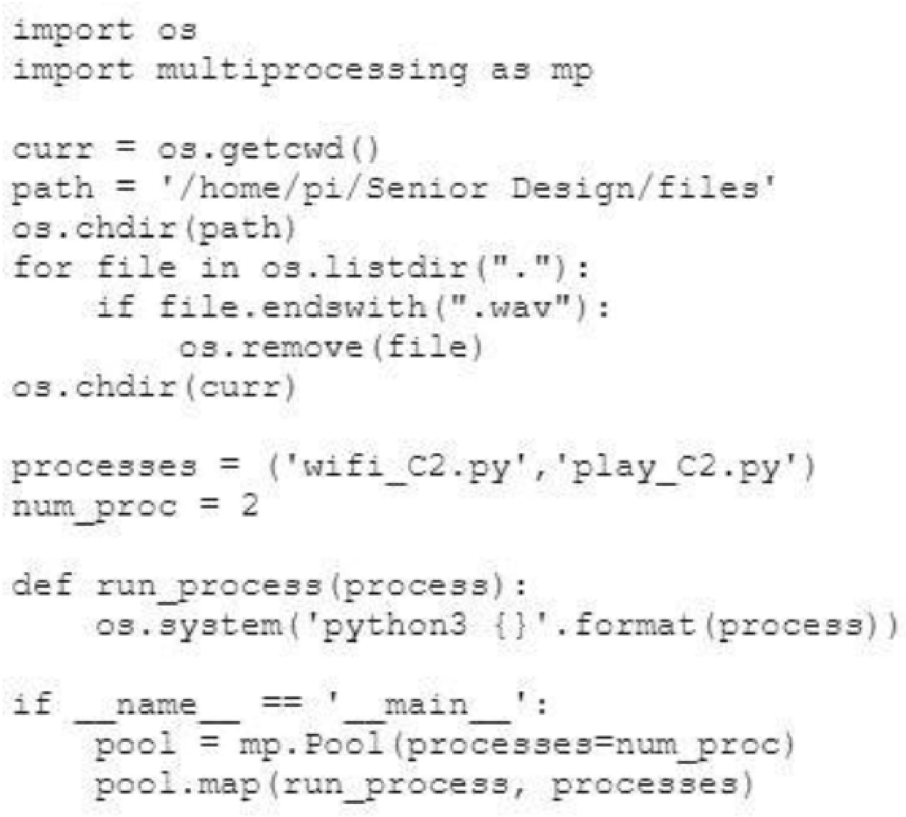
Figure 7.1.3 shows a Python script “mulit.pi” for WAT1 which calls three other scripts to run them in parallel using the multi-processing and os libraries. First, the script clears all previously recorded audio files from the file directory in order to sample new ones. This script overall allows WAT1 to sample, playback, send audio data over Wi-Fi, and communicate with WAT2 over BT simultaneously. It simply requires that all of the other scripts running be in the same directory that the multi.py script has access to.



```
import os
import multiprocessing as mp

curr = os.getcwd()
path = '/home/pi/Desktop/Senior Deisgn/multitask test/files'
os.chdir(path)
for file in os.listdir('.'):
    if file.endswith('.wav'):
        os.remove(file)
os.chdir(curr)

processes = ('samp_C1.py','wifi_C1.py','play_C1.py')
num_proc = 3

def run_process(process):
    os.system('python3 {}'.format(process))

if __name__ == '__main__':
    pool = mp.Pool(processes=num_proc)
    pool.map(run_process, processes)
```

Figure 7.1.3: WAT1 Module 1 multi.py Script:

Figure 7.1.4 shows a Python script “mulit_C2.pi” for WAT2 which calls three other scripts to run them in parallel using the multi-processing and os libraries. First, the script clears all previously recorded audio files from the file directory in order to sample new ones. This script overall allows WAT1 to playback and send audio data over Wi-Fi, and communicate with WAT1 over BT simultaneously. It simply requires that all of the other scripts running be in the same directory that the multi_C2.py script has access to.



```python
import os
import multiprocessing as mp

curr = os.getcwd()
path = '/home/pi/Senior Design/files'
os.chdir(path)
for file in os.listdir("."):
    if file.endswith(".wav"):
        os.remove(file)
os.chdir(curr)

processes = ('wifi_C2.py','play_C2.py')
num_proc = 2

def run_process(process):
    os.system('python3 {}'.format(process))

if __name__ == '__main__':
    pool = mp.Pool(processes=num_proc)
    pool.map(run_process, processes)
```

Figure 7.1.4 WAT2 multi_C2.py Script:

8. Low Level Design:

Module 1: ADC

This module includes digital sampling with the SSS1629 ADC and perfectly reconstructing analog waveforms as digital signals.

