Home Speaker Project - Signal Processing

Designing a Phased Speaker Array to Steer Sound

A Major Qualifying Project
submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

Sponsoring Agency: The Bose Corporation

Advisor: Joe Stabile, WPI

Submitted by:

Corey Coogan
Mark Panetta
Suilio Simo
Avik Vimal

Date Submitted:
April 18th, 2018

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see http://www.wpi.edu/Academics/Projects.

Table of Contents

Abstract 2

1. Introduction 2

2. Background 3

2.1 Cornell Phased Array 3

2.2 Diffraction Theory 4

2.3 Hawksford Smart Loudspeaker 6

2.4 Audio Testing Software Options 8

3. Methods 13

3.1. MATLAB Simulations 14

4. Implementation 18

4.1 Arduino Generated Sine Wave to One Speaker 18

4.2 Arduino Generated Sine Wave to Six Speakers with Phasing 20

4.3 Audio Input Circuit 23

4.4 Configuring Arduino ADC 23

4.5 Addition of Low Pass Filter After DAC 25

4.6 PIR Sensors 26

4.7 Stepper Motors 28

5. Recommendations/Conclusions 31

6. References 33

7. Appendix 34

4.1 One Speaker Full Code 42

4.2 Full 6 Speaker Program 44

4.4 Program with using ADC 46

4.6 PIR Sensor and Stepper Motor Code 47

Abstract

In this project we design a system to beam steer the soundwaves being outputted by a speaker based upon the location of a listener in the room. The concept is modeled after a phased array system which uses a progressive time delay calculated based upon the distance between speakers and the desired steering angle. The desired angle will be determined using infrared sensors to track an individual in front of the speaker and obtain the required delay to steer the sound towards them using the sensor's voltage output. Each time delay value will have a corresponding, unique, resulting steering angle of the beam. The microcontroller chosen to process this information is the Arduino Mega 2560 which will be used to sample the audio and apply the chosen time delays as well as obtain the sensor output voltage and calculate the corresponding delay using a second Arduino Mega. The basic implementation of the system starts with a circuit to obtain the analog audio input from the source which is then sampled and converted to a digital value using the on board ADC. The Arduino then manipulates when the samples are played out of each speaker in order to delay them properly. The samples are sent through an 8-bit DAC which converts the samples back to an analog waveform. The waveform is then low pass filtered and amplified before being played through the speaker. For the testing of the audio quality we created a MATLAB program for purposes of audio testing. The program generates a sine sweep signal and sends the signal to testing speaker and records the speaker's response. The recorded signal is then filtered and analyzed at different frequency points and a plot of the average peak values at different points versus frequency is obtained.

1. Introduction

The audio and sound market today is one of the largest and most valuable market in the world. The wireless audio market size is expected to almost double from USD 16.13 Billion in 2016 to USD 31.80 Billion by 2023. Looking at the current market, there seemed to be a lack of speakers that could be thin enough to hang on a wall similar to that of a television. Furthermore, there were very few affordable sound systems that could dynamically steer the sound to a desired angle. The goal of this project was to design and implement a system that will track an individual in front of the speaker unit using passive infrared sensors, process that location, and adjust the audio output to steer the sound waves accordingly. Steering the sound waves requires a unique, frequency independent time delay to be added to the output of each radiating element which will cause the wave to sum and cancel different locations resulting in an offset main lobe. The value of the time delay will control the exact angle the sound is steered at. The final system will provide an affordable way for consumers to purchase a non-obtrusive, high quality sound system which can provide features that are currently not present in most products on the market. The design of the steering array of speakers is modeled after phased arrays which are used very frequently in radar applications. Since radar applications usually use one constant frequency it can be easier to control and know where the main lobe is going. Phased arrays are designed with a thorough knowledge of diffraction theory, specifically N-slit diffraction. N-slit diffraction models how waves interact with each other when they pass through adjacent slits. These interactions, such as where they sum and cancel, are all dependent upon the size of the slits, the distance they are apart, and the frequency of the wave traveling through it. These concepts, how they were implemented, and how they performed in the system are discussed more thoroughly in the sections following.

2. Background

<u>2.1 Cornell Phased Array</u>

A recent project completed at Cornell [8] is very similar to our goal. They fabricated a unit that consisted of 12 speakers in a single row that is able to steer sound by adjusting a potentiometer. In order to steer the sound they derived an equation for the angle the sound is steered that is only dependent upon the time delay for each speaker. In the following equation, v_s represents the speed of sound, d is the distance between elements, and t_d is the time delay.

$$\theta = sin^{-1}(-\frac{v_s}{2\pi d}t_d)$$

This allows them to simplify the adjustment that needs to be made to a single time delay for each angle. This function was tested in MATLAB to show how adjusting the time delay will shift the main lobe of the sound wave. These test results are shown in the next section.

The report also details some of the electrical work that needed to be done for the system to work properly. The main elements of the system are an input buffer to amplify and re-bias the signal. That signal is then processed by an ADC to produce a digital value that can then be sent to the microcontroller. The microcontroller then performs the necessary operations to apply the time delay to each speaker. Each channel is then sent to a DAC to switch to the corresponding analog voltage and into the amplifier which contains a low pass filter to remove any noise from the DAC. A buffer is also needed to reduce the impedance of the DAC as the speaker has an impedance of just 8 ohms.

The first input amplifier is designed to bias the input voltage from -1V to +1V to 0V to 5V. This is necessary in order for the ADC to function properly and assign the appropriate digital value to each signal. A high pass filter is used to stop the low frequencies below the speaker's frequency response. A voltage divider is designed to lower the voltage from 12 down to as close to 2.5 as possible. An op-amp circuit is then used to amplify the -1V to 1V swing to a swing that is approximately 5V.

The ADC is needed to communicate the analog signal to the microcontroller in digital format. The ADC will have a range of 0-5V, which is why the signal needed to be biased in the previous step, and should be able to sample and communicate at a rate of 44.1 kHz. The microcontroller should be able to run the serial peripheral interface (SPI) at the same rate. The SPI is an interface bus that can be used to send data between microcontrollers and small peripherals such as an ADC. This rate is chosen since it is twice the 22 kHz is the range of human hearing so 44.1 kHz is slightly higher than double for a sampling frequency.
The DAC must take the output from the microcontroller and convert it into an analog voltage that can be amplified in the next step. The DAC needs to be able to write to all the speakers at least at the rate of 44.1 kHz in order to ensure the entire array will receive its time delay and execute it in order to steer the sound lobe. An efficient way to do this is using a parallel-input DAC as opposed to a serial-input as parallel takes less time to write. Another amplifier is implemented in order to undo the DC bias and filter out any quantization noise. An op-amp is used to implement a low pass filter that is applied to each of the DAC channels while a high pass filter is designed to cancel out the DC-bias and lower the output impedance to improve the efficiency. Since the

speaker impedance is very low, a high impedance into it will create a large voltage drop, wasting energy.

The report also goes into detail about the configuration of the microcontroller and how it runs the system. The microcontroller in this design was set to have an interrupt every 44.1 kHz since that is the standard Nyquist rate for audio. Upon every interrupt, the microcontroller would sample the input audio and convert the analog voltage to a digital value. The microcontroller will then check the current time delay value and write to the DAC the sampled digital value with the added time delay in order to produce the correct output signal. In this design the current time delay value is simply controlled by a potentiometer which the user adjusts. This application needs a that microcontroller has enough ports for each channel and obviously has the capability to perform sampling and calculations this fast. That can be made easier with efficient coding but will certainly be something to be careful of.

2.2 Diffraction Theory

The Cornell design references diffraction theory and more specifically, N-slit diffraction, in obtaining the wave intensity equation. To better understand the principles and derivations behind that equation, research on diffraction [1] and multiple slit diffraction [5] was conducted. When looking at multiple slit diffraction, the basic rule is the peak intensity of a wave is proportional to the square of the number of slits and the more slits that are added will result in a more narrow maximum beam as well as a much higher maximum intensity. Certain parts of the derivation involved Huygen's Principle, so in an effort to fully understand the concepts that principle was investigated [7].

Huygens principle states every point on a wavefront (surface containing points affected in the same way by a wave at a given time) is a source of spherical wavelet (oscillation that starts at zero, increases, decreases, and returns to zero, "brief oscillation"). An example of this is two rooms connected by a doorway. If the speaker is radiating in the corner of one room, someone listening in the other room will think the sound is originating from the doorway. Amplitude at point Q which is a distance r_0 from origin point P_0 represented by:

$U(r_0) = \frac{U_0 e^{ikr_0}}{r_0}$ k = wavenumber which is equal to $\frac{2\pi}{\lambda}$

U_0 = starting amplitude

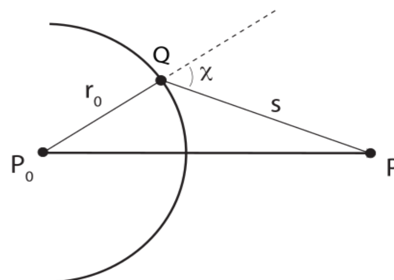


Figure 2.2.1 - Huygens Principle

To find amplitude at point P the contributions from all the points along the sphere are summed. Constants $-\frac{i}{\lambda}$ and K(x) are added in order for the following equation to hold true. Originally these constants were assumed but later emerged when Kirchoff derived diffraction. The first $-\frac{i}{\lambda}$ term is

needed because secondary waves will be a quarter out of phase with the primary wave and that the magnitude is equal to $\frac{1}{\lambda}$. K(x) is assumed to have a maximum value when x = 0, *K(x) = .5(1+cos(x)).*

N-Slit Diffraction Derivation:

Using Huygens principle an equation can be modeled for a monochromatic plane wave (general sine wave) passing through N slits and generates complex wave. The wave is modeled using huygens principle and represented as:

$\psi = \int_{slit} \frac{i}{r\lambda} \psi' e^{-ikr} dslit$, Where ψ' is the monochromatic wave

The distance r is derived to an equation dependent upon the xyz coordinates of both the slit and the distance from the slit. The equation is then modified to

$$r_j = z\left(1 + \frac{(x - x' - jd)^2 + y^2}{z^2}\right)^{\frac{1}{2}}$$

Using this equation the sum of the N contributions to the wave can be modeled using a summation of the complex amplitude from each slit. This is represented with the following equation.

$\Psi = \sum_{j=0}^{N-1} C \int_{-\frac{a}{2}}^{\frac{a}{2}} e^{\frac{ikx(x\prime - jd)}{z}} e^{\frac{-ik(x\prime - jd)^2}{2z}} dx'$, C is equivalent to $\Psi' \sqrt{\frac{i}{z\lambda}}$.

The above equation is then simplified to because the term $\frac{k(x\prime - jd)^2}{z}$ is considered very small so the second exponential term is equivalent to 1. Equation 1 below represents the modified result and equation 2 shows the same equation manipulated to be able to apply a specific identity.

Eq. 1: $\Psi = C \sum_{j=0}^{N-1} \int_{-\frac{a}{2}}^{\frac{a}{2}} e^{\frac{ikx(x\prime - jd)}{z}} dx'$, *Eq. 2:* $= aC \frac{sin\frac{ka\, sin\theta}{2}}{\frac{ka\, sin\theta}{2}} \sum_{j=0}^{N-1} e^{ijkd\, sin\theta}$

Using the identity below the summation can be substituted out so the second term is now a fraction similar to the first sine term. The identity will remove the 'j' term and take the remaining exponential and subtract it from 1. The result will be the divisor which is will divide a similar exponential only the 'j' is now replaced with the value of N. The equation is shown below to clarify. This will be the foundation of the wave intensity equation.

Identity: $\sum_{j=0}^{N-1} e^{xj} = \frac{1-e^{Nx}}{1-e^{x}}$ *Equation:* $\Psi = aC \frac{sin\frac{ka\, sin\theta}{2}}{\frac{ka\, sin\theta}{2}} \left(\frac{1-e^{iNkd\, sin\theta}}{1-e^{ikd\, sin\theta}}\right)$

Using trigonometry the equation is manipulated into sine functions and a remaining exponential. Euler's formula is used to convert the subtracting exponential equations to sine functions. The result is show below

$$\Psi = aC \frac{sin\left(\frac{ka\, sin\theta}{2}\right)}{\frac{ka\, sin\theta}{2}} \frac{sin\left(\frac{Nkd\, sin\theta}{2}\right)}{sin\left(\frac{kd\, sin\theta}{2}\right)} e^{i(N-1)kd\frac{sin\theta}{2}}$$

The last exponential is dropped as it is equivalent to e^0 which equals 1 due to $\left(e^{ix}|e^{ix}\right) = e^0$. By then squaring this result the equation for the intensity of the total wave in relation to the angle theta is found. The value of k defined above is substituted in as well as I_0 for all necessary

constants. The first term can be converted to a sinc function as that is defined as $\frac{sin(x)}{x}$. The equation is represented below.

$$I(\theta) = I_0 \left[sinc\left(\frac{\pi a}{\lambda} sin\theta\right)\right]^2 \cdot \left[\frac{sin\left(\frac{N\pi d}{\lambda} sin\theta\right)}{sin\left(\frac{\pi d}{\lambda} sin\theta\right)}\right]^2$$

In order to change the angle at which the maximum intensity will be found, a phase offset needs to be added to the fringe effect term or the $kd * sin(\theta)$. The phase offset can be manipulated to be controlled by a time delay since for any frequency a time delay is equal to $t_d = \phi/f$. This results in the following equation where $\phi = t_d/f$:

$$I = I_0 \left(\frac{sin(\frac{\pi a}{\lambda} sin\theta)}{\frac{\pi a}{\lambda} sin\theta}\right)^2 \left(\frac{sin\left(\frac{\pi}{\lambda} Nd\ sin\theta + \frac{N}{2}\phi\right)}{sin\left(\frac{\pi d}{\lambda} sin\theta + \phi\right)}\right)^2$$

The next step is to isolate theta in order to find what time delays will result a shift of a certain angle. Since only the second term is dependent on theta, the numerator of the second term is set to equal to its maximum value of 1 to find the maximum angle. This results in the equation $\theta = sin^{-1}(-\frac{\lambda}{2\pi d}\phi)$. Substituting λ for $\frac{v_s}{f}$ yields $\theta = sin^{-1}(-\frac{vs}{2\pi d}\frac{\phi}{f})$ which then becomes $\theta = sin^{-1}(-\frac{vs}{2\pi d}t_d)$ by substitution. This gives an equation that relates the time delay to the angle of maximum intensity that can be used to find correct delay in order to steer the wave in the desired direction. Using this model we can apply this to our audio output to control where the most sound intensity will be directed.

2.3 Hawksford Smart Loudspeaker

Designing a filter to apply to the audio signal is another approach to dynamically steering sound. The approach detailed in a report by Malcolm Hawksford [4] requires updateable FIR filters located between each speaker and the input signal. However, unlike most FIR filters, in this design the signal frequency will be considered constant and the sampling frequency will be dynamic. The overall approach is to design a number of filters for a discrete set of frequencies and then use interpolation to approximate the filter specifications for the remaining frequencies. Spline interpolation is recommended which involve constructing a piecewise polynomial using different low degree polynomials to find the best curve fit. The curve will represent the filter coefficients as a function of frequency and new coefficients will be interpolated in between the discrete set of calculated coefficients. Using this approach, one can ideally control both the beam width as well as the direction.

When designing the dynamic low pass filters, each is designed for a specific frequency. The cutoff frequency of these filters will affect only the width of the beam but not the angle in which the beam is directed. The first step in designing these filters is to determine the desired width of the beam. This width will affect the effective sampling rate of the filter. The following equation presented in the report shows the relationship between the beam width, L_x, and the effective sampling rate, f_{sx}.

$$f_{sx} = \frac{c}{g\ sin(\frac{L_x}{2})}$$

In the equation above, c represents the speed of sound and g represents the spacing between each speaker element. Using the value of f_{sx}, the cutoff frequency can be obtained using a rounding function ‘ceil’ which rounds up to the nearest integer. The following equation shows the relationship where N is the number of elements and ‘ncf’ is the harmonic cutoff frequency.

$$ncf = ceil\left(\frac{f}{f_{sx}}N\right) = ceil\left[0.5N\frac{f}{f_{high}}\frac{g}{g_{opt}}sin\left(\frac{L_x}{2}\right)\right]$$

A sinc function is then applied in order to obtain a rectangular window function in the frequency domain. The rectangular window is needed to smooth the FIR filter response.

The next aspect to designing this system is to investigate how to steer the output beam. The approach presented to do this requires introducing progressive time delays to each element in order to offset the beam from its original symmetrical position about the normal. To find the time delay for each element, path lengths need to be derived first. The figure below shows the path delays for a 4 element array where elements to the right of the center are positive numbers and negative numbers for elements to the left.

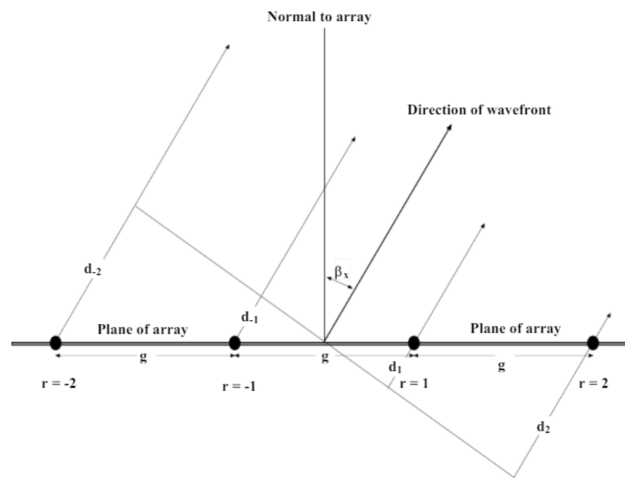


Figure 2.3.1 - Path Delays

The path length for element ‘r’ with a desired beam offset of □$_x$ can be represented by the following equation presented by Hawksford,

$$d_r = g(0.5 + r)\ sin\beta_x$$

Using these delay values the required time delay T_r, can be found. That value is then transferred into the frequency domain to obtain the transfer function of the delay. The delay transfer function differs slightly for elements to the right of the normal as opposed to the left of the normal as a simple negative sign is introduced. The time domain time delay equation as well as both frequency domain equations are shown below.

Time domain: $T_r = \frac{g(0.5+r)\ sin\beta_x}{c}$

Frequency domain, Left side: $delay_{rl} = exp\left[+j\frac{2\pi f}{c}(0.5 + r)\ sin\beta_x\right]$

Frequency domain, Right side: $delay_{rr} = exp\left[-j\frac{2\pi f}{c}(0.5 + r)\ sin\beta_x\right]$

Applying these time delays to signal will effectively steer it but it will cause undesired changes to the beam width as the angle changes. This is because as the steering angle changes, the inter element spacing, g, is now observed to be different than it was when the beam was centered about the normal. Therefore in order to maintain a constant beam width, this must be compensated for. This requires modifying the original harmonic cutoff frequency equation by simply changing 'g' to *'g*(cosβ*$_x$*).'*

To complete the process of designing these filters, a frequency selective mask 'scf' is applied in terms of the harmonic cutoff frequency. This mask is used to represent the ideal brick wall response of the filter which is depicted in figure 2.3.2 below.

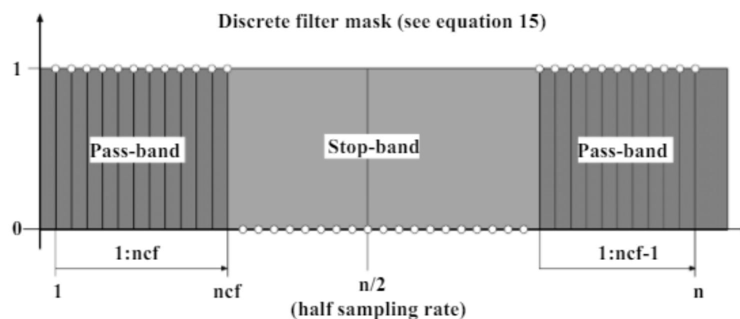


Figure 2.3.2 - Brick wall filter mask

The value ncf represents the cutoff harmonic which depends upon the selected frequency (f), the effective sampling frequency (f_{sx}), and the number of coefficients (N) which is also equivalent to the number of speaker elements. The equation for ncf was stated previously where "ceil" is a rounding function that will always round up to the next integer. This is the starting point for deriving the element channel transfer function (ECTF) filter that will apply both amplitude weighting and phase shifting for the selected frequency. Next, the fourier transform of the unit impulse which is equivalent to 1 is multiplied by the frequency mask and then the inverse fourier transform is applied to the result. The resulting time domain value is then multiplied by the rectangular window function defined above in order to limit the coefficients to the number of elements in the array.

In order to implement the desired offset, the calculated delay value for each element is multiplied by the corresponding coefficient. To clarify, the resulting filter will have the same number of coefficients as the array does speaker elements. Applying the rth delay requires that value to be multiplied by the rth coefficient in the vector. This will result in a filter that can beam steer a single frequency over a range of angles while keeping the beam width constant. This process is repeated for a discrete frequency set and then interpolated to approximate this process for the remaining frequencies in the desired bandwidth.

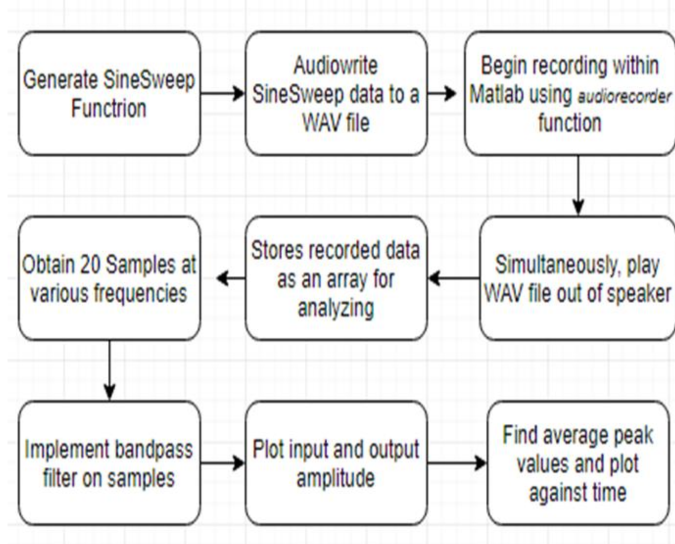
2.4 Audio Testing Software Options

Once we have a working prototype for the speaker, it will be important to perform audio testing on the speaker to measure its performance. We researched multiple software options for the

project as well as some open-source options. The front-runner for an affordable and reliable option for audio testing was a program called TrueRTA, a Real Time Audio Spectrum Analyzer. The website advertises the product as giving the user a “a detailed picture of you’re hearing in real-time”. All that is needed, besides the software, is a windows PC with just basic sound capability. For the purposes of our project a nice microphone or preamp would be useful in maximizing the potential for our product but is not essential to testing. TrueRTA includes multiple test instruments that could be useful to testing the sound quality of our speaker. Among many of the features, it comes with a High Resolution Real Time Analyzer, a Low Distortion Signal Generator, and an TrueRTA Oscilloscope.

The feature we felt would be the most helpful for the project is the High Resolution Real Time Analyzer. We was able to download the free version and while it lacked all the features of the paid version, we got great first-hand experience using the analyzer using different features including showing the distortion characteristics. With a speaker it would be able to measure the little intricacies in the sound quality and give us an idea of how it fares against competition. This audio spectrum analyzer can give us the changing spectrum, in real-time, of signals up to 48Khz which should be more than enough. This spectrum analyzer can also be used in conjunction with the signal generator.

The TrueRTA also has a QuickSweep feature which we felt would assist the team in measuring the sound quality of our speaker. Essentially what is does is give off a small, computer generated sweep (or a little chirp basically) in order to measure the frequency response of the acoustic system. You can mess around with the system by adding in a high pass filter and generating a test loop. Overall the design for the audio-testing software was ideally going to look something like the following diagram:



In addition to looking through existing software options for the audio testing, we also analyzed matlab options for a similar purpose which we analyzed as a comparison to the spectrum analyzer as well as a resource if we choose to edit these files to utilize in our project. The signal generator in the MatLab files might suffice with some tweaks, and that way we wouldn’t have to

purchase this add-on in TrueRTA. We were able to test with the Matlab files and it too has a spectrum analyzer function.

We then wanted to analyze and test Koohyar Pooladvand's code and see if it could be used for our project. The code first reads the audio file. This audio file might not be able to be in real time which could be the issue but it would read the file and the define the audio recording object.

```
info = audiodevinfo;
recorder1 = audiorecorder(48000,16,info.input(end).ID);
[f1, fs1] = audioread('f1.wav');
```

In this case the f1 and fs1 are inputs that had been imported into the work space so for the use case of this project we would need to get a recording of our speaker with a precise microphone. The program then returns a structure array containing all the input and output audio on the system.

```
if (Continious_paly) while Continious_paly
sound(y,sample_rate,nBits) Continious_paly=0;
Continious_paly=get(handles.Continuous,'value'); end
tic ;

FS = stoploop({'Stop me before', ' 60 seconds have elapsed'}) ;
% Display elapsed time
while(~FS.Stop() && toc < 60),      % Check if the loop has to be stopped
Start_button_Callback(hObject, eventdata, handles) tic
record(recorder1); sound(y,sample_rate,nBits) pause(duration)
toc
end
```

It then checks to see if the spectrum analyzer is on or not.

```
if  (Spectrum_status) signal = myRecObj(:,1);
```

If it is turned on, it begins to plot the spectrum of the recorded sound.

```
freq_range_rec=linspace(freq_min,freq,length(signal));
freq_range_sig=linspace(freq_min,freq,round(length(y(:,1))));
freq_range_rec=linspace(freq_min,freq,48000);
freq_range_sig=linspace(freq_min,freq,fs);
```

As you can see, similar to the TrueRTA Analyzer, the matlab code is made to give signals of up to 48Khz. After testing the Matlab code, there seemed to be some issues with it. Most of them were resolved after meeting with the code's author, however, that meeting wasn't able to be set up for some time so in the meantime we worked on our own code for plotting the frequency response of the speaker.

We was able to figure out how to accurately plot a wav file (for purposes of this project it would be a recording on the speaker). For this part you need to first import the data (the wave file) into your matlab workspace. You then need to change the directory to the one where the wavfile exists and call all the wav files in that folder.

The code below is listed in the appendix (Appendix B). The code utilizes the audioread function in order to read the data from the wave file. It then returns that sampled data, here known as "s" (for speaker), and a sample rate for the data it reads (fs).

```
[s,fs] = audioread('f4.wav');
```

We decided to utilize the fft command in order to eventually get the magnitude and phase responses plotted separately. Essentially what this function does is compute the discrete Fourier transform of the function defined (in this case "s") using the fft algorithm in matlab. We felt that utilizing the discrete Fourier transform would help because it is useful in finding the frequency components of signals buried in noise.

```
sdft = fft