DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATION ENGINEERING
UNIVERSITY OF MORATUWA

GROUP PROJECT REPORT
EN1093-LABORATORY PRACTICES

Simple Voice Recorder

Authors:	*Index Number:*
Munasinghe B.S.V.W.	190397E
Munasinghe M.M.R.H.	190399L
Nifla M.N.F.	190413D
Nushath M.N.M.	190423H

This is submitted as a partial fulfilment for the module EN1093.

August 16, 2021

Abstract

The proposed project was to design a simple voice recorder to do the basics of a voice recorder such as record, save, and playback. Also implementing some spectral changes is preferred.
The inbuilt ADC pin of the ATmega328p microcontroller gets the voice signal using the MAX9841 microphone module. Then it takes samples, quantizes, and saves them as a WAVE file according to the sampling theorem. We use five keys to customize the operation of the device. We use an OLED display is to produce a GUI. DAC output is generated by the PWM technique that may not the best because of the noise. But a preferred solution when considering the constraints because using a DAC module will need more power, space, and pins in the microcontroller. The two main spectral changes we implement were frequency scaling and signal enhancing. We performed frequency scaling by downsampling without much complexity. Signal enhancement was done by first filtering the signal with different filters and then creating separate WAV files using convolution in the frequency domain. We did the initial prototyping using Arduino Uno. Later we customize the Arduino functions to use less memory and be compatible with all the expectations of the device. Finally, we implement all the functionalities in the ATmega328p microcontroller that fasten the performance of the device. We use an RC lowpass filter followed by an amplifier circuit to enhance the output signal.

Contents

1 Introduction

Simple Voice Recorder is a simple tool to record multiple voice tracks and then play them back. In day-to-day life, this device helps people in many ways to listen to music, to record their voice, etc. The purpose of this project is to create a Simple Voice Recorder using theories of signal processing and information theory.
Voice Recorder deals with Analog and Digital Signals. So, the Recorder must convert Analog signal to Digital signal using ADC (Analog to Digital Converter) when recording a voice. And when play it back, the Recorder must convert the Digital signal to an Analog signal using DAC (Digital to Analog Converter). In this project, 8bit PCM is used for ADC and Fast PWM is used for DAC.
The main objectives of this project are to record and save multiple voice recordings, selectively playback the recordings, and do minor adjustments to its spectrum during playback. text..

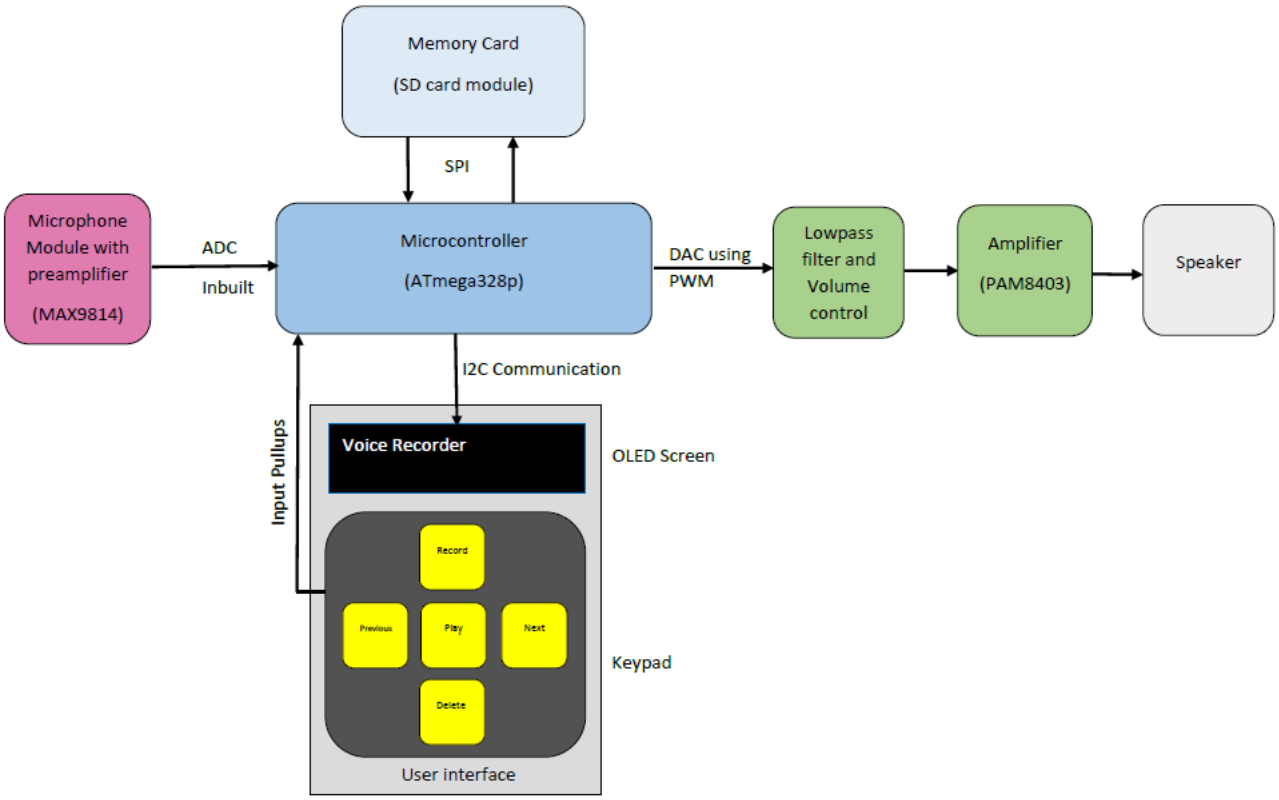


Figure 1: Block Diagram

2 Method

2.1 Audio Input

Audio input
MAX9841 module is used to get the input voices for this device. It offers a cascade amplification of about 60dB, which equals a gain of 1000. It has an automatic gain control feature, which means it will amplify quiet noises and suppress loud noises. So perfect for this application.

ADC
ADC converts a continuous-time, continuous-amplitude analog signal into a discrete-time, discrete-amplitude digital signal, which is often represented by 0s and 1s. By limiting the input signal's permissible bandwidth, the conversion entails sampling, quantization, and encoding of the input signal. When we convert an analog signal to a digital signal, we always lose data. Using an infinite resolution and a high sample rate ADC, we can minimize this loss. To convert the analog data loss lessly, it requires an infinite resolution level and an infinite sampling rate ADC, which is impossible to archive, so the Nyquist–Shannon sampling theorem is applied and the sampling rate is chosen.

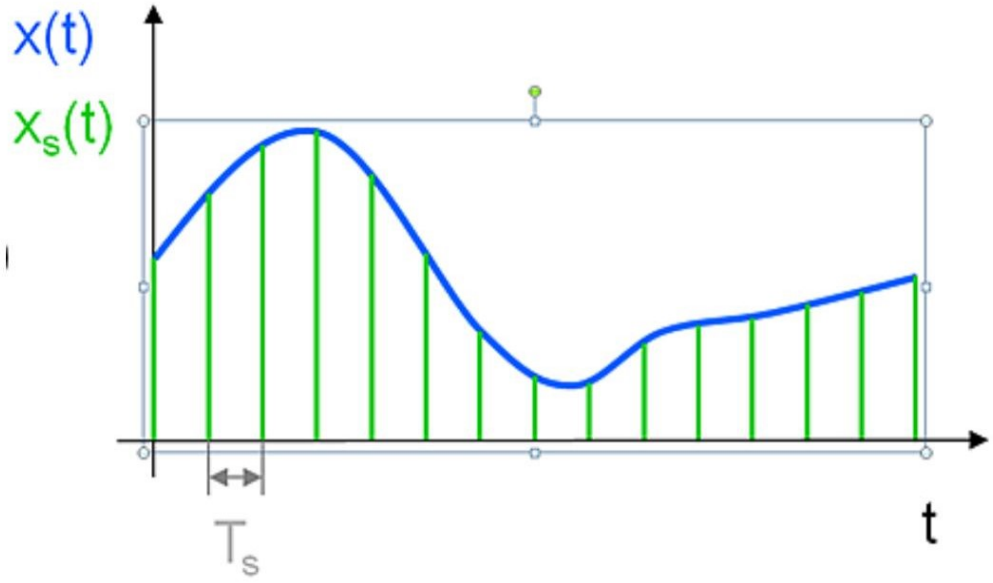


Figure 2: Sampling

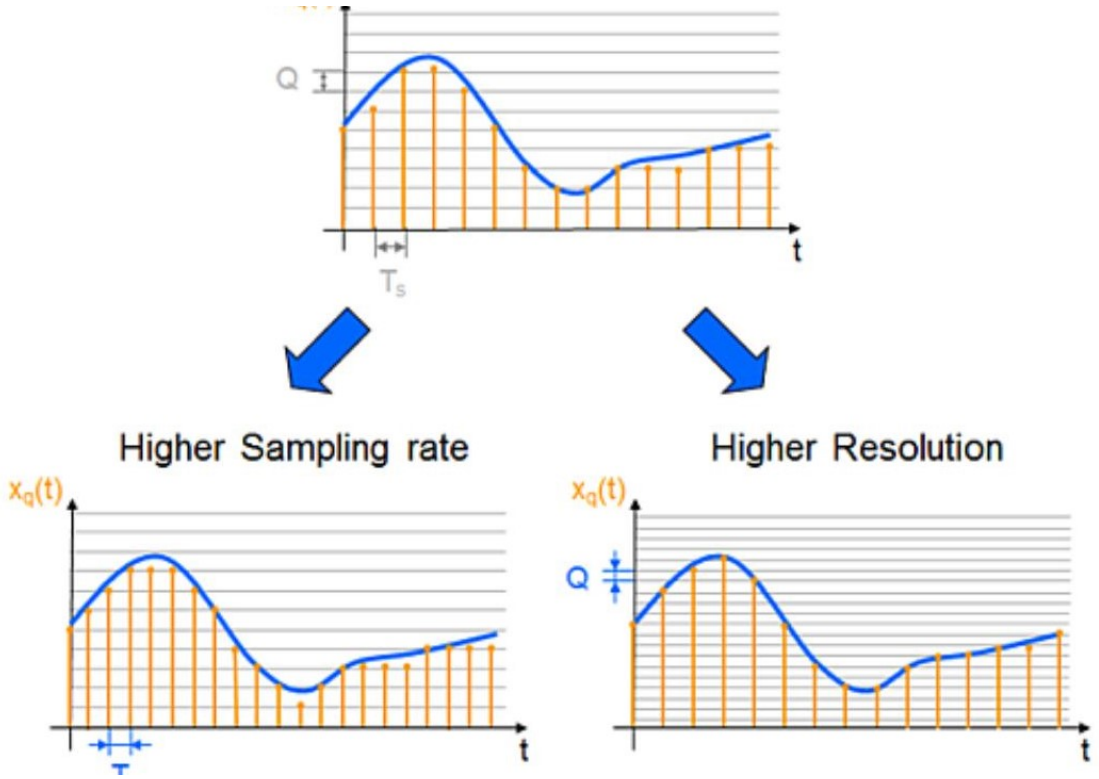


Figure 3: Sampling

$$T_s = Sampling\_Period = \frac{1}{Sampling\_Frequency}$$

Here reference voltage is 5V and 8bit Resolution is used. Hence, quantized voltage value is calculated by,

$$ADC = \frac{V_{IN} \cdot 255}{V_{REF}}$$

As This device is only dealing with Voices, we can limit the input signal bandwidth to 4kHz. To recreate the input signal, the sampling frequency must be more than 8kHz based on Nyquist–Shannon sampling theorem.
The Analog to Digital Conversation is done here by using inbuilt ATmega328p ADC. ATmega328p has an external clock frequency of 16MHz. By default, the successive approximation circuitry requires an input clock frequency between 50kHz and 200kHz to get a maximum resolution for 10bit. If 8bit resolution is needed, then the input clock frequency to the ADC should be higher than 200kHz to get a higher sample rate. By using pre-scaler, we can change the input clock frequency. Here 16 is set as the pre-scaling value to archive an input clock frequency of about 1MHz. But overall sampling rate is not 1MHz. Because it will take some time for the code to execute the mathematical function and store the sample value to SD card. So, we can archive an overall Sampling rate at about 12.5kHz which is more than 8kHz. So input signal can be recreated at the output.

2.2 Store the Recordings

After converting the analog audio signal, the samples should be stored in memory for future access. To achieve this functionality a proper audio format needed to be selected. The choice was WAVE file format. This format is a raw data format, therefore data can be written and accessed easily within the program. There is a certain standard format for this file format. So, the method was to follow that format and make the WAVE files. The advantage of using a standard data format is, we can play the recordings on any platform, not only on our device's program.

The WAVE file format is a subset of Microsoft's RIFF specification for multimedia file storing. A RIFF file starts with a header. After the header data chunks can be seen. A WAVE file is a RIFF file with a single "WAVE" chunk which consists of two sub-chunks a "fmt " chunk specifying the data format and a "data" chunk containing the actual sample data.
The part which must pay more attention to is the endian format of the binary data we are writing to the file. If the endian format is wrong, the file format will not be identified by other players in devices.

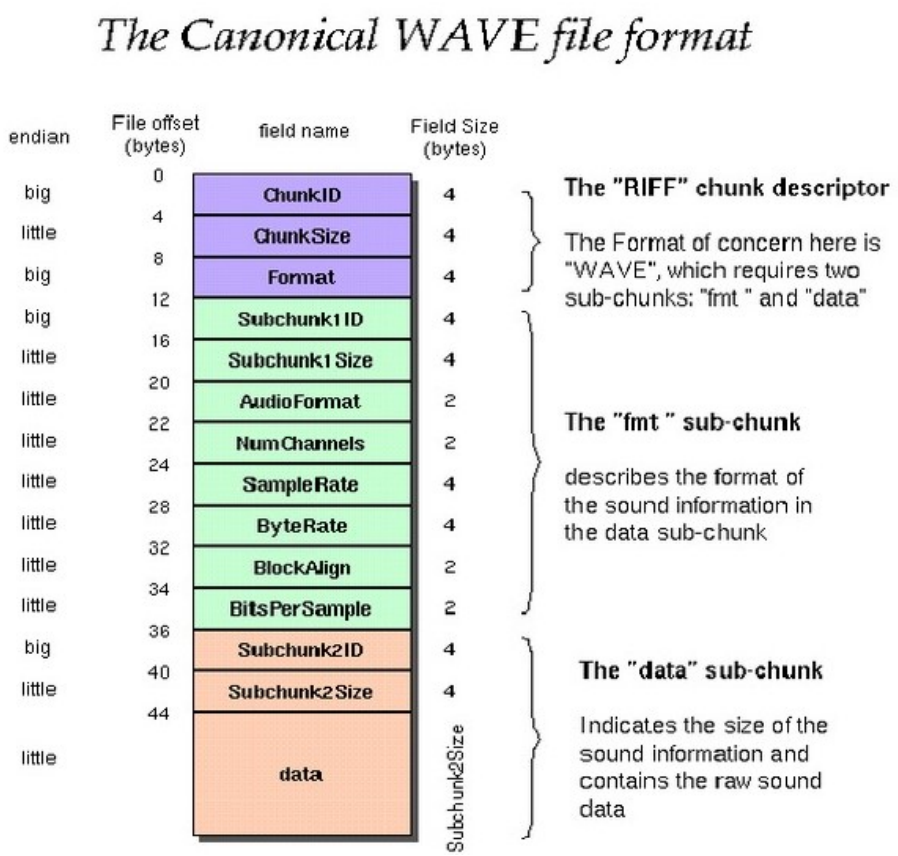


Figure 4: Format of a WAVE file
Source : Microsoft WAVE soundfile format (sapp.org)

Audio file attributes in this application:

- Sample rate – 12.5kHz
- 8bit-PCM format
- Mono Channel

When creating the wave file through the program three main steps were followed.

1. Make the header file including the main attributes of the audio (chunk size slots kept empty)
2. Take the inputs to the microcontroller and write sampled values as bytes to the data chunk
    - Input values are taken by inbuilt ADC.
    - Since we use 8bit PCM audio one sample has the size of one byte. So, writing the sample values is easy, only need to write as bytes.
3. Fill the chunk size slots

```
void makeWaveFile(File sFile) {
  /*
     This function creates the wave header file required
     All bytes should be in little endian format, except String values
  */

  sFile.write((uint8_t*)"RIFF    WAVEfmt ", 16);//Starting bytes of the wave header file
  uint8_t chunk[] = {16, 0, 0, 0, 1, 0, 1, 0, lower_Byte(sampleRate), higher_Byte(sampleRate)};
  /*
     chunk[]
     first 4 bytes: size of  previous data chunk
     next 2 bytes: Audio format (1 - PCM)
     next 2 byte: No of channels (Mono = 1, Stereo = 2) (in our case 1)
     last two are the first two bytes of sample rate
  */
  sFile.write((uint8_t*)chunk, 10);

  chunk[0] = 0; chunk[1] = 0; //end of sample rate bytes

  //byteRate = (sampleRate/8)*monoStereo*8;
  chunk[2] = lower_Byte(byteRate); chunk[3] = higher_Byte(byteRate); chunk[4] = 0; chunk[5] = 0; // byteRate

  //byte blockAlign = monoStereo * (bps/8);
  //this is always equal to 1 in 8bit PCM mono channel
  chunk[6] = 1; chunk[7] = 0; //BlockAlign

  chunk[8] = 8; chunk[9] = 0; //bits per sample

  sFile.write((uint8_t*)chunk, 10);
  sFile.write((uint8_t*)"data    ", 8);

}
```

Figure 5: Syntax for the wave file header creation

```
void finalizeWave(File sFile) {
  /*
     This function finalizes the wave file
  */
  unsigned long fSize = sFile.size();

  fSize -= 8;
  sFile.seek(4);
  uint8_t chunk2[4] = {lower_Byte(fSize), higher_Byte(fSize), fSize >> 16, fSize >> 24};
  sFile.write(chunk2, 4);//Writing chunk size to 5 - 8 bytes in wave file

  sFile.seek(40);
  fSize -= 36 ;
  chunk2[0] = lower_Byte(fSize); chunk2[1] = higher_Byte(fSize); chunk2[2] = fSize >> 16; chunk2[3] = fSize >> 24;
  sFile.write((uint8_t*)chunk2, 4);//Writing number of samples to 41-44 bytes in wave file
}
```

Figure 6: Syntax for the wave file finalizing

2.3 Signal Processing

2.3.1 Spectral Changes in Audio Signal with MATLAB Demonstration

Frequency Scaling

The process of changing the speed or duration of an audio signal is called Frequency Scaling. In MATLAB Implementation, we changed the Sampling Frequency of the Audio file. Instead of using the original sampling frequency, we used a scaled sampling frequency. If we use twice sampling frequency to sample audio files it will increase the speed of playing the track twice. For plotting graphs, we used the following Time and Frequency scaling property of Fourier Transform.

$$x(at) \xrightarrow{\mathcal{F}} \frac{1}{|a|} \cdot X(\frac{j\omega}{a})$$

Frequency and Time spectrum graphs are shown in Figure 7 and MATLAB code is shown in Figure 8.

Signal Enhancing

An increase or decrease in amplitudes of frequency components in a particular frequency region is known as Signal Enhancing. In this project, we categorized Signal Enhancing into three parts. Namely,

- Low Pass Enhancing
- Band Pass Enhancing
- High Pass Enhancing

Filtering

Using the Filter Designer app, we have obtained low, band, and high pass filter coefficients. As shown in Figure 9, we used the FIR Hamming Window filter and suitable filter order in MATLAB implementation. Then we convolve the audio signal with filter coefficients to get the filtered signal. The algorithm we have used for low pass filtering (high pass and bandpass algorithms are the same as high pass, the only difference is the value of coefficients) is shown in Figure 10.

Low Pass Enhancing

Here we used a low pass filter to acquire selected frequency components in the low pass region then multiply it by a necessary constant to enhance amplitudes in that region. Then we used a high pass filter to get frequency components except for low pass region frequency components. Add frequency components obtained from high pass filter and enhanced frequency components obtained from low pass filter. Simulated graphs are shown in Figure 11 and Algorithm is shown in Figure 15.

Band Pass Enhancing

Here we use a bandpass filter to acquire selected frequency components in the bandpass region then

multiply it by a necessary constant to enhance amplitudes in that region. Then we used a high pass filter to get higher frequency components except for bandpass region frequency components. Also, we used a low pass filter to get lower frequency components except for bandpass region frequency components. Add frequency components obtained from high pass filter and low pass filter and enhanced frequency components obtained from bandpass filter. Simulated graphs are shown in Figure 12 and Algorithm is shown in Figure 15.

High Pass Enhancing

Here we used a high pass filter to acquire selected frequency components in the high pass region then multiply it by a necessary constant to enhance amplitudes in that region. Then we used a low pass filter to get frequency components except for high pass region frequency components. Add frequency components obtained from low pass filter and enhanced frequency components obtained from high pass filter. Simulated graphs are shown in Figure 13 and Algorithm is shown in Figure 15.

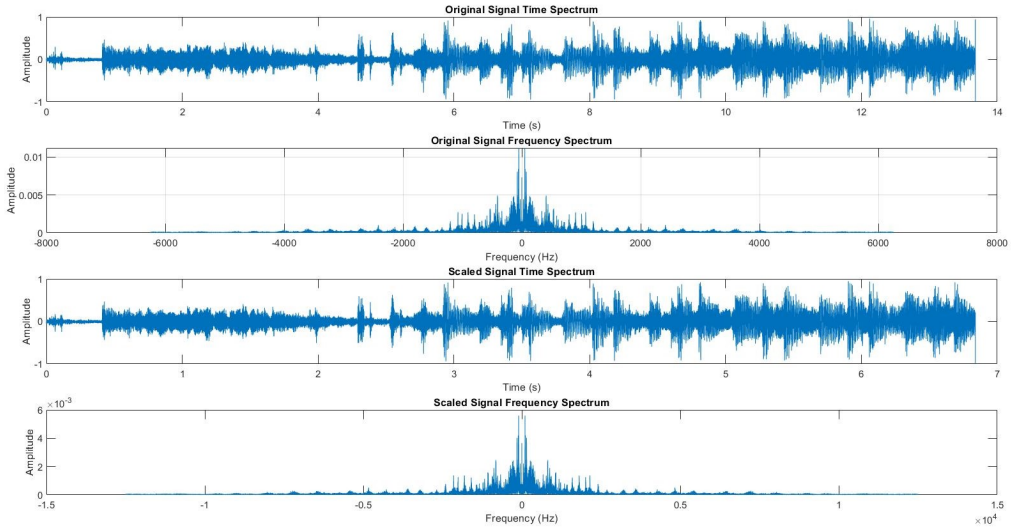


Figure 7: Spectrum and signals of frequency scaling

```
[Signal,Fs] = audioread('Sound.wav');
t = [0:1/Fs:(length(Signal)-1)/Fs];

subplot(4, 1, 1);
plot(t, Signal)%time domain plot of original signal
xlabel("Time (s)");
ylabel("Amplitude");
title("Original Signal Time Spectrum");
subplot(4, 1, 2);
Show_Frequency(Signal, length(Signal), Fs, "Original Signal Frequency Spectrum");%
frequency domain plot of original signal

a = input("Enter Scaling Coefficient: ");
at = linspace(0, (length(Signal)-1)/(a*Fs), length(Signal));
fft_coef = abs(fft(Signal)/length(Signal))/abs(a);
f_a = linspace(-length(Signal)/2,length(Signal)/2-1,length(Signal))*Fs*abs(a)/length
(Signal);

subplot(4, 1, 3);
plot(at, Signal);%time domain plot of scaled signal
xlabel("Time (s)");
ylabel("Amplitude");
title("Scaled Signal Time Spectrum");
subplot(4, 1, 4);
plot(f_a, fftshift(fft_coef));%frequency domain plot of scaled signal
xlabel("Frequency (Hz)");
ylabel("Amplitude");
title("Scaled Signal Frequency Spectrum");

player = audioplayer(Signal,a*Fs);%play edited .wav file
play(player)
```

Figure 8: Algorithm for frequency scaling

Frequency Shifting

Move the frequency spectrum to a different frequency is known as Frequency shifting. Modulation means multiplying a signal with a complex exponential. In the Fourier Transform, modulating a signal in the time domain corresponds to shifting it in the frequency domain. As such, it is the counterpart of shifting in the time domain (which corresponds to modulation in the frequency domain). Mathematically, we have

$$e^{j2\pi f_0 t} \cdot x(t) \xrightarrow{\mathcal{F}} X(f - f_0)$$

Since audio signals have negative frequency components, we cannot use this formula to do frequency shifting. Therefore, we used the cos function instead of the complex exponential function. So, we achieved frequency shifting in an audio signal. Mathematically, we have

$$\cos(2\pi f_0 t) \cdot x(t) \xrightarrow{\mathcal{F}} \frac{1}{2}(X(f - f_0) + X(f + f_0))$$

After multiplying with the Cos function with the audio signal we have to send the shifted signal through a high pass filter to remove unnecessary frequency components to get shifted signal of the original audio signal.
The frequency shifting graph is shown in Figure 14 and MATLAB algorithm is shown in Figure 16.

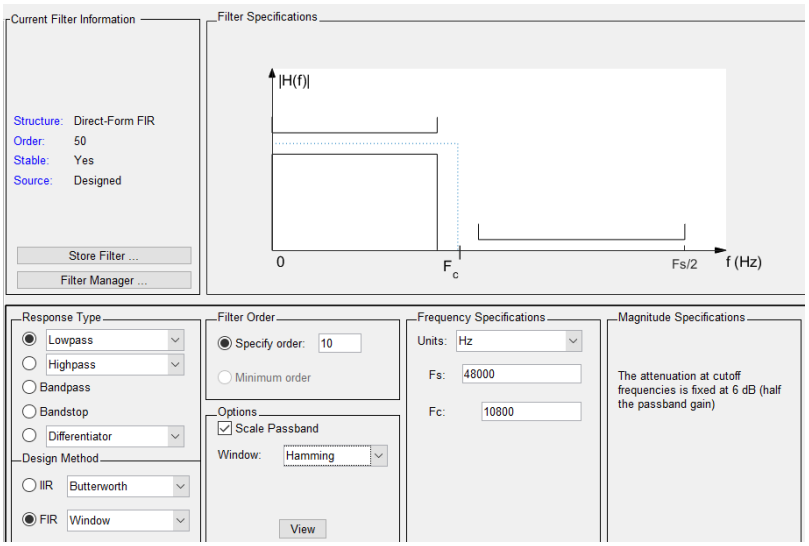


Figure 9: App for MATLAB filterDesigner

```
function [Y] = Low_Pass_Signal_Convol(Original_Signal)
    H = [0.011558612939385695325689162871185544645
0.026912718422614351904131879678061523009
0.068957775242509386504075052926054922864
0.124811024018249089317933453457953874022
0.172306936575946478829024499646038748324
0.190905865602590069096677893867308739573
0.172306936575946478829024499646038748324
0.124811024018249089317933453457953874022
0.068957775242509386504075052926054922864
0.026912718422614351904131879678061523009
0.011558612939385695325689162871185544645];

    %Convolution Algorithm
    X = Original_Signal;
    m = size(X);
    m = m(1,1);
    n = size(H);
    n = n(1,1);
    Y = zeros(m,1);
    for i = n+1:m
        Y(i)=0;
        for j = 1:n
            Y(i) = Y(i) + X(i-j)*H(j);
        end
    end
end
```

Figure 10: Algorithm for low pass filtering

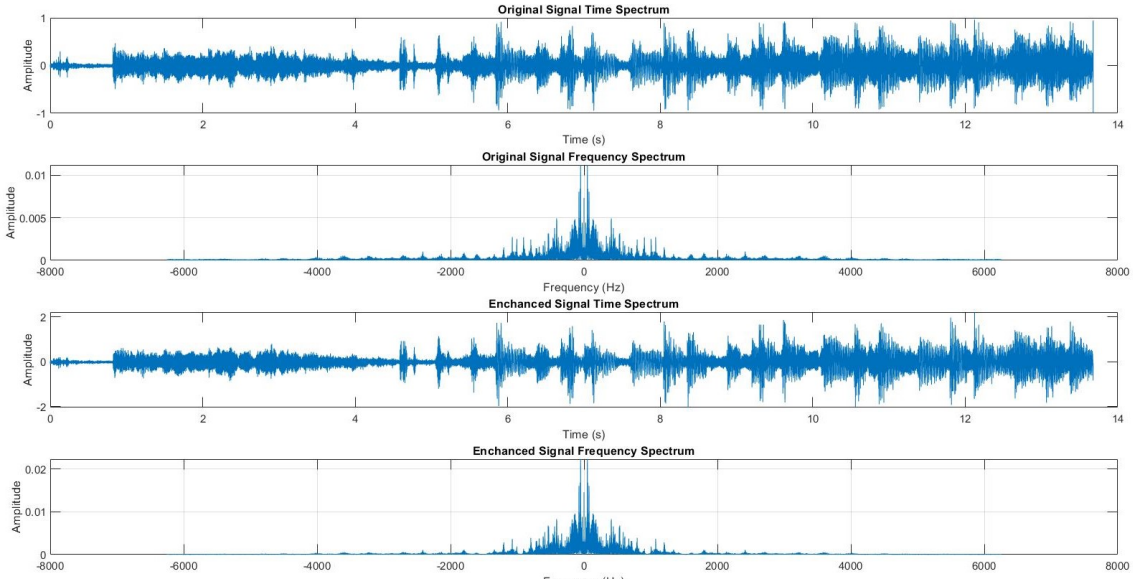


Figure 11: Low pass signal enhancing

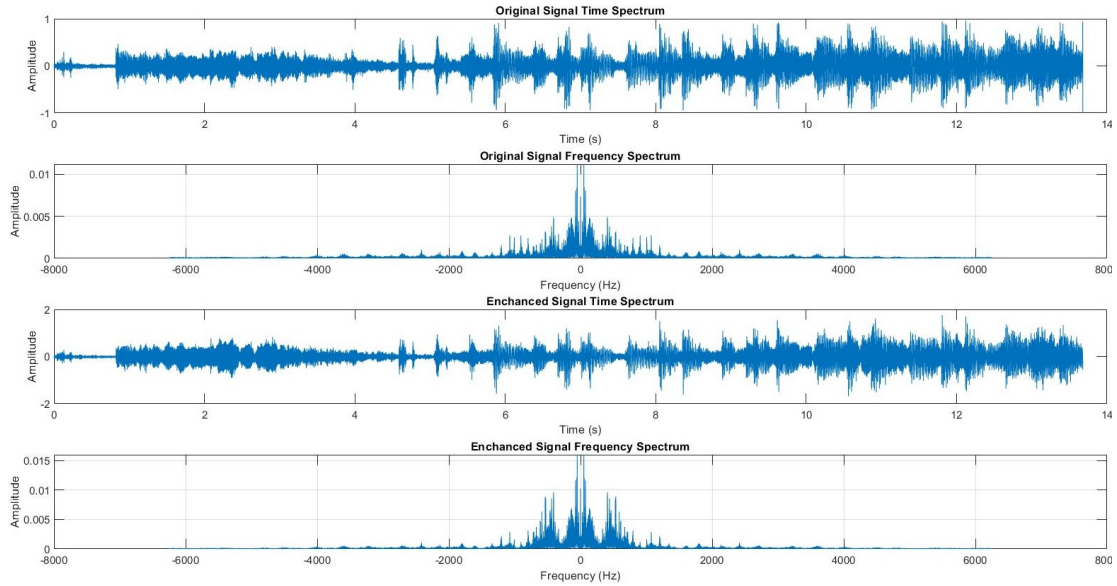


Figure 12: Band pass signal enhancing

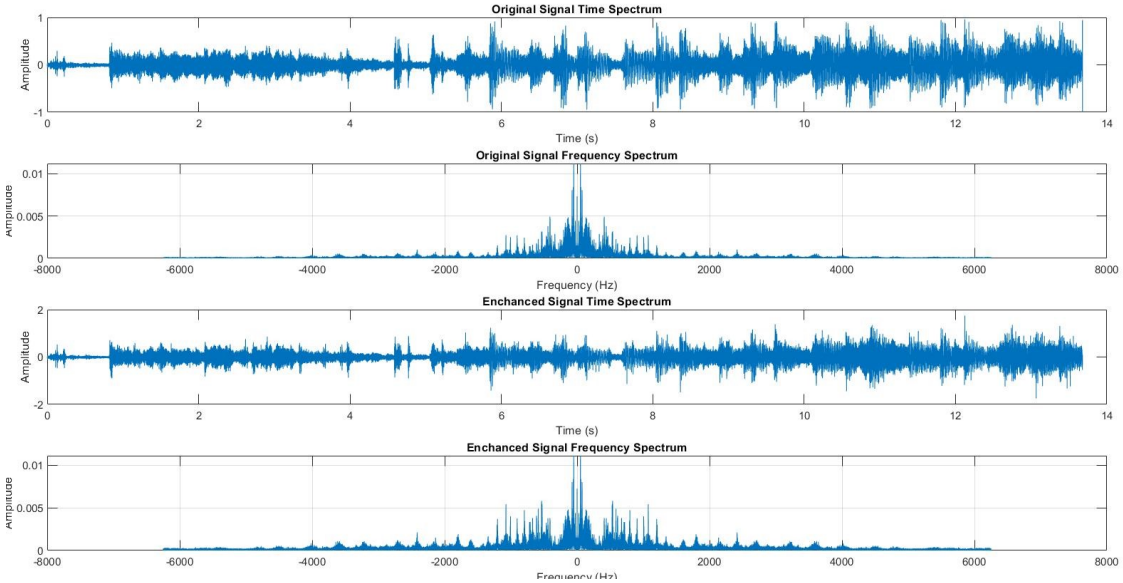


Figure 13: High pass signal enhancing

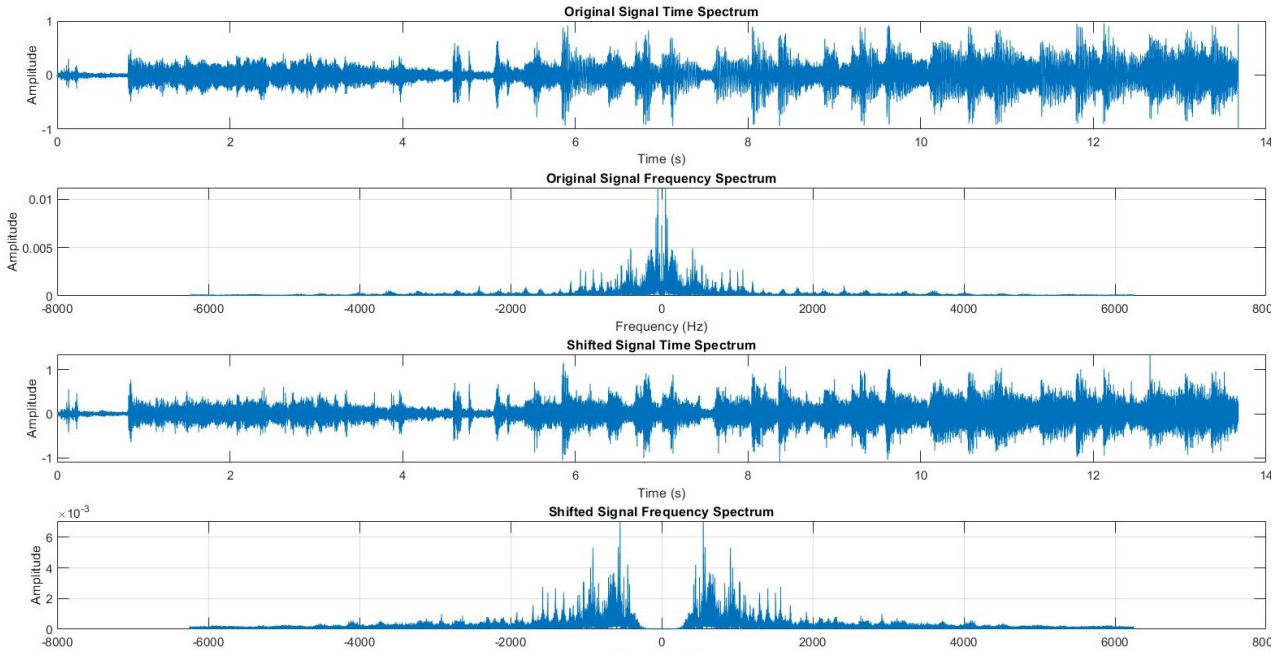


Figure 14: Spectrum of frequency shifting

```
[Signal,Fs] = audioread('Sound.wav');
t = [0:1/Fs:(length(Signal)-1)/Fs];

subplot(4, 1, 1);
plot(t, Signal)%time domain plot of original signal
xlabel("Time (s)");
ylabel("Amplitude");
title("Original Signal Time Spectrum");
subplot(4, 1, 2);
Show_Frequency(Signal, length(Signal), Fs, "Original Signal Frequency Spectrum");%↙
frequency domain plot of original signal

Typ_Fre = input("Type of filter(High_Pass(H-500)/Band_Pass(B-300-500)/Low_Pass(L-↙
300)): ",'s');
Enchance_coeff = input("Input Enchancing Coefficient: ");
%enchancing components of the signal
if Typ_Fre == 'H'
    Signal_high = High_Pass_Signal_Convol(Signal);
    Signal_high = Signal_high.*Enchance_coeff;
    Signal_low = Low_Pass_Signal_Convol(Signal);
    New_Signal = Signal_high + Signal_low;
elseif Typ_Fre == 'B'
    Signal_high = High_Pass_Signal_Convol(Signal);
    Signal_band = Band_Pass_Signal_Convol(Signal);
    Signal_band = Signal_band.*Enchance_coeff;
    Signal_low = Low_Pass_Signal_Convol(Signal);
    New_Signal = Signal_high + Signal_band + Signal_low;
else
    Signal_high = High_Pass_Signal_Convol(Signal);
    Signal_low = Low_Pass_Signal_Convol(Signal);
    Signal_low = Signal_low.*Enchance_coeff;
    New_Signal = Signal_high + Signal_low;
end

subplot(4, 1, 3);
plot(t, New_Signal)%time domain plot of enchanced signal
xlabel("Time (s)");
ylabel("Amplitude");
title("Enchanced Signal Time Spectrum");
subplot(4, 1, 4);
Show_Frequency(New_Signal, length(New_Signal), Fs, "Enchanced Signal Frequency↙
Spectrum");%frequency domain plot of enchanced signal

player = audioplayer(New_Signal,Fs);%play edited .wav file
play(player)
```

Figure 15: MATLAB algorithm for signal enhancing

```
[Signal,Fs] = audioread('Sound.wav');
t = [0:1/Fs:(length(Signal)-1)/Fs];

subplot(4, 1, 1);
plot(t, Signal);%time domain plot of original signal
xlabel("Time (s)");
ylabel("Amplitude");
title("Original Signal Time Spectrum");
subplot(4, 1, 2);
Show_Frequency(Signal, length(Signal), Fs, "Original Signal Frequency Spectrum");%
frequency domain plot of original signal

frequency_shift = 500;
w0 = 2*pi*frequency_shift;
t = t';
modulated_signal = cos(w0*t).*Signal;
shifted_signal = 2*High_Pass_Signal_Convol(modulated_signal);

subplot(4, 1, 3);
plot(t, shifted_signal);%time domain plot of shifted signal
xlabel("Time (s)");
ylabel("Amplitude");
title("Shifted Signal Time Spectrum");
subplot(4, 1, 4);
Show_Frequency(shifted_signal, length(shifted_signal), Fs, "Shifted Signal Frequency
Spectrum");%frequency domain plot of shifted signal

player = audioplayer(shifted_signal,Fs);%play edited .wav file
play(player)
```

Figure 16: MATLAB algorithm for frequency shifting

2.3.2 AVR Implementation

Frequency Scaling
As in the theory, frequency scaling cannot be done easily in the microcontroller because the microcontroller has a limited speed. But as a technique, we can use the downsampling method to achieve such modification. Some information will be lost from this method, but when using limited resources this is a working solution. So downsampling is used to achieve frequency scaling.

Signal Enhancing
As planned the enhancements were done by filtering. To achieve this in the microcontroller program a certain method needs to be carried. Since this process is a digital filtering method, the main mathematical operation to be used is convolution. To do the calculations there are some important aspects to consider.

- Samples are integer values in the range 0-255 (8bit-PCM). There is a DC offset in these signal values because the microcontroller only accepts positive voltages. So, before the calculations, these sampled values should be taken without the DC offset. The signal is given a DC offset of 2.5V, Therefore the corresponding PCM value of 127 should be deducted before calculating. And after the calculation, that DC offset value should be added to the result to get our output samples in the range 0-255.

  Example:
  Sample Value = 139
  Dc offset = 2.5V
  Corresponding quantized value for DC offset = 127
  The value taken for the calculation = 139 – 127 = 12

- Order of the filter

  * A single sample of the output signal is taken by multiplying the overlapped samples and filter coefficients then by adding them. Therefore, filter length should be minimum as possible to make the calculations faster.

  Example:
  Highpass Filter coefficients used in the application (Kaiser window, beta = 1.84, order = 10)
  Filter Structure: Direct-Form FIR
  Filter Length: 11

  -0.018186273607202034507945853647470357828
  -0.046497845276318477969468290211807470769
  -0.083354649048323320448261597448436077684
  -0.120678014680785372636506735943839885294
  -0.148555726582558306203551978796895127743
  0.834158421438040531548097078484715893865
  -0.148555726582558306203551978796895127743
  -0.120678014680785372636506735943839885294
  -0.083354649048323320448261597448436077684
  -0.046497845276318477969468290211807470769
  -0.018186273607202034507945853647470357828

- Storing the filter coefficients

  * The filter coefficients that were taken by the Matlab filter designer are normalized values between 1 and -1. So, the filter coefficients are floating-point values.
  * But in the microcontroller, decimal values need more memory to store than integer values. Therefore, as a method to convert those decimal values to integer values, the coefficients are multiplied by 1000. Then those values are rounded up to the nearest integers and stored.

    - To avoid the error, we get from this kind of conversion, a single overlap multiplication value is divided by 1000 and added to the result.

- Speed up the calculation

    - The calculations are done by reading the audio file samples from the SD card. To read a certain sample takes some time. And, to read a bunch of data takes the same time. Therefore, reading samples one by one is a cost of time. As a solution buffered set of samples are kept inside the program while calculating. Since the samples are inside the microcontroller's memory now, it is easy to access and cost less time.

Example:
Stored Coefficients: {-18, -46, -83, -120, -148, 834, -148, -120, -83, -46, -18}
S = Sample Value
V = DC off set
K = Overlapped filter coefficient

$$TemporaryValue = \frac{(S - V) \times K}{1000}$$

The temporary values are added up together. Finally, the DC offset value is added to the summation.
Output sample = Summation + DC Offset

2.4 Output

We wanted to get a smooth analog signal from the speaker. But the audio file stored in the SD card can only be produced digitally by the ATmega328p microcontroller. Using an additional DAC module in the circuit will need more pins from the microcontroller. It will also need an external power supply which makes the circuit even complex. Using the PWM technique for audio output will reduce the hardware complexity of the circuit.

PWM
Considering the available pins in the ATmega328p microcontroller, use the Pulse Width Modulation (PWM) technique for Digital to Analog (DAC) is preferred.
The more the PWM frequency the better DC will be obtained. Therefore, we set the pre scaling factor to 1 and obtain a PWM frequency of 62.5kHz.

$$f_{OCnxPWM} = \frac{f_{clk\_I/O}}{N \cdot 256} = \frac{16MHz}{1 \times 256} = 62.5kHz$$

Filtering
As our PWM signal frequency is 62.5kHz, we can choose any low pass filter with a cut-off frequency of less than 62.5kHz to filter out the DC value of the signal.
The lower the cut-off frequency cleaner the DC output will be. Here, we used a simple RC low-pass filter with a cut-off frequency of around 8kHz.

$$R = 1k, C = 0.02\mu F$$

$$f_c = \frac{1}{2 \cdot \pi \cdot RC} = \frac{1}{2 \times \pi \times 1 \times 10^3 \times 0.02 \times 10^{-6}}$$

$$f_c = 7.957kHz$$

Amplification
Signal strength is not always enough after performing sampling and filtering in different modes.
We use the PAM8403 amplifier circuit for amplifications mainly because it is readily available in local markets. Also, it can be powered up by a simple 5V input and can directly drive the speaker through the output. The maximum gain of the amplifier is 24dB.

2.5 Enclosure and PCB Design

2.5.1 PCB Design

The PCB design for this device was made using the software Altium Designer 21.6.1.
All the four controlling push buttons and the reset push button in the circuits are placed at the bottom potion of the PCB. Also, an OLED display was placed on top of the PCB to ensure proper connection.
The schematic of the PCB and circuit diagrams can be seen in Appendices.

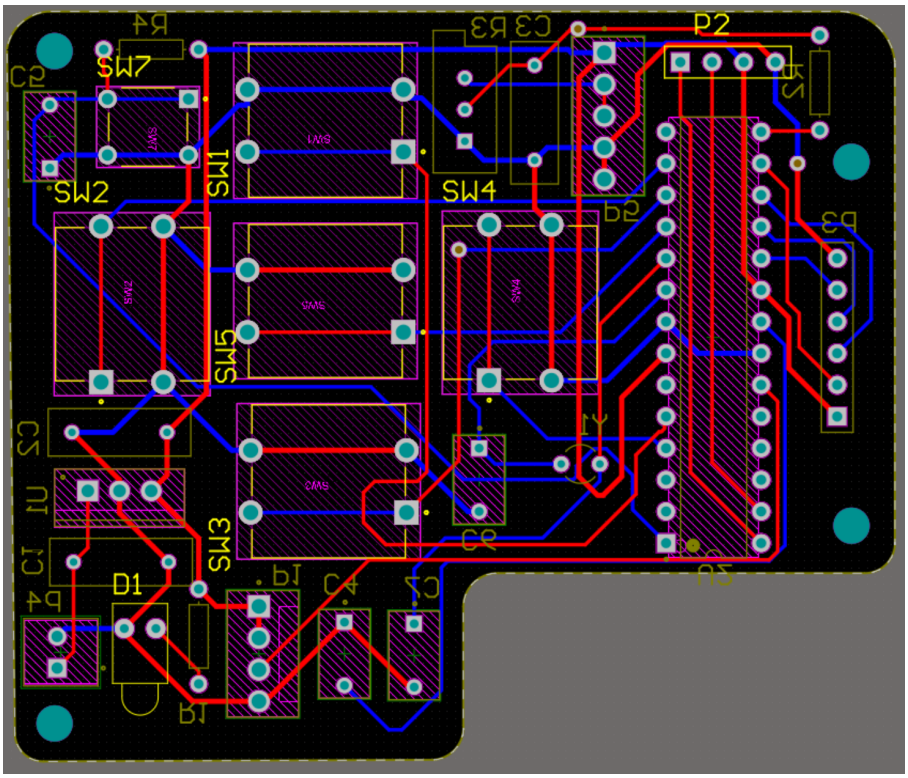


Figure 17: PCB using Altium

2.5.2 Enclosure Design

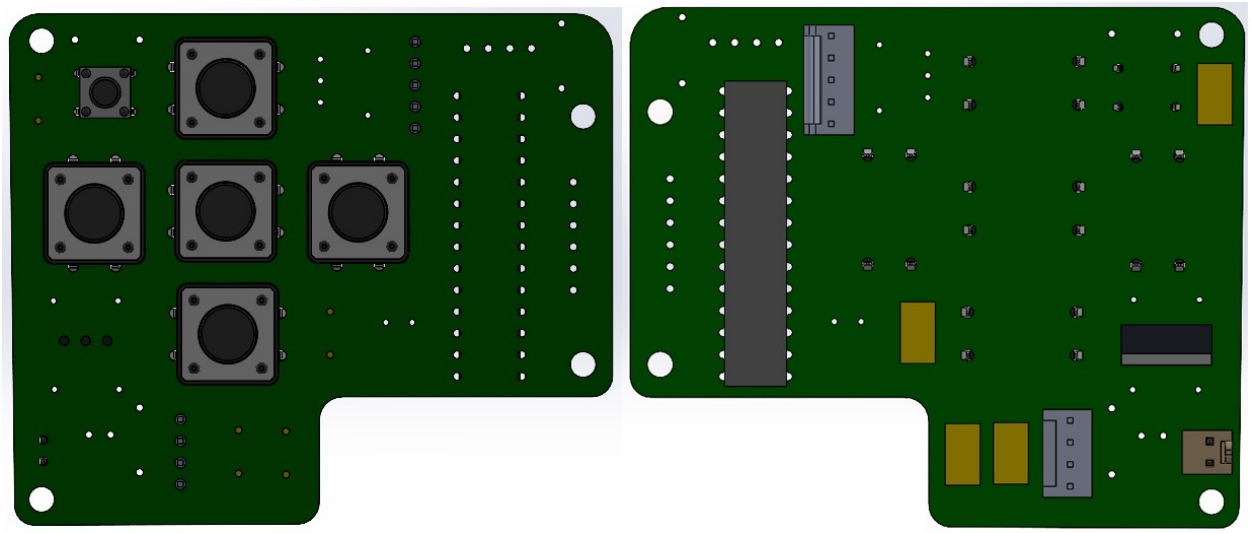


Figure 18: 3D model of the PCB

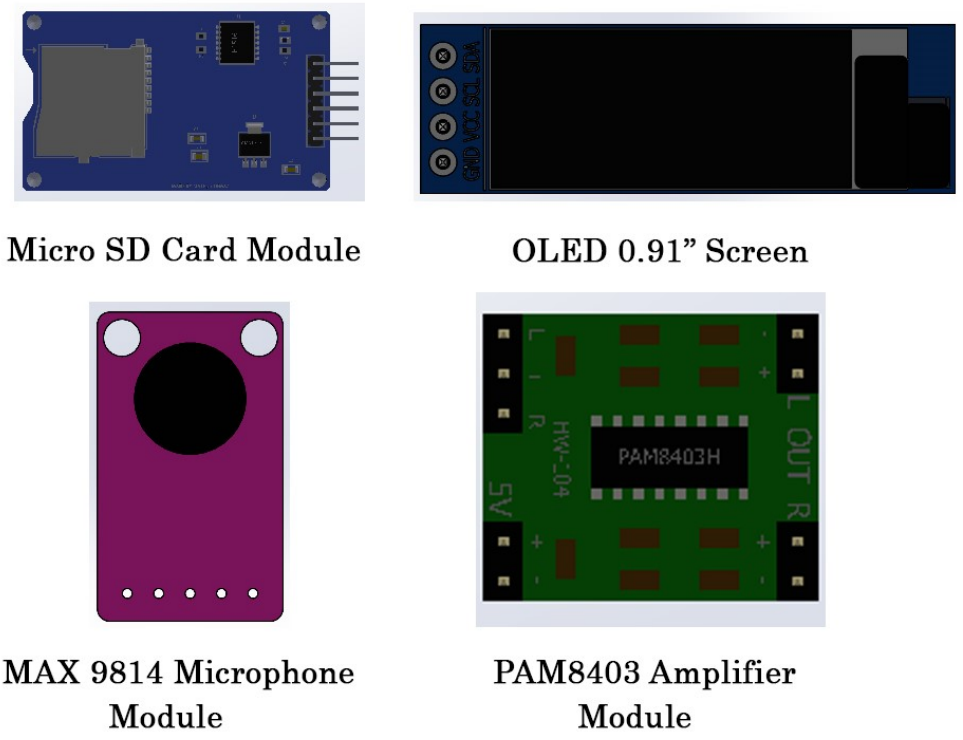
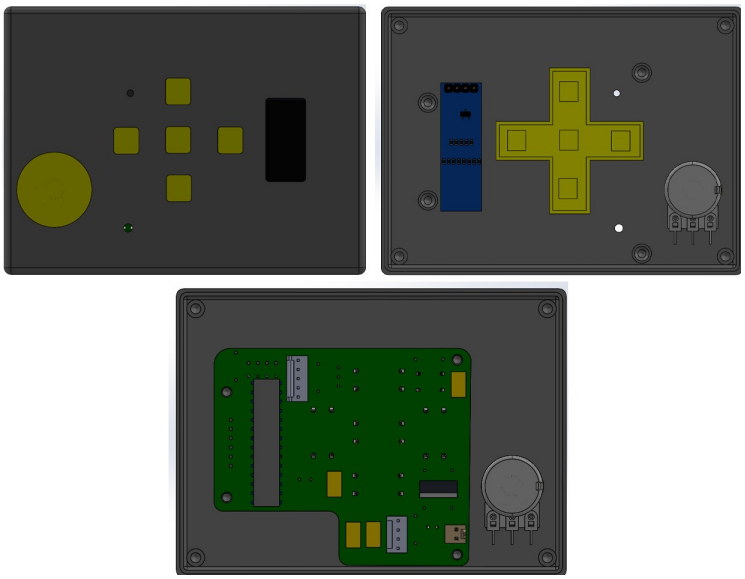
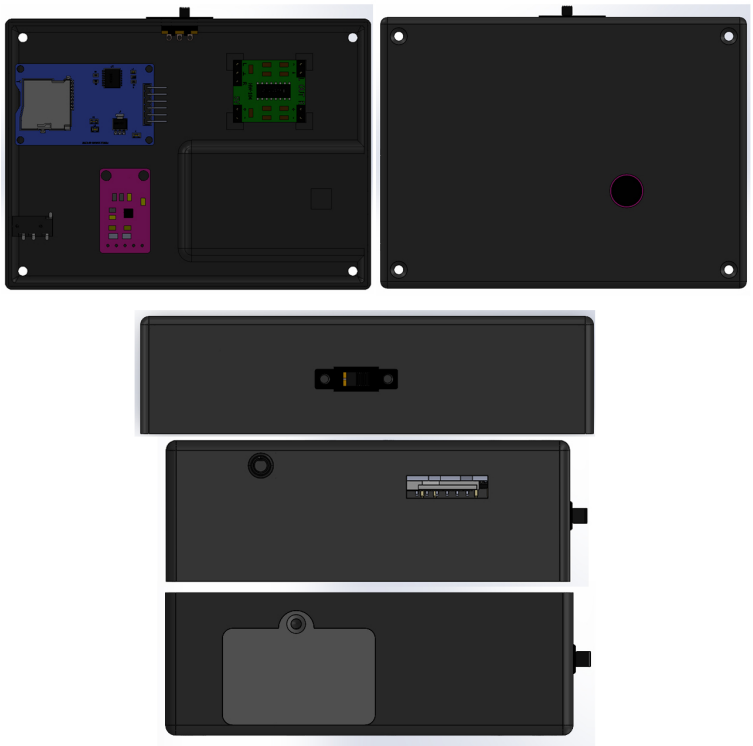


Figure 19: Modules



(a) Top Enclosure



(b) Bottom Enclosure

Figure 20: Parts of the Enclosure

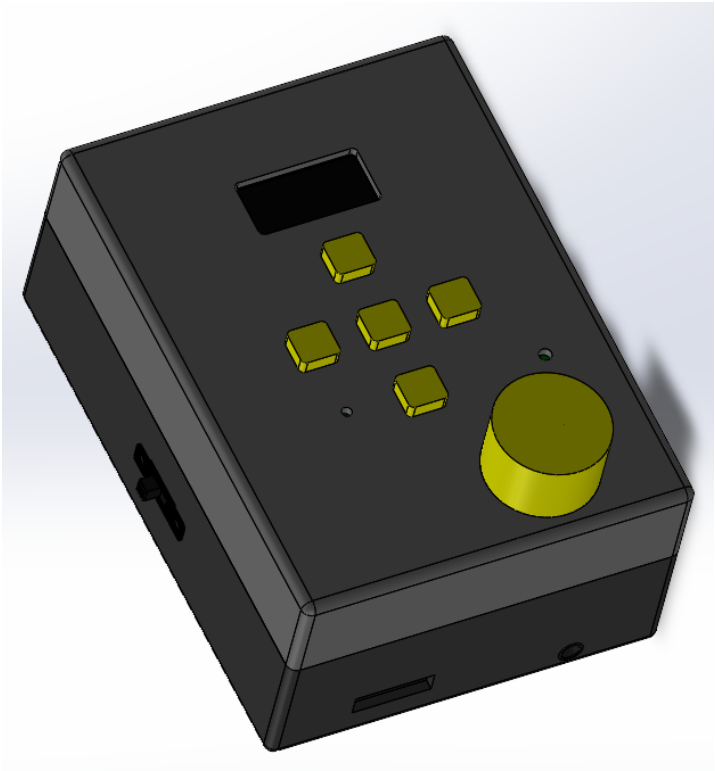


Figure 21: Fully Assembled Enclosure

2.6 Component List

Here we have listed only the main components that we used in our device and included some of the specifications that we considered as most relevant to the operation of the device.

Component	Specifications	Advantages
ATmega328p MC	CPU Type: 8-bit AVR Max CPU speed: 20 MHz Flash memory: 32 KB SRAM: 2 KB EEPROM: 1KB Package pin count: 28 / 32 Maximum I/O pins: 23 External Interrupts: 2	Supports SPI and Serial communication Manageable number of pins Sufficient memory availability Fairly good CPU speed
MAX9814 Microphone	Supply Voltage: 2.7V-5.5V Output: 2Vpp on 1.25V bias Frequency Response: 20Hz – 20kHz Automatic gain (40dB/50dB/60dB) Input-Referred Noise Density ($\frac{30nV}{\sqrt{Hz}}$)	Automatic Gain Control (AGC) Low-cost, high-quality amplifier Low-noise microphone biasing
OLED (0.91”) Display	Organic LED technology Resolution: 128 $\times$32 Size: 0.91” I2C or SPI compatible Power: 3.3V – 5V	No back light needed, self-illuminance Lower power consumption compared to LCDs Higher Resolution Good looking and handy to use Uses only 2 pins on MCU
Micro SD card Module	SPI communication Power: 2.7V-3.6V	SPI communication
PAM8403 Amplifier	Operating range: 2.5V-5.5V Output power: 3W + 3W (at 4Ω) Board size: 24 $\times 15mm$ provides a maximum gain of 24dB	Operating range ideal for audio capability High amplification efficiency (88%) Low cost Portable Built-in short circuit protection No heat sink needed Low noise

Table 1: List of Components

3 Results

3.1 Simulation Results

Before going for an actual representation, the whole implementation was done in a virtual platform. Proteus Simulation Software is used in this project. Sound recording and storing could be done in real-time in the simulations. But the sound output was not happening in the way we want. So, we used the simulation just to simulate recording, Storing, and comparing input and output signals. And for the sound output test, we used a breadboard implementation.
Shown below are graphs of input and output waves took from the simulation. This simulation uses a single OpAmp to amplify the signal. Therefore, the signal is not amplified properly. The recovered signal was approximately equal to the input signal. Some noise was there.

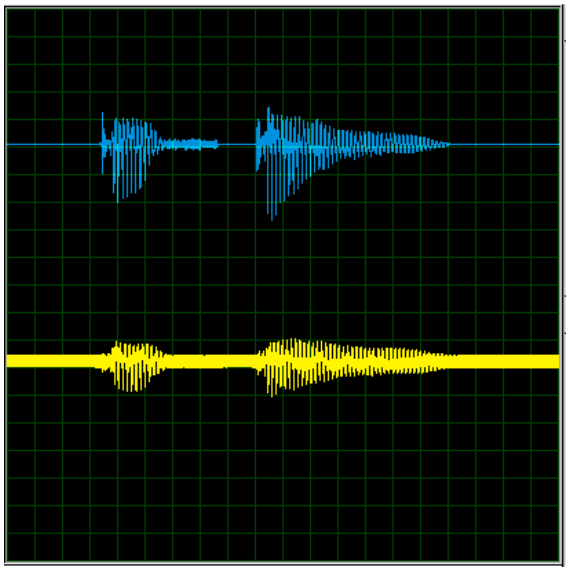


Figure 22: Input and Output Signals from Simulation

Blue : Input Signal
Yellow : Output Signal

The amplifier we used in this application was not available in this software, therefore we cannot assure how the waveform will look like only from this simulation result.

3.2 Prototype Results

The prototype was made using the same schematic diagram used in PCB design. But the actual design is a double layer PCB. For the prototype, we did a separate PCB design so it can be made at home using etching techniques. After that, all the modules and other components were assembled in one place. The functionalities of the final product were fully implemented in this prototype.

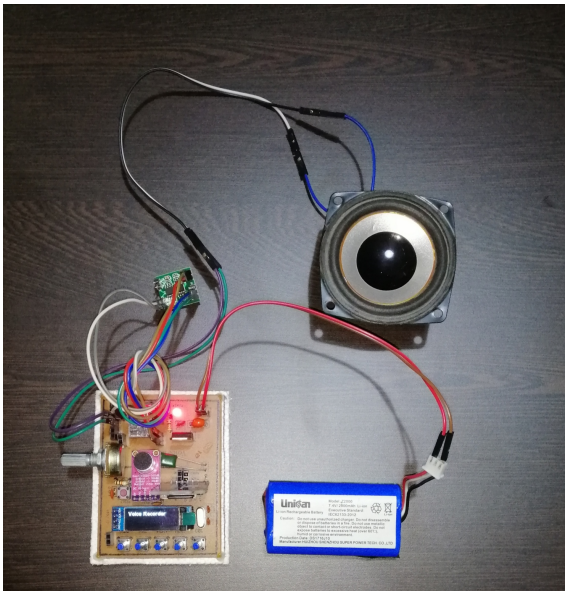


Figure 23: A picture of the prototype

3.2.1 Record and Play

The major goal of the project was to record a voice signal, store it and play it. As planned the voice signal could be recorded in the way we want, and recordings were successfully stored as WAVE files. 8bit resolution is less quality but overall recordings were clear.
And for the output, the method used was PWM. This method is not very suitable for audio but with the resources in hand, it was the best solution. The output sound was clear to understand but there were some noises we can hear. The reason for that can be understood as the DAC method

we used. Fast PWM is not a smooth converting method, therefore noises exist.

3.2.2 Frequency Changes

Frequency Scaling
The down sampling was done in the program in real-time. Therefore, no additional files were created. The changes were understandable by hearing. Since there are no additional files created, we could not check the spectrum of the scaled version.

Signal Enhancing
After filtering the signals digitally, new files were created to store the enhanced version. So, the spectrums of those signals were able to monitor.

Original Signal Spectrum:

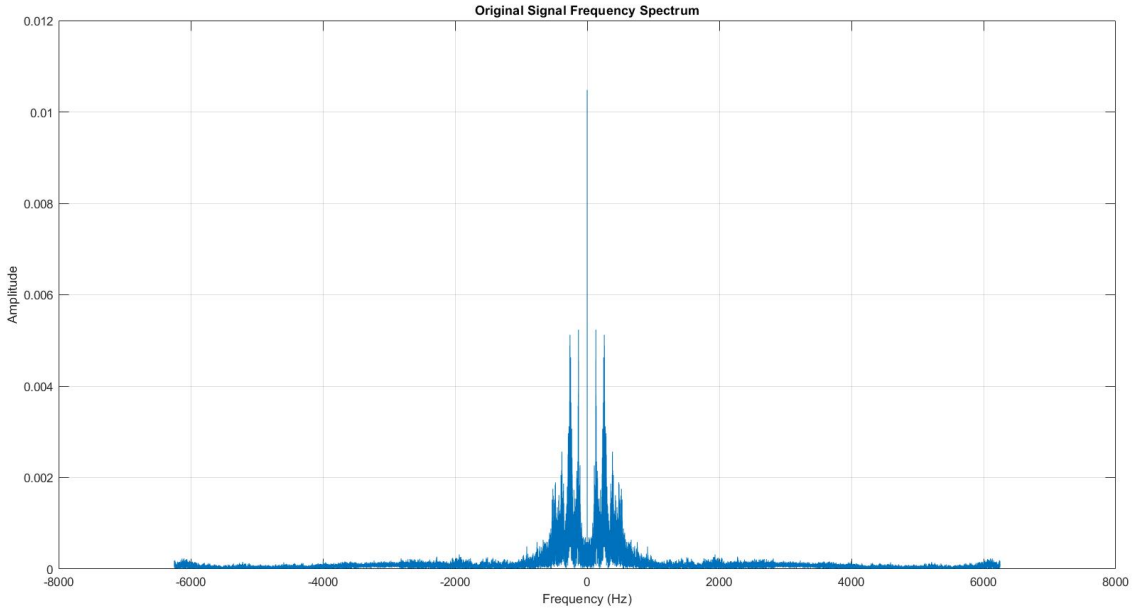


Figure 24: Original signal spectrum

Filtered Signal Spectrum:

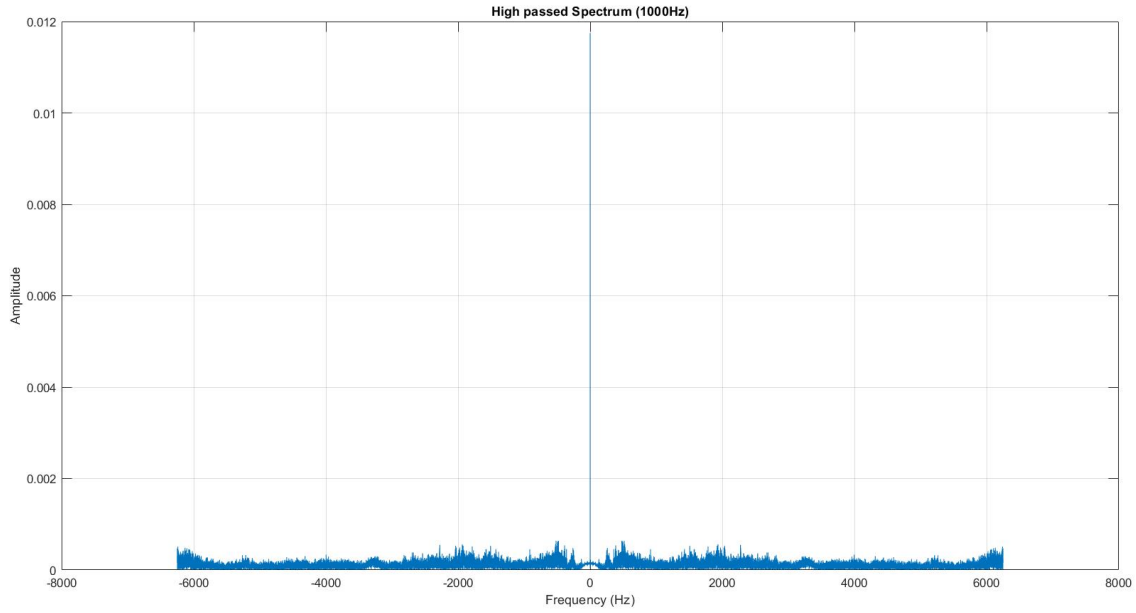


Figure 25: High pass Filter (Cutoff = 1kHz)

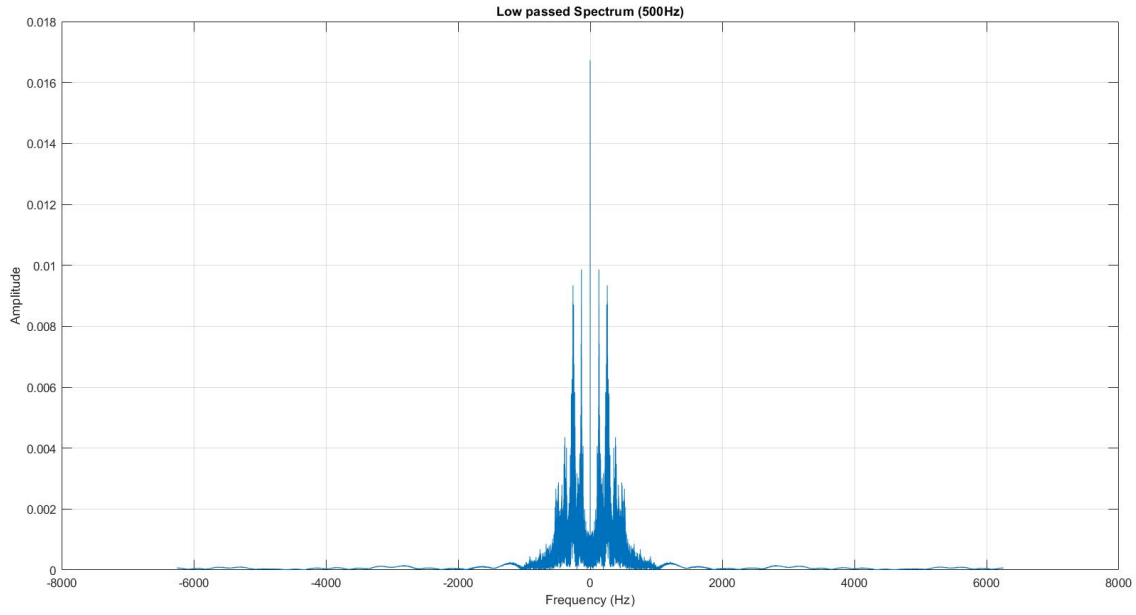


Figure 26: Lowpass Filter (Cutoff = 500Hz)

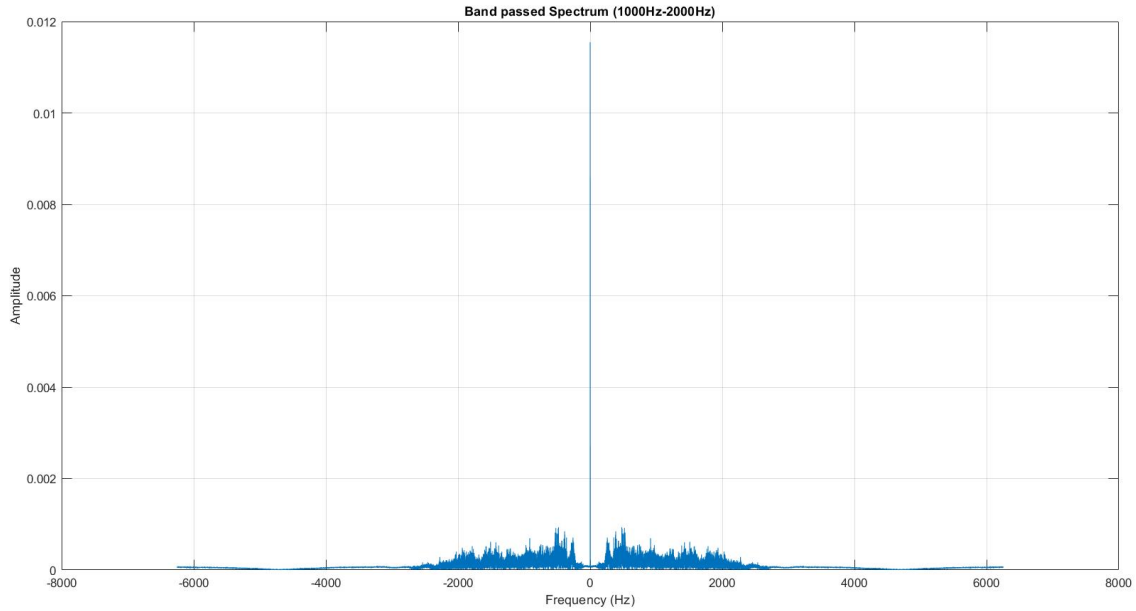


Figure 27: Bandpass Filter (F1 = 1kHz, F2 = 2kHz)

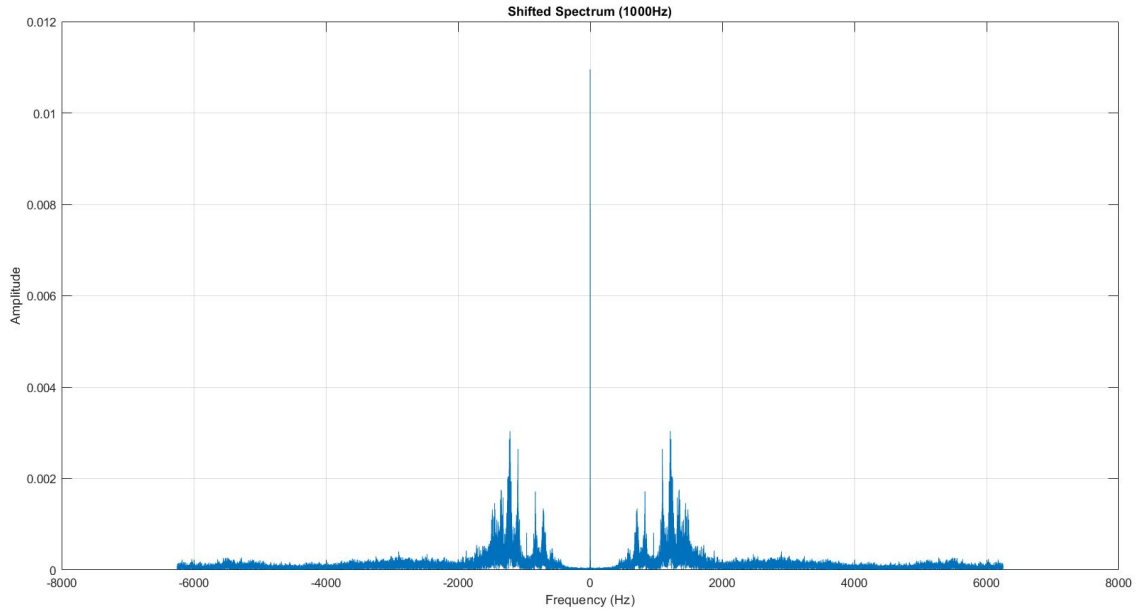


Figure 28: Frequency shift (1kHz)

4 Discussion

Initially Arduino Uno was used to do this project. Later it was decided that using a microcontroller unit would be a better option as MCUs have better processing power compared to Arduino Uno. ATmega328P was chosen as the MCU as it is suitable for this application. Therefore, all the central ideas and algorithms which were implemented in the Arduino environment were later changed into AVR codes.
Difficulties were faced when choosing a proper Digital to Analog Conversion method. Initially, we went with the R-R bridge which uses a lot of resistors. Due to the high-power dissipation at resistors, this method proved to be inefficient in terms of power consumption.
A couple of options were considered to overcome this issue.

1. Using a DAC 0808
2. Using a PWM technique

Despite being highly efficient with power consumption, using a DAC 0808 raised a few other concerns. This method would require 8 pins from the microcontroller. This cannot be afforded with the limitation on the number of pins we have on an ATmega328P microcontroller. Hence, finally we ended up with the PWM method where we can take the output from a signal input pin.
Getting a proper sampling rate is yet another problem that we faced. Based on Nyquist–Shannon sampling theorem we must need a sampling rate of more than 8kHz for voice signals. We were able to overcome this issue by changing the Prescaler value to 16 by making changes in the ADCSRA register.
When performing the Frequency enhancing, we have found that filter coefficient values are floating-point numbers which would take a lot of memory space. This created memory issues. This problem was handled by multiplying the values by 1000 and converting them to integer values, which would require a significantly lesser memory space, and then performing the necessary operations. Finally, the end answer is obtained by dividing the output from the filtering by 1000.
It was observed that the following improvements would help make this product a better and a more marketable product.

1. PWM was used to get the output signal from the speaker, but that using a proper DAC module that uses a smaller number of pins will give a better output signal.
2. The use of a smaller-in-size enclosure, the voice recorder would be more convenient as a compatible device.

5 Acknowledgement

We would like to extend our deepest gratitude to our supervisor Mr.Sahan Liyanaarachchi, for his valuable guidance, collaboration, patience, and other essential instructions and comments provided during the weekly progression inspection sessions. That was the backbone of the project. We also like to pay our regards to our instructor Mr.Lahiru Wijerathna for his valuable comments and guidance, especially in PCB and enclosure designs.
We must also thank all the academic lecturers, instructors, and other academic staff whose contribution indirectly helped us a lot in the completion of this project successfully.
Finally, we would like to thank our fellow batch mates for sharing their knowledge and experience during the hard times.

References

[1] "Microsoft WAVE soundfile format." http://soundfile.sapp.org/doc/WaveFormat/.

[2] TheElectroSTUD, "PAM8403 6W STEREO AMPLIFIER TUTORIAL." https://www.instructables.com/PAM8403-6W-STEREO-AMPLIFIER-TUTORIAL/.

[3] Amandaghassaei, "Arduino Audio Output." https://www.instructables.com/Arduino-Audio-Output/.

[4] "Detailed explanation of WAV file format." https://www.fatalerrors.org/a/detailed-explanation-of-wav-file-format.html.

[5] "ATMEL AVR Tutorial 2 : How to access Input / Output Ports ?." https://www.elecrom.com/avr-tutorial-2-avr-input-output/, Feb. 2008.

[6] "DAC using PWM | Playing with Systems." https://sysplay.in/blog/tag/dac-using-pwm/, language = en-US.

[7] "Turn Your PWM into a DAC - Technical Articles." https://www.allaboutcircuits.com/technical-articles/turn-your-pwm-into-a-dac/.

[8] "PWM DAC | Open Music Labs." http://www.openmusiclabs.com/learning/digital/pwm-dac.1.html.

[9] "init()." https://garretlab.web.fc2.com/en/arduino/inside/hardware/arduino/avr/cores/arduino/wiring.c/init.html, Apr. 2019. Section: arduino.

[10] "Windowed-Sinc Filters." http://www.dspguide.com/ch16.htm, urldate = 2021-08-16.

[11] "Properties of the Fourier Transform - DSPIllustrations.com." https://dspillustrations.com/pages/posts/misc/properties-of-the-fourier-transform.html.

[12] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, *Signals and Systems in English - 2nd ed.* Book News, Inc., Portland, OR, 1997.

[13] "Using Filter Designer - MATLAB & Simulink - MathWorks India." https://in.mathworks.com/help/dsp/ug/using-filter-designer.html.

[14] "megaavr® Data Sheet - ATmega48A/PA/88A/PA/168A/PA/328/P." https://www.microchip.com/en-us/product/ATmega328P#document-table.

[15] "ATMEL AVR Tips – Input Output Ports Code Snippets." https://www.elecrom.com/atmel-avr-tips-inputoutput-ports-code-snippets/, Aug. 2015.

[16] S. Deuty, "Planet Analog - ADC's Analog-to-Digital Converters basics." https://www.planetanalog.com/adcs-analog-to-digital-converters-basics/, Nov. 2017.

6 Appendices

6.1 Proteus Simulation Schematic

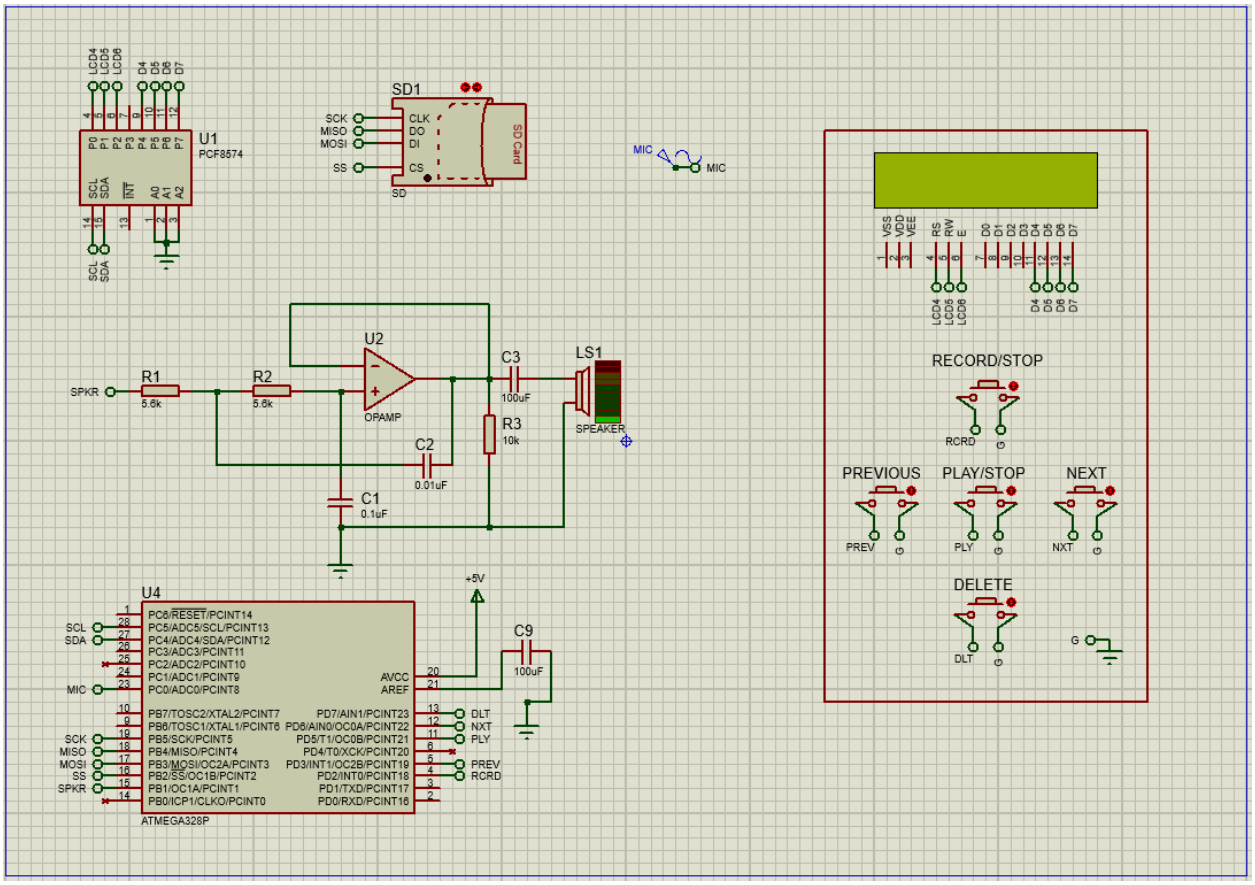


Figure 29: Proteus Simulation Schematic

6.2 PCB Schematic

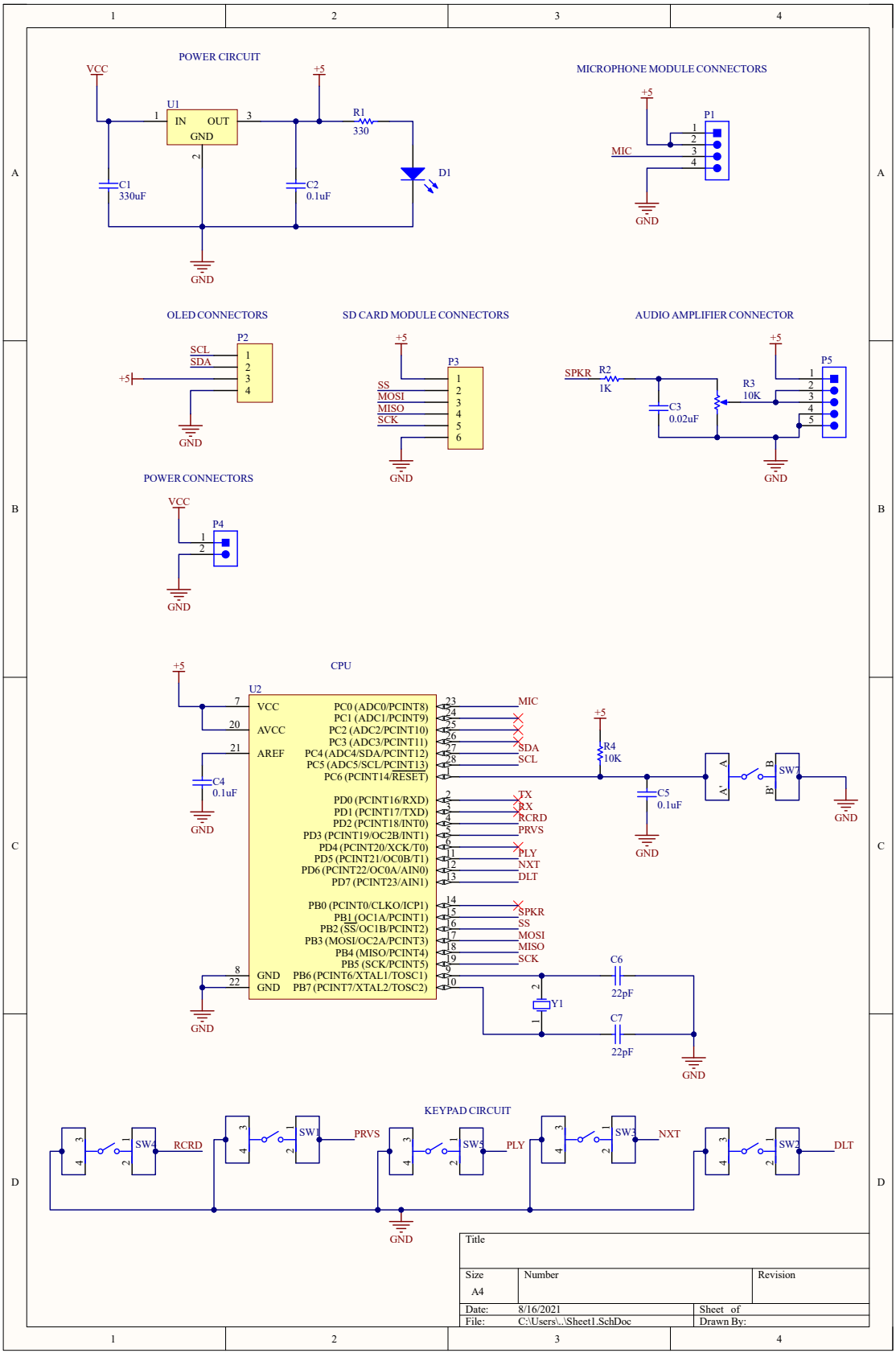
Click here

6.3 AVR Code

Click here

6.4 Instructions for Use

Click here


POWER CIRCUIT
VCC
+5
U1
IN
OUT
GND
R1
330
C1
330uF
C2
0.1uF
D1
GND
MICROPHONE MODULE CONNECTORS
+5
P1
MIC
GND
OLED CONNECTORS
P2
SCL
SDA
+5
GND
SD CARD MODULE CONNECTORS
+5
P3
SS
MOSI
MISO
SCK
GND
AUDIO AMPLIFIER CONNECTOR
SPKR
R2
1K
C3
0.02uF
R3
10K
+5
P5
GND
POWER CONNECTORS
VCC
P4
GND
CPU
+5
U2
VCC
AVCC
AREF
C4
0.1uF
GND
PC0 (ADC0/PCINT8)
PC1 (ADC1/PCINT9)
PC2 (ADC2/PCINT10)
PC3 (ADC3/PCINT11)
PC4 (ADC4/SDA/PCINT12)
PC5 (ADC5/SCL/PCINT13)
PC6 (PCINT14/RESET)
PD0 (PCINT16/RXD)
PD1 (PCINT17/TXD)
PD2 (PCINT18/INT0)
PD3 (PCINT19/OC2B/INT1)
PD4 (PCINT20/XCK/T0)
PD5 (PCINT21/OC0B/T1)
PD6 (PCINT22/OC0A/AIN0)
PD7 (PCINT23/AIN1)
PB0 (PCINT0/CLKO/ICP1)
PB1 (OC1A/PCINT1)
PB2 (SS/OC1B/PCINT2)
PB3 (MOSI/OC2A/PCINT3)
PB4 (MISO/PCINT4)
PB5 (SCK/PCINT5)
PB6 (PCINT6/XTAL1/TOSC1)
PB7 (PCINT7/XTAL2/TOSC2)
GND
GND
MIC
SDA
SCL
TX
RX
RCRD
PRVS
PLY
NXT
DLT
SPKR
SS
MOSI
MISO
SCK
+5
R4
10K
C5
0.1uF
GND
A
A'
B
B'
SW7
GND
C6
22pF
Y1
C7
22pF
GND
KEYPAD CIRCUIT
SW4
RCRD
SW1
PRVS
SW5
PLY
SW3
NXT
SW2
DLT
GND
Title
Size
A4
Number
Revision
Date:
8/16/2021
Sheet of
File:
C:\Users\..\Sheet1.SchDoc
Drawn By:

```
1  //>----------------- LIBRARIES -----------------<
2  #include <avr/io.h>
3  #include <util/delay.h>
4  #include <avr/interrupt.h>
5  #include <SD.h>
6  #include <SPI.h>
7  #include <Wire.h>
8  #include <SSD1306Ascii.h>
9  #include <SSD1306AsciiAvrI2c.h>
10 //>----------- PIN  CONFIGURATIONS -------------<
11 #define mic 0b0000
12 #define sdcard 10
13 //>----------------- CALCULATIONS -----------------<
14 #define lower_Byte(w) ((uint8_t) ((w) & 0xff))
15 #define higher_Byte(w) ((uint8_t) ((w) >> 8))
16 #define sbi(port,bit) port |= 1<<bit
17 //>-------------- common values -------------------<
18 //Audio file attributes
19 #define sampleRate 12500
20 #define byteRate (sampleRate/8)*monoStereo*8
21 #define monoStereo 1
22 #define dcOffset 127
23 #define maxFiles 15
24 #define lcdAddr 0x27 //Change this to 0x20 for proteus simulation,0x27 for real application
25 #define I2C_ADDRESS 0x3C
26 //for filters
27 #define bufflen 26
28 #define temp_buff_size 75
29 #define coslen 13
30
31 //variables for mode
32 char mode = 'i';
33 char mode_ = 'j';
34 //variables for frequency changing configuration
35 uint8_t freqScal = 1;
36 //variables to handle file operations
37 char tracks[maxFiles];
38 uint8_t files = 0;
39 uint8_t fcount = 0;
40 char fname_temp[7];
41 //OLED instance
42 SSD1306AsciiAvrI2c oled;
43 //functions used in recorder
44 char keyInput();
45 //functions for LCD
46 void firstLine(const char*);
47 void clrDisplay(const char*);
48 void secondLine(const char*);
49 //Recording,playing functions
50 void record();
51 void playTrack();
52 void checkChanges();
53 void getTrackList();
54 void nextTrack();
55 void previousTrack();
56 void checkDuplicates();
57 void deleteTrack();
```

```
58  //wave file creating functions
59  void makeWaveFile(File);
60  void finalizeWave(File);
61  //frequency modification functions
62  void pickFilter(char);
63  void convolve(int[],char[],uint8_t,int);
64  void sig_freqShift(char[]);
65  //IO functions
66  void initialize_Things();
67  uint8_t analog_in(int);
68
69  int main(void){
70      //This function is from arduino...need to edit
71      initialize_Things();
72      fname_temp[1] = '.';fname_temp[2] = 'W';fname_temp[3] = 'A';fname_temp[4] = 'V';
73      //PORTD FOR KEYS
74      DDRD = 0b00000000;
75      PORTD = 0b11111111;
76      //CONFIGURING PINS FOR ANALOG INPUT
77      DDRC &= 0b11111110;
78      //CONFIGURING SPEAKER FOR OUTPUT
79      DDRB |= (1<<DDB1);
80      OCR1A = 0;
81      //beginning the OLED
82      oled.begin(&Adafruit128x32, I2C_ADDRESS);
83      //INITIALIZING THE SD CARD
84      if (!SD.begin(sdcard))
85      {
86          clrDisplay("Error");
87          while (1);
88      }
89      getTrackList();
90      _delay_ms(1000);
91      while (1)
92      {
93            //>-------------------------------< RECORD MODE (LEVEL 1)                    ↵
                >-------------------------------<
94            if (mode == 's' && mode_ == 'j')
95            {
96              record();
97              getTrackList();
98              mode = 'i';
99            }
100           //>-------------------------<< ENTERING PLAYER MODE (LEVEL 1)                ↵
                >---------------------------<
101           if (mode == 'p' && mode_ == 'j')
102           {
103             mode_ = 'k';
104             mode = 'i';
105             /*
106               This is the player mode
107               It loads the tracks in alphabetic order
108               Press 'Play/Stop' when a track is loaded to the player
109               Press 'Play/Stop' to stop playing
110               Press 'next' or 'previous' to toggle between tracks
111               Press 'record/stop' in track loaded mode to exit player mode
112             */
```

```
113             if (files == 0)
114             {
115               mode_ = 'j';
116               clrDisplay("No Tracks");
117               _delay_ms(1000);
118             }
119             else
120             {
121                 clrDisplay("Ready to Play");
122               _delay_ms(1000);
123               fname_temp[0] = tracks[fcount];
124               secondLine(fname_temp);
125             }
126           }
127           //>-------------------------< PLAYER MODE (LEVEL 2)
                >-------------------------------------<
128           if (mode_ == 'k')
129           {
130               while (1)
131               {
132                   char key_input = keyInput();
133                   if (key_input)
134                   {
135                       mode = key_input;
136                       break;
137                   }
138               }
139             if (mode == 'p')
140             {
141               //Play the track
142               playTrack();
143               clrDisplay("Ready to Play");
144             }
145             else if (mode == '>')
146             {
147               //Load the next track
148               nextTrack();
149             }
150             else if (mode == '<')
151             {
152               //load the previous track
153               previousTrack();
154             }
155             else if (mode == 's')
156             {
157               //Exit from player mode
158               mode_ = 'j';
159               mode = 'i';
160             }
161             else if (mode == 'd')
162             {
163                 //This mode deletes the track loaded in payer
164                 clrDisplay("Delete?");
165                 secondLine("DELETE NO(Play)");
166                 while (true)
167                 {
168                     char key = keyInput();
```

```
169                     if (key && key == 'd')
170                     {
171                         deleteTrack();
172                         clrDisplay("Deleted");
173                         getTrackList();
174                         if(fcount == files){
175                             fcount--;
176                         }
177                         _delay_ms(1000);
178                         break;
179                     }
180                     else if (key=='p')
181                     {
182                         clrDisplay("Not Deleted");
183                         _delay_ms(1000);
184                         break;
185                     }
186                 }
187                 if (files == 0)
188                 {
189                     clrDisplay("No Tracks");
190                     _delay_ms(1000);
191                     mode_ = 'j';
192                 }
193                 else{
194                     clrDisplay("Ready to Play");
195                     fname_temp[0] = tracks[fcount];
196                 }
197               }
198               secondLine(fname_temp);
199               mode = 'i';
200             }
201             //>-------------------------< PAUSE MODE (LEVEL 1)                    ↵
                  >--------------------------------------<
202             if (mode == 'i' && mode_ == 'j')
203             {
204               clrDisplay("Voice Recorder");
205               while (1)
206               {
207                   char key_input = keyInput();
208                   if (key_input)
209                   {
210                       mode = key_input;
211                       break;
212                   }
213               }
214             }
215     }
216 }
217 /*
218    Definitions and implementations all the functions used in recorder
219 */
220 //>---------------------------------< KEYPAD FUNCTIONS >-----------------------------------<
221 char keyInput(){
222   /*
223     This function detects a key press and return the corresponding key
224   */
```

```
225   char k = 0;
226   while (uint8_t m = ~PIND){
227     switch (m) {
228       case 1: k = '_'; break;
229       case 2: k = '*'; break;
230       case 4: k = 's'; break;
231       case 8: k = '<'; break;
232       case 32: k = 'p'; break;
233       case 64: k = '>'; break;
234       case 128: k = 'd'; break;
235       default: k = 0; break;
236     }
237     _delay_ms(300);
238     PORTD = 0b11111111;
239   }
240   return k;
241 }
242 //END OF KEYPAD FUNCTIONS
243 //>-------------------------------< LCD DISPLAY FUNCTIONS >---------------------------------<
244 void firstLine(const char *msg){
245   //Prints the string passed in the first line of the display
246   oled.setFont(Arial_bold_14);
247   oled.println(msg);
248 }
249 void clrDisplay(const char *msg){
250   //Clears the screen and displays the msg in first line
251   oled.clear();
252   oled.setFont(Arial_bold_14);
253   oled.println(msg);
254 }
255 void secondLine(const char *msg){
256   //Prints the string passed in the second line of the display
257   oled.setFont(ZevvPeep8x16);
258   oled.println(msg);
259 }
260 //END OF LCD DISPLAY FUNCTIONS
261 //>----------------------------< RECORD AND PLAY FUNCTIONS >--------------------------------<
262 void record(){
263      /*Used to record the data got from input into a file*/
264       checkDuplicates();
265       File test_File = SD.open(fname_temp, FILE_WRITE);
266       if (!test_File) {
267         clrDisplay("Error");
268       }
269       else {
270         clrDisplay("Recording");
271         makeWaveFile(test_File);
272         uint8_t pot_Read;
273         while (true) {
274           pot_Read = analog_in(mic) + 64;
275           char key = keyInput();
276           if (key && key == 's') {
277             break;
278           }
279           test_File.write(pot_Read);
280           _delay_us(28);// 12.5kHz
281         }
```

```
282           finalizeWave(test_File);
283           clrDisplay("Saved");
284           _delay_ms(1000);
285         }
286         test_File.close();
287   }
288   void playTrack(){
289       /*This function reads data from the specified file and play*/
290       checkChanges();//check for frequency change requirements
291       clrDisplay("Playing");
292       File test_File = SD.open(fname_temp);
293       if (!test_File) {
294       // if the file didn't open, print an error:
295       secondLine("Error");
296       _delay_ms(1000);
297       }
298       else {
299           test_File.seek(44);
300           secondLine(fname_temp);
301           //Check whether a frequency scale is set
302           //>--------------------------< NORMAL OUTPUT >---------------------------------<
303           if (freqScal == 0 || freqScal == 1) {
304           while (test_File.available()) {
305           OCR1A = test_File.read();
306           _delay_us(40);  //Use this delay for 12.5KHz play
307           char key = keyInput();
308           if (key && key == 'p') {
309             break;
310           }
311         }
312       }
313       //>--------------------------< SCALED OUTPUT >---------------------------------<
314       //Output for frequency scaled track
315       //Using down sampling
316       else {
317         uint8_t count = 1;
318         while (test_File.available()) {
319           char key = keyInput();
320           if (key && key == 'p') {
321             break;
322           }
323           if (count == 1) {
324             //Accept the first sample among (# of samples=freqScal)
325             OCR1A = test_File.read();
326             _delay_us(40);  //Use this delay for 12.5KHz play
327           } else {
328             test_File.read();//This is to neglect samples in between
329           }
330           count++;
331           if (count == freqScal + 1) {//resetting the count
332             count = 1;
333           }
334         }
335       }
336       // close the file:
337       OCR1A = 0;
338       secondLine("End of play");
```

```
339     test_File.close();
340   }
341   _delay_ms(1000);
342    fname_temp[0] = tracks[fcount];fname_temp[1] = '.';fname_temp[2] = 'W';fname_temp[3] =
        'A';fname_temp[4] = 'V';fname_temp[5] = NULL;
343 }
344 void checkChanges(){
345     //This function checks for frequency change requirements
346     clrDisplay("SCL M");
347     char fsc =49;
348     char fshift='X';
349     char row[6] = {' ',fsc,' ',' ',fshift};
350     while(true){
351         char key_input = keyInput();
352         if (key_input=='p')
353         {
354             freqScal = uint8_t(fsc) - 48;
355             break;
356         }
357         else if(key_input=='>'){
358             if(fshift=='X'){
359                 fshift='H';
360             }
361             else if(fshift=='H'){
362                 fshift='L';
363             }
364             else if(fshift=='L'){
365                 fshift='B';
366             }
367             else if(fshift=='B'){
368                 fshift='S';
369             }
370             else if(fshift=='S'){
371                 fshift='X';
372             }
373         }
374         else if(key_input=='<'){
375             if(fsc==51){
376                 fsc=48;
377             }
378             fsc++;
379         }
380         row[1]=fsc;row[4]=fshift;
381         secondLine(row);
382     }
383     if(fshift!='X'){
384           clrDisplay("Processing");
385           pickFilter(fshift);
386     }
387 }
388 //END OF RECORD AND PLAY FUNCTIONS
389 //>-------------------------------------< FILE HANDLING FUNCTIONS
    >-------------------------------------<
390 void getTrackList(){
391   /*
392      This function checks for files and make a list of available files, Program only accept 15
        tracks
```

```
393       counts the number of files
394    */
395    char ASCIIcount = 65;
396    files = 0;
397    fname_temp[0] = ASCIIcount;
398    uint8_t arrIndex = 0;
399    while (true) {
400      if (arrIndex == maxFiles || ASCIIcount == 91) {
401        break;
402      }
403      if (SD.exists(fname_temp)) {
404        tracks[arrIndex++] = fname_temp[0];
405        files++;
406      }
407      fname_temp[0] = ++ASCIIcount;
408    }
409    for (uint8_t i = arrIndex; i < maxFiles; i++) {
410      tracks[i] = '_';
411    }
412  }
413  void nextTrack(){
414    //Checks tracks in order and returns the next track
415    fcount++;
416    if (tracks[fcount] == '_') {
417      fcount = 0;
418    }
419    fname_temp[0] = tracks[fcount];
420  }
421  void previousTrack(){
422    //Checks tracks in order and returns the previous track
423    if (fcount == 0) {
424      fcount = files - 1;
425    }
426    else
427    {
428      fcount--;
429    }
430    fname_temp[0] = tracks[fcount];
431  }
432  void checkDuplicates(){
433    /*This function checks if the new file to be made is existing,
434      if does it generates a new name for the file*/
435    char count = 65;
436    fname_temp[0] = count;
437    while (true) {
438      if (SD.exists(fname_temp)) {
439        fname_temp[0] = ++count;
440      } else {
441        break;
442      }
443    }
444  }
445  void deleteTrack(){
446      //Checks for depending files of the current file and delete them.
447      SD.remove(fname_temp);
448      fname_temp[0] = 'S';fname_temp[1] = tracks[fcount];fname_temp[2] = '.';fname_temp[3] =
         'W';fname_temp[4] = 'A';fname_temp[5] = 'V';
```

```
449     if(SD.exists(fname_temp)){SD.remove(fname_temp);}
450     fname_temp[0] = 'H';
451     if(SD.exists(fname_temp)){SD.remove(fname_temp);}
452     fname_temp[0] = 'L';
453     if(SD.exists(fname_temp)){SD.remove(fname_temp);}
454     fname_temp[0] = 'B';
455     if(SD.exists(fname_temp)){SD.remove(fname_temp);}
456     fname_temp[0] = tracks[fcount];fname_temp[1] = '.';fname_temp[2] = 'W';fname_temp[3] =
          'A';fname_temp[4] = 'V';fname_temp[5] = NULL;
457 }
458 //END OF FILE HANDLING FUNCTIONS
459 //>------------------------------< FUNCTIONS FOR WAVE FILE CREATION
      >--------------------------------<
460 void makeWaveFile(File sFile){
461   /*
462      This function creates the wave header file required
463      All bytes should be in little endian format, except String values
464   */
465   sFile.write((uint8_t*)"RIFF    WAVEfmt ", 16);//Starting bytes of the wave header file
466   uint8_t chunk[] = {16, 0, 0, 0, 1, 0, 1, 0, lower_Byte(sampleRate), higher_Byte(sampleRate)};
467   /*
468      chunk[]
469      first 4 bytes: size of  previous data chunck
470      next 2 bytes: Audio format (1 - PCM)
471      next 2 byte: No of channels (Mono = 1, Stereo = 2) (in our case 1)
472      last two are the first two bytes of sample rate
473   */
474   sFile.write((uint8_t*)chunk, 10);
475   chunk[0] = 0; chunk[1] = 0; //end of sample rate bytes
476   //byteRate = (sampleRate/8)*monoStereo*8;
477   chunk[2] = lower_Byte(byteRate); chunk[3] = higher_Byte(byteRate); chunk[4] = 0; chunk[5] =
        0; // byteRate
478   //byte blockAlign = monoStereo * (bps/8);
479   //this is always equal to 1 in 8bit PCM mono channel
480   chunk[6] = 1; chunk[7] = 0; //BlockAlign
481   chunk[8] = 8; chunk[9] = 0; //bits per sample
482   sFile.write((uint8_t*)chunk, 10);
483   sFile.write((uint8_t*)"data    ", 8);
484 }
485 void finalizeWave(File sFile) {
486   //This function finalizes the wave file
487   unsigned long fSize = sFile.size();
488   fSize -= 8;
489   sFile.seek(4);
490   uint8_t chunk2[4] = {lower_Byte(fSize), higher_Byte(fSize), fSize >> 16, fSize >> 24};
491   sFile.write(chunk2, 4);//Writing chunksize to 5 - 8 bytes in wave file
492   sFile.seek(40);
493   fSize -= 36 ;
494   chunk2[0] = lower_Byte(fSize); chunk2[1] = higher_Byte(fSize); chunk2[2] = fSize >> 16; chunk2
        [3] = fSize >> 24;
495   sFile.write((uint8_t*)chunk2, 4);//Writting num of samples to 41-44 bytes in wave file
496 }
497 //END OF WAVE FILE CREATE FUNCTIONS
498 //>-------------------------------------< FREQUENCY CHANGES
      >-------------------------------------<
499 void sig_freqShift(char tempName[]){
500         File out = SD.open("temp.bin", FILE_WRITE);
```

```
501         File target = SD.open(tempName, FILE_READ);
502         target.seek(44);
503         uint8_t buff[bufflen];
504         //int16_t cosWave12_5[coslen] = {10, 10, 9, 7, 5, 3, 1, -2, -4, -6, -8, -9, -10, -10, -9,
               -8, -6, -4, -2, 1, 3, 5, 7, 9, 10};
505         int16_t cosWave12_5[coslen] =   {10,8,5,0,-4,-8,-9,-9,-6,-1,3,7,9};
506         uint8_t count = 0;
507         uint8_t buffCount = 0;
508         while (target.available()) {
509             buff[buffCount++] = (uint8_t)((int)(target.read() - dcOffset) * cosWave12_5[count+
                +] / 10 + dcOffset);
510             if (count == coslen )
511             {
512                 count = 0;
513             }
514             if (buffCount == bufflen) {
515                 buffCount = 0;
516                 out.write((uint8_t*)buff, bufflen);
517             }
518         }
519         out.close();
520         target.close();
521 }
522 void pickFilter(char M){
523     char tempName[6] = {tracks[fcount],'.','W','A','V'};
524     fname_temp[0] = M;fname_temp[1] = tracks[fcount];fname_temp[2] = '.';fname_temp[3] =
        'W';fname_temp[4] = 'A';fname_temp[5] = 'V';
525     if(!SD.exists(fname_temp)){
526         if(M=='S' || M=='H'){
527             int filter[11] =    {-18,-46,-83,-120,-148,834,-148,-120,-83,-46,-18}; //kaiser 1000fc
                beta1.84
528             if(M=='S'){
529                 sig_freqShift(tempName);
530                 convolve(filter,tempName,11,500);
531             }
532             else{
533                 convolve(filter,tempName,11,500);
534             }
535         }
536         else if(M=='B'){
537             int filter[11] =    {-69,-128,-109,13,170,242,170,13,-109,-128,-69}; //kaiser beta1.9
538             convolve(filter,tempName,11,500);
539         }
540         else if(M=='L'){
541             int filter[12] =    {37,57,77,96,111,119,119,111,96,77,57,37}; //KAISER BETA=-2
542             convolve(filter,tempName,12,500);
543         }
544     }
545 }
546 void convolve(int filter[],char tempName[],uint8_t filterlen,int divider){
547     uint8_t signal_in[filterlen];
548     uint8_t temp_buff[temp_buff_size];
549     float temp = 0;
550     uint8_t temp_count = 0;
551     File target;
552     if(fname_temp[0]=='S'){target = SD.open("temp.bin", FILE_READ);}
553     else{target = SD.open(tempName, FILE_READ);}
```

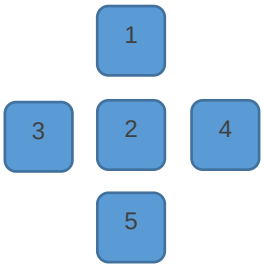
```
554     File out = SD.open(fname_temp, FILE_WRITE);
555     makeWaveFile(out);
556     unsigned long fSize = target.size();
557     target.read(signal_in, filterlen);
558     target.read(temp_buff, temp_buff_size);
559     while (fSize) {
560         if (temp_count == temp_buff_size) {
561             target.read(temp_buff, temp_buff_size);
562             temp_count = 0;
563         }
564         temp = dcOffset;
565         for (uint8_t i = 0; i < filterlen - 1; i++) {
566             temp += ((float(signal_in[i]) - dcOffset) * filter[i] / divider);
567             signal_in[i] = signal_in[i + 1];
568         }
569         temp += ((float(signal_in[filterlen - 1]) - dcOffset) * filter[filterlen - 1] / divider);
570         signal_in[filterlen - 1] = temp_buff[temp_count++];
571         out.write(uint8_t(temp));
572         fSize --;
573     }
574     finalizeWave(out);
575     out.close();
576     target.close();
577     if(SD.exists("temp.bin")){SD.remove("temp.bin");}
578 }
579 //END OF FREQUENCY CHANGES
580 //>-------------------------------------< IO FUNCTIONS >-------------------------------------<
581 uint8_t analog_in(int inputPin = 0000){
582     ADMUX |= inputPin;
583     ADCSRA = ADCSRA | (1 << ADSC);
584     while(ADCSRA & (1 << ADSC));
585     ADMUX &= 0b11110000;
586     return ADCH;
587 }
588 void initialize_Things(){
589     // this needs to be called before setup() or some functions won't
590     // work there
591     interrupts();
592     // on the ATmega168, timer 0 is also used for fast hardware pwm
593     // (using phase-correct PWM would mean that timer 0 overflowed half as often
594     // resulting in different millis() behavior on the ATmega8 and ATmega168)
595     sbi(TCCR0A, WGM01);
596     sbi(TCCR0A, WGM00);
597     // set timer 0 prescale factor to 64
598     // this combination is for the standard 168/328/1280/2560
599     sbi(TCCR0B, CS01);
600     sbi(TCCR0B, CS00);
601     // enable timer 0 overflow interrupt
602     sbi(TIMSK0, TOIE0);
603     // timers 1 and 2 are used for phase-correct hardware pwm
604     // this is better for motors as it ensures an even waveform
605     // note, however, that fast pwm mode can achieve a frequency of up
606     // 8 MHz (with a 16 MHz clock) at 50% duty cycle
607     TCCR1B = 0;
608     //select no-prescaling
609     sbi(TCCR1B, CS10);
610     //select the Wave form generation mode as FAST PWM
```

```
611     //select the non-inverting mode
612     sbi(TCCR1A, WGM10);
613     sbi(TCCR1A, COM1A1);
614     sbi(TCCR1B, WGM12);
615     // set timer 2 prescale factor to 64
616     sbi(TCCR2B, CS22);
617     // configure timer 2 for phase correct pwm (8-bit)
618     sbi(TCCR2A, WGM20);
619     // set a2d prescaler(16) so we are inside the desired 50-200 KHz range.
620     sbi(ADCSRA, ADPS2);
621     // enable a2d conversions
622     sbi(ADCSRA, ADEN);
623     //set the reference voltage as AVCC
624     //set the Left adjust result
625     //keeping last 3bits as 0, because for the default pin selection as ADC0
626     ADMUX = 0b01100000;
627     // the bootloader connects pins 0 and 1 to the USART; disconnect them
628     // here so they can be used as normal digital i/o; they will be
629     // reconnected in Serial.begin()
630     UCSR0B = 0;
631 }
632 //END OF IO FUNCTIONS
```

VOICE RECORDER

Instructions for use

❖ Keypad



*1 – **Record / Stop** Button*

*2 – **Play / Stop** Button – To play and stop the recordings, '**No**' command in delete mode*

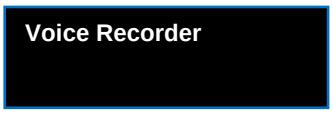
*3 – **Previous** Button – To load the previous recording, Selecting the frequency scale*

*4 – **Next** Button – To load the next recording, Selecting the filter*

*5 – **Delete** Button – Used to delete a recording*

Additionally, there is a volume controller beside the keys.

❖ When the device is powered up the screen looks like this. This is the pause state. No actions will happen.



If SD card is not inserted the device won't work. There will be an error message.

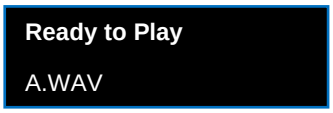
Insert the SD card and power it up again.

❖ When in the pause state press 'Record' to start recording.

Press '**Record**' again to stop recording.

Do not remove the SD card while recording.

❖ When in the pause state press 'Play'. That will guide you to play mode.



If there are no valid recordings in the SD card It will give you a message and will go back to pause mode.

(Valid means this device accepts recording names like A.WAV, B.WAV, ..., Z.WAV)

- Press **'Next**' and '**Previous**' to toggle between recordings. (Recordings will be in alphabetic order i.e.. A.WAV, B.WAV, ...,Z.WAV)
- Player holds only 15 recordings.
- Then press '**Play**' again. It will bring you to the frequency change requirement checking mode.

- Press '**Next**' button to change '**M**'(H, L, B, S, X) and press '**Previous**' to change **'SCL**'(1, 2, 3)
    - SCL - Frequency scale (Default = 1)
        - 1 - normal frequency
        - 2 - 2x frequency
        - 3 - 3x frequency

- M - Filtering method (Default = 'X')
  - X - No filtering
  - H - High pass (cut off = 1000Hz)
  - L - Low pass (cut off = 500Hz)
  - B - Band pass (1000Hz - 2000Hz)
  - S - Frequency shift (1000Hz)

- After selecting frequency changes press '**Play**' again.

  - If you didn't select any filters (M = X), the recording will play as you click play.
  - If You select value for **M** it will take some time to process the recording. After processing it will play.

    (Processed recording will be named by adding a prefix to the original wav file name. Prefixes- M values)

If 'M' is selected

Playing the recording

- At the end of play it will display a message. (Or you can press '**Play**' when playing the recording to stop the play.)

If the device fails to open the file specified, it will give an error.

(Possible reasons will be SD card is removed before playing or unknown malfunction in firmware)

❖ **When you are in the play mode you can delete the recording if you want. Press 'Delete' to proceed. It will give you a prompt.**

Press '**Delete**' again to proceed.

Press '**Play**' to avoid deleting.