Processing narrative for ADC

All sampling is done on the WAT1 and handled by the following Python script:

- samp_C1.py

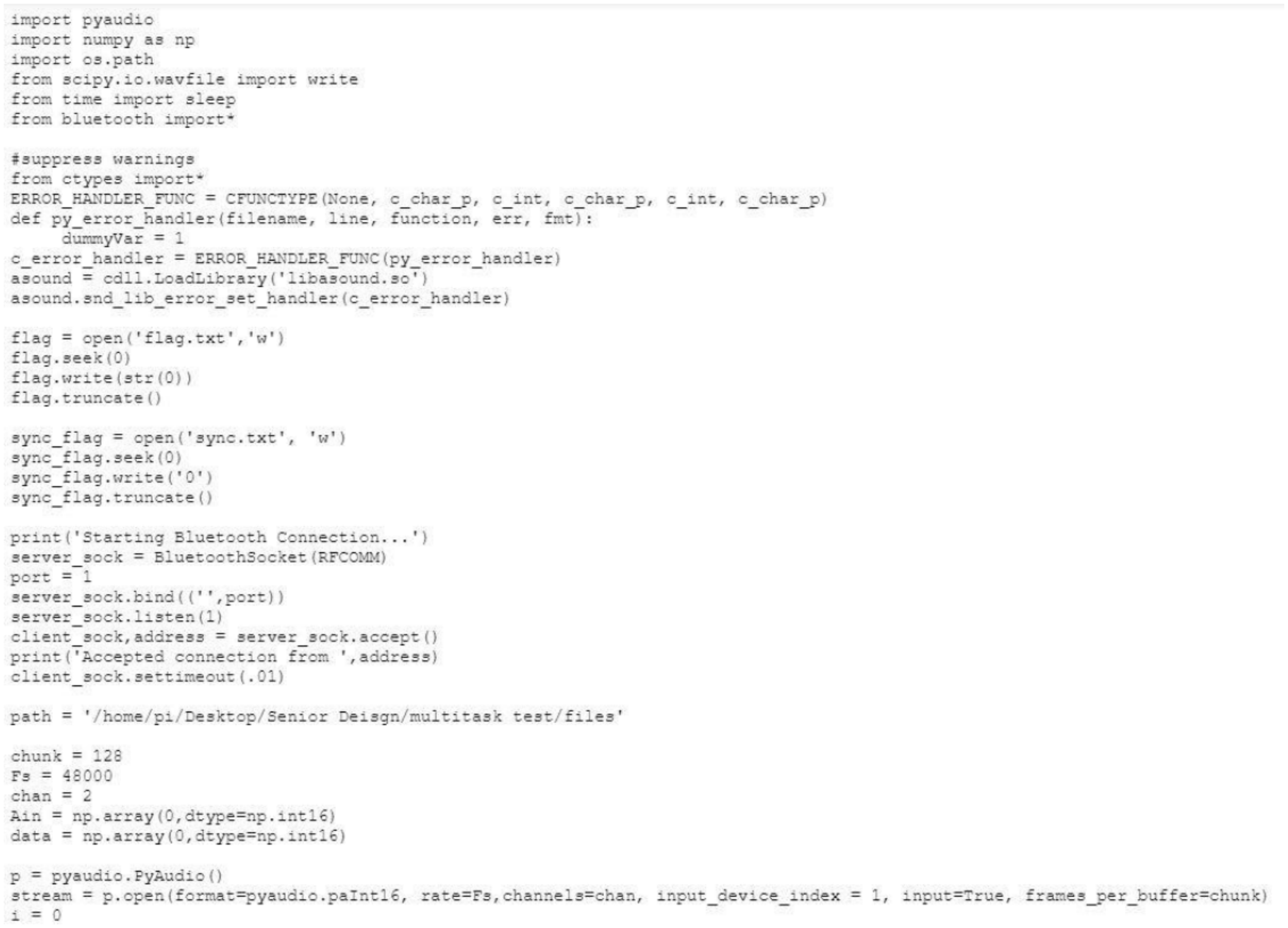
ADC interface description

The ADC module received an input from a 3.5 mm AUX cord into a TRS port attached to the SSS1629. Once the analog audio was sampled and digitized, I2C and USB are used to transfer the digital output audio signal to the RPi3’s ARM CPU so it could be processed and transmitted via Wi-Fi.

ADC processing details

This script in Figure 8.1.1 and 8.1.2 used the pyaudio library in order to stream audio, numpy in order to use a numpy array data type, os.path in order to access file directories, scipy.io to write .wav files, time in order to use delays, and the pybluez library to access the RPi3’s built in Bluetooth BLE 4.1 module. The pyaudio library cluttered the command terminal with useless error messages, so the ctypes library was used to suppress the error messages.

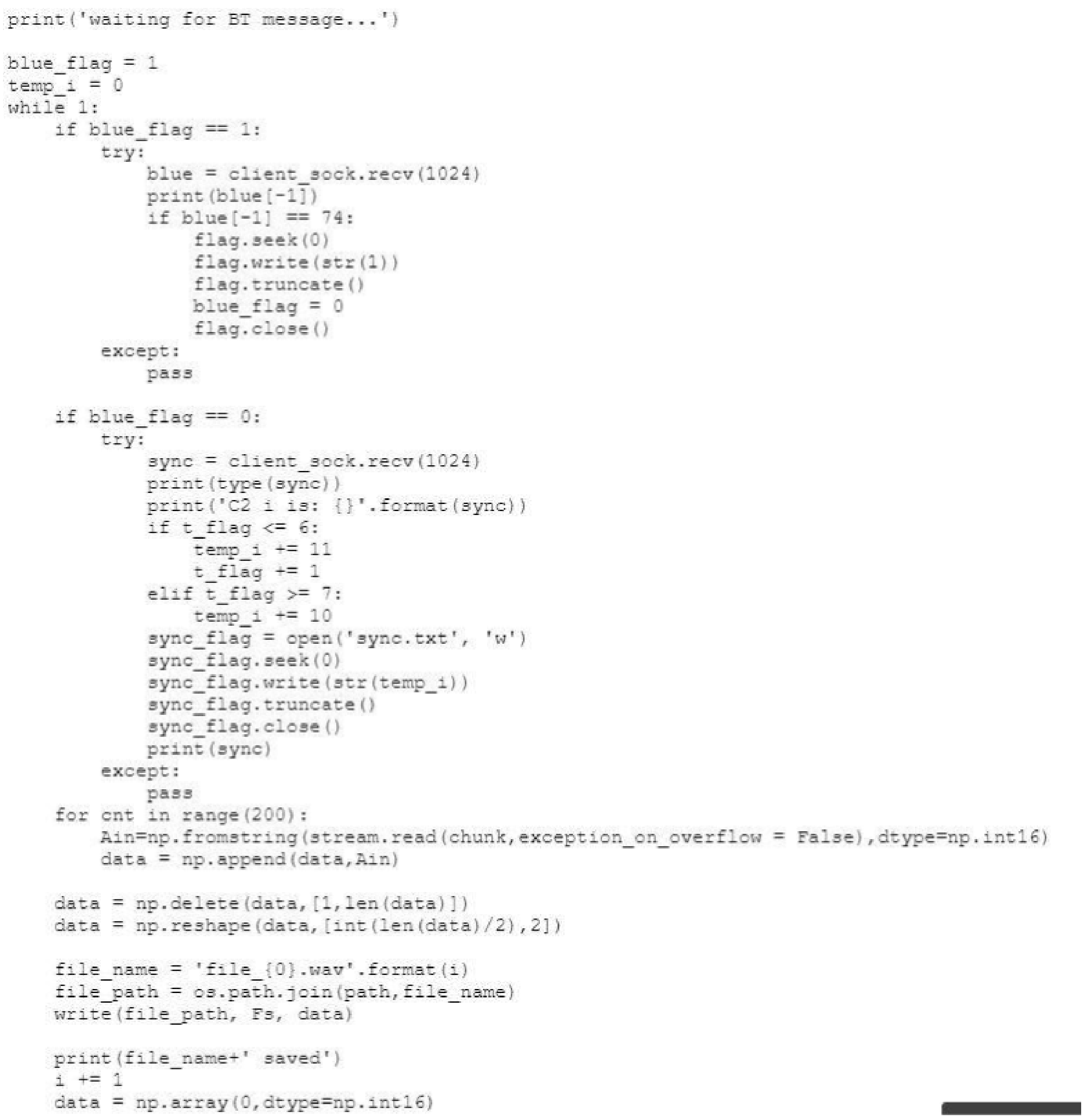
Audio was sampled at 48 kSPS in 128 byte chunks. Then, 200 of these chunks were concatenated into a numpy array. After this, the numpy array was reformatted from a single row vector with alternating left and right channel data values into a 2D array. Then, it was converted to a wav file. and stored into a directory location.



```
import pyaudio
import numpy as np
import os.path
from scipy.io.wavfile import write
from time import sleep
from bluetooth import*

#suppress warnings
from ctypes import*
ERROR_HANDLER_FUNC = CFUNCTYPE(None, c_char_p, c_int, c_char_p, c_int, c_char_p)
def py_error_handler(filename, line, function, err, fmt):
     dummyVar = 1
c_error_handler = ERROR_HANDLER_FUNC(py_error_handler)
asound = cdll.LoadLibrary('libasound.so')
asound.snd_lib_error_set_handler(c_error_handler)

flag = open('flag.txt','w')
flag.seek(0)
flag.write(str(0))
flag.truncate()

sync_flag = open('sync.txt', 'w')
sync_flag.seek(0)
sync_flag.write('0')
sync_flag.truncate()

print('Starting Bluetooth Connection...')
server_sock = BluetoothSocket(RFCOMM)
port = 1
server_sock.bind(('',port))
server_sock.listen(1)
client_sock,address = server_sock.accept()
print('Accepted connection from ',address)
client_sock.settimeout(.01)

path = '/home/pi/Desktop/Senior Deisgn/multitask test/files'

chunk = 128
Fs = 48000
chan = 2
Ain = np.array(0,dtype=np.int16)
data = np.array(0,dtype=np.int16)

p = pyaudio.PyAudio()
stream = p.open(format=pyaudio.paInt16, rate=Fs,channels=chan, input_device_index = 1, input=True, frames_per_buffer=chunk)
i = 0
```

.

Figure 8.1.1 WAT1 First Half of the Sampling and Bluetooth Script:



```
print('waiting for BT message...')

blue_flag = 1
temp_i = 0
while 1:
    if blue_flag == 1:
        try:
            blue = client_sock.recv(1024)
            print(blue[-1])
            if blue[-1] == 74:
                flag.seek(0)
                flag.write(str(1))
                flag.truncate()
                blue_flag = 0
                flag.close()
        except:
            pass

    if blue_flag == 0:
        try:
            sync = client_sock.recv(1024)
            print(type(sync))
            print('C2 i is: {}'.format(sync))
            if t_flag <= 6:
                temp_i += 11
                t_flag += 1
            elif t_flag >= 7:
                temp_i += 10
            sync_flag = open('sync.txt', 'w')
            sync_flag.seek(0)
            sync_flag.write(str(temp_i))
            sync_flag.truncate()
            sync_flag.close()
            print(sync)
        except:
            pass
    for cnt in range(200):
        Ain=np.fromstring(stream.read(chunk,exception_on_overflow = False),dtype=np.int16)
        data = np.append(data,Ain)

    data = np.delete(data,[1,len(data)])
    data = np.reshape(data,[int(len(data)/2),2])

    file_name = 'file_{0}.wav'.format(i)
    file_path = os.path.join(path,file_name)
    write(file_path, Fs, data)

    print(file_name+' saved')
    i += 1
    data = np.array(0,dtype=np.int16)
```

Figure 8.1.2: WAT1 Second Half of Sampling and Bluetooth Script

Module DSP:

Julian Bell was responsible for the DSP block. An LPF and HPF were created using a second order digital Z-transform, discrete time impulse response, and filter coefficients.

Processing narrative for DSP

All of the filtering was handled in one python script:

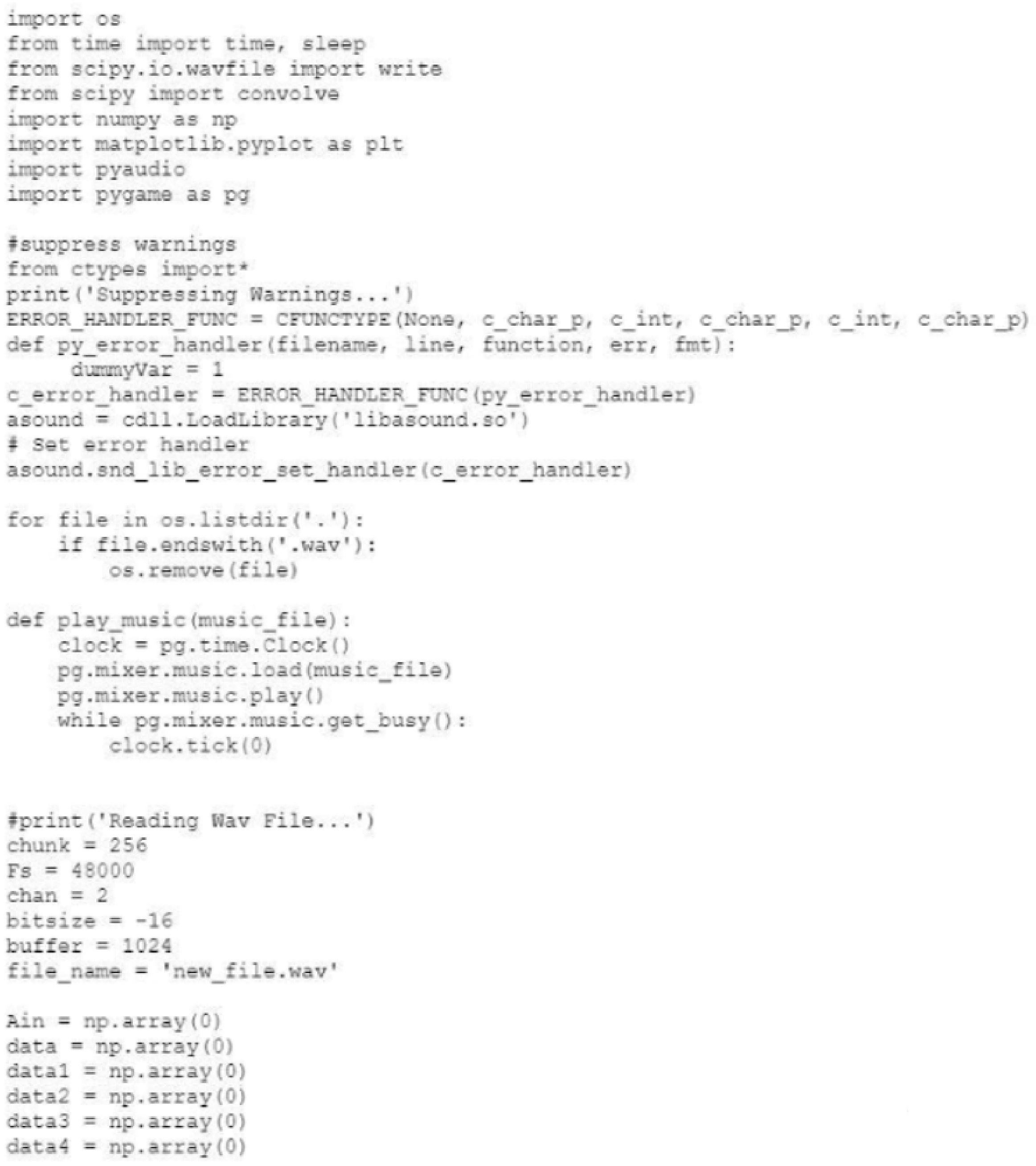
- filters_C1.py

DSP interface description

The DSP received a digital audio signal via I2C and USB from the SSS1629 ADC and outputs to the SSS1629 DAC via I2S.

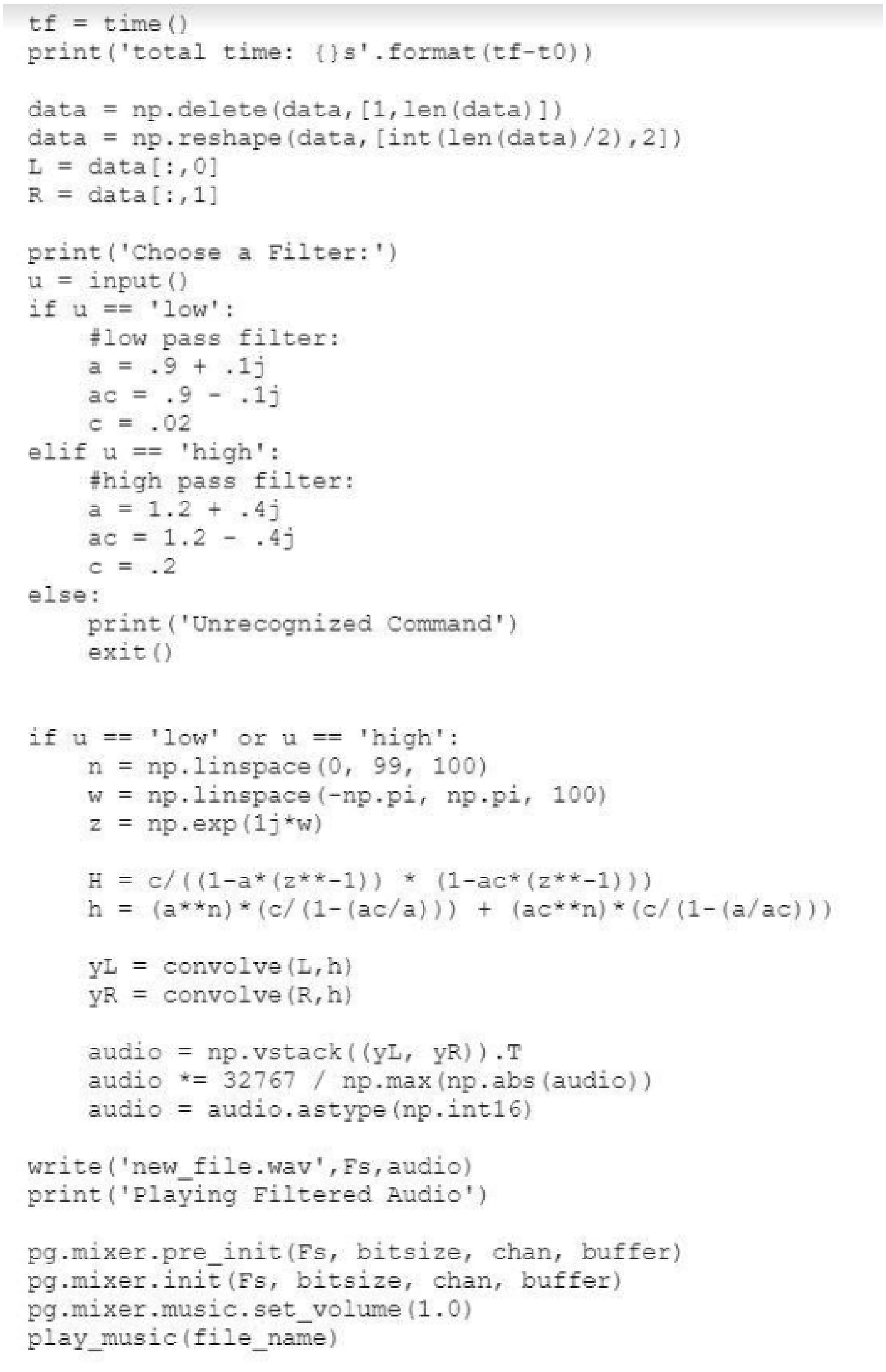
DSP processing details

This script in Figures 8.1.3, 8.1.4, and 8.1.5 used the scipy.signal library to convolute the audio data with the impulse response of the filter and the matplotlib library to plot data. The script initialized all streaming parameters for the pyaudio object and all numpy arrays needed. The filter was created with filter coefficients calculated in Section 4.1. Analog audio was sampled for 10 seconds and played back in real time. The user had a choice to apply a LPF or a HPF. The filter was applied by convoluting the left and right channel data with the discrete time impulse response of the Z transform filter and the audio is played back. Once playback completed, the FFT was taken of the filtered and unfiltered audio channels and plotted on a logarithmic scale of magnitude vs frequency.



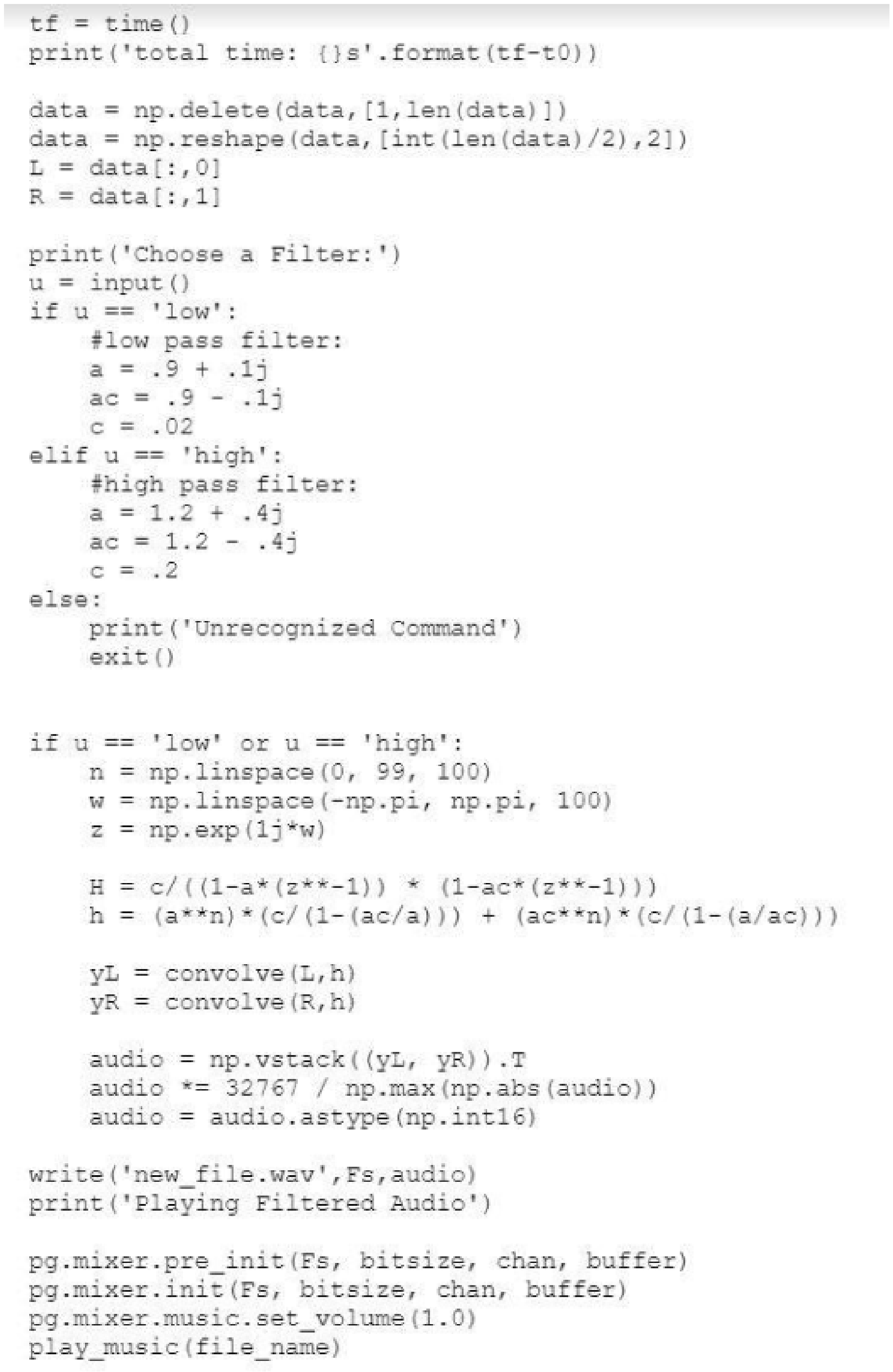
```
import os
from time import time, sleep
from scipy.io.wavfile import write
from scipy import convolve
import numpy as np
import matplotlib.pyplot as plt
import pyaudio
import pygame as pg

#suppress warnings
from ctypes import*
print('Suppressing Warnings...')
ERROR_HANDLER_FUNC = CFUNCTYPE(None, c_char_p, c_int, c_char_p, c_int, c_char_p)
def py_error_handler(filename, line, function, err, fmt):
     dummyVar = 1
c_error_handler = ERROR_HANDLER_FUNC(py_error_handler)
asound = cdll.LoadLibrary('libasound.so')
# Set error handler
asound.snd_lib_error_set_handler(c_error_handler)

for file in os.listdir('.'):
    if file.endswith('.wav'):
        os.remove(file)

def play_music(music_file):
    clock = pg.time.Clock()
    pg.mixer.music.load(music_file)
    pg.mixer.music.play()
    while pg.mixer.music.get_busy():
        clock.tick(0)


#print('Reading Wav File...')
chunk = 256
Fs = 48000
chan = 2
bitsize = -16
buffer = 1024
file_name = 'new_file.wav'

Ain = np.array(0)
data = np.array(0)
data1 = np.array(0)
data2 = np.array(0)
data3 = np.array(0)
data4 = np.array(0)
```

Figure 8.1.3: First part of filters_C1.py script:



```
tf = time()
print('total time: {}s'.format(tf-t0))

data = np.delete(data,[1,len(data)])
data = np.reshape(data,[int(len(data)/2),2])
L = data[:,0]
R = data[:,1]

print('Choose a Filter:')
u = input()
if u == 'low':
    #low pass filter:
    a = .9 + .1j
    ac = .9 - .1j
    c = .02
elif u == 'high':
    #high pass filter:
    a = 1.2 + .4j
    ac = 1.2 - .4j
    c = .2
else:
    print('Unrecognized Command')
    exit()


if u == 'low' or u == 'high':
    n = np.linspace(0, 99, 100)
    w = np.linspace(-np.pi, np.pi, 100)
    z = np.exp(1j*w)

    H = c/((1-a*(z**-1)) * (1-ac*(z**-1)))
    h = (a**n)*(c/(1-(ac/a))) + (ac**n)*(c/(1-(a/ac)))

    yL = convolve(L,h)
    yR = convolve(R,h)

    audio = np.vstack((yL, yR)).T
    audio *= 32767 / np.max(np.abs(audio))
    audio = audio.astype(np.int16)

write('new_file.wav',Fs,audio)
print('Playing Filtered Audio')

pg.mixer.pre_init(Fs, bitsize, chan, buffer)
pg.mixer.init(Fs, bitsize, chan, buffer)
pg.mixer.music.set_volume(1.0)
play_music(file_name)
```

Figure 8.1.4: Second part of filters_C1.py script:



```
tf = time()
print('total time: {}s'.format(tf-t0))

data = np.delete(data,[1,len(data)])
data = np.reshape(data,[int(len(data)/2),2])
L = data[:,0]
R = data[:,1]

print('Choose a Filter:')
u = input()
if u == 'low':
    #low pass filter:
    a = .9 + .1j
    ac = .9 - .1j
    c = .02
elif u == 'high':
    #high pass filter:
    a = 1.2 + .4j
    ac = 1.2 - .4j
    c = .2
else:
    print('Unrecognized Command')
    exit()


if u == 'low' or u == 'high':
    n = np.linspace(0, 99, 100)
    w = np.linspace(-np.pi, np.pi, 100)
    z = np.exp(1j*w)

    H = c/((1-a*(z**-1)) * (1-ac*(z**-1)))
    h = (a**n)*(c/(1-(ac/a))) + (ac**n)*(c/(1-(a/ac)))

    yL = convolve(L,h)
    yR = convolve(R,h)

    audio = np.vstack((yL, yR)).T
    audio *= 32767 / np.max(np.abs(audio))
    audio = audio.astype(np.int16)

write('new_file.wav',Fs,audio)
print('Playing Filtered Audio')

pg.mixer.pre_init(Fs, bitsize, chan, buffer)
pg.mixer.init(Fs, bitsize, chan, buffer)
pg.mixer.music.set_volume(1.0)
play_music(file_name)
```

Figure 8.1.5: Third part of filters_C1.py script:

9. Test Procedure:

Julian Bell decided on the SSS1629 DAC and ADC combo. The ADC is 16 bit, 48k SPS and uses USB and I2C for serial communication. It has a variable sampling rate that goes up to 48k SPS. He performs a test that measures SNR, THD+n, and waveform sampling.

To ensure that sampling continues smoothly without interference from other scripts, Julian Bell created a multi-processing script that runs each script in parallel with each other. This way, there is no latency in the sampling from other functions, such as Wi-Fi or bluetooth, by running in parallel. This way, no audio data is missed.

Julian Bell also performed filtering tests. Since this added too much latency to the live playback and there isn't enough time to fully implement them, he has done filtering separately as a proof of concept. He tests two different second order Z-transform filters, with the intent of making a low pass and high pass filter.

When integrating this with all the other components, Julian Bell found that it added too much latency for the RPi3 to sample, filter, and send the audio. Even when running them in parallel, it still added about 50 ms of delay, which is very noticable when listening to playback. The solution was to run the script separately from the other scripts.

Julian Bell noticed that audio playback would be very choppy. This was fixed when the size of the audio buffer chunk was lowered in order to lower latency. Also, making small arrays of audio and concatenating them later reduced latency compared to concatenating audio into one large array.

Julian Bell noticed that the filters were yielding unreliable results and maybe the coefficients need to be recalculated. Once coefficients were recalculated, the filter behaved more as to be expected and the audio came out properly filtered.

10. Test Results:

Test 1:

Test 1 was for the ADS1115, which did not pass feasibility, so it isn't relevant to the integrated system.

Test 2:

1. The results of test two remained the same as experiment two. The ADC was able to reconstruct signals ranging from 20 Hz to 20 kHz, the full audible range of human hearing.
2. The results were exactly as expected; with a sampling rate of 48k SPS, which is more than the Nyquist rate of audio, the analog waveform is able to be fully reconstructed.
3. The full audible range of frequencies were able to be reconstructed. The SNR and THD+n matched what was expected from the datasheet.
4. No corrective actions needed

Test 3:

1. The audio filtering did not match what was expected at first. The filtered frequencies did not match until filter coefficients were recalculated.
2. The results were different from what was expected, as the high pass filter would filter out low frequencies and vice versa.

3. The poles and zeros of the high pass filter were within the unit circle on the Z plane and the the poles and zeros were out of the unit circle for the low pass filter
4. Calculating filter coefficients fixed this issue

11. Conclusion:

Julian Bell designed the audio sampling and audio filtering. The sampling worked perfectly well. Audio was successfully sampled at 48k SPS in 128 byte chunks. 200 chunks were concatenated and converted to a .wav file which is half a second of audio data and 100 kB big. Though the filters were unable to be integrated in the whole system, they work perfectly well as a system by itself. Audio is able to be sampled and played back and filter is able to be applied to the original data.

Julian Bell learned a lot about how sampling affects the digital reconstruction of the analog waveform. He learned how the size of the data affects the processing time. He learned about multi-processing and running Python scripts in parallel to make the system more efficient. He also learned about digital filters and how to use Z transform filter coefficients to create both low pass and high pass filters. He learned how the frequency response of the filter affects the filtered audio and how the poles and zeros can be used to predict aspects of the filter.

Julian Bell also learned how to read datasheets efficiently and pick out the useful information inside.

12. Acknowledgments:

I would like to acknowledge my design teammates Duane Buchholz, Ethan Kim and Robert Jimenez for working on their own subsystems and helping to integrate the project together. Duane Buccholz tested and implemented the DAC and designed the Bluetooth synchronization. Ethan Kim tested and implemented the Wi-Fi communication and the Wi-Fi server. Robert Jimenez tested Bluetooth functionality and designed a boost converter for powering the project. Without them, I would not have been able to complete this design project.

I would like to acknowledge my faculty mentor Professor Roman Chomko. He helped guide our team through the design process. He provided useful information when we ran into testing issues. He also made sure that we were continuing to progress each and every week. This was helpful as we did not fall behind during the process. I would like to thank the honors program for the opportunities and advantages it has provided me throughout my undergraduate education. Finally, I would like to thank Professor Tofigh Heidarzadeh for teaching me the technical writing style and helping to improve my technical writing ability.

References:

- ADS1115 datasheet: http://www.ti.com/lit/ds/symlink/ads1114.pdf

- SSS1629 datasheet: http://www.3system.com.tw/upload/product/46ee1aa50fd3eabf0a799590f26cdf46.pdf

- Audio Playback:

  "Play Sound in Python." *Play Sound in Python - Python Tutorial* , Pythonbasics.org/Python-play-sound/.

  https://Pythonbasics.org/Python-play-sound/

- Audio Sampling:

  Real Python. "Playing and Recording Sound in Python." *Real Python*, Real Python, 23 Apr. 2019, realPython.com/playing-and-recording-sound-Python/.
  https://realPython.com/playing-and-recording-sound-Python/

- Digital filters:
  http://www.eas.uccs.edu/~mwickert/ece5650/notes/N5650_9.pdf

Appendices:

Appendix A: Parts List with Prices:

Part name	Unit price	Unit price	Total price + Tax (CA) + Shipping
Raspberry Pi 3B+	$35.00	$35.00	$76.30
USB ADC/DAC	$9.95	$9.95	$21.64
3.3 ft. 3.5mm AUX Cords (2 pack)	$6.99	$6.99	$7.62
RPi Power Supply	$9.99	$9.99	$21.72
Transistors (10 pack): BJT MJE3055T NPN	$8.49	$8.49	$9.15
Inductor 100uH	$6.99	$6.99	$7.53
Capacitor (10 pack): 4700uF	$8.99	$8.99	$9.69
Schottky Diode (10 pack)	$5.38	$5.38	$5.80
Insulated Wire 24AWG	$7.95	$7.95	$8.57
Lithium Ion Batteries (4 pack)	$19.96	$19.96	$24.77
Battery Holder (10 pack)	$6.58	$6.58	$7.09
Micro USB Male Port Connector (10 pack)	$6.99	$6.99	$7.53
Boost Regulator MC34063A	$6.95	$6.95	$14.98
32 GB micro-SD Card	$8.99	$8.99	$18.52

Total : $240.92 ($120.46 per WAT module)

Appendix B: Equipment List:

- Oscilloscope,
- Power Supply,
- Monitors,
- USB cables,
- HDMI Cables,
- Variable AC Waveform Generators

Appendix C: Python Libraries:

- matplotlib: https://matplotlib.org/
- numpy: https://numpy.org/
- pyaudio: https://pypi.org/project/PyAudio/
- pygame: https://www.pygame.org/
- scipy: https://scipy.org/scipylib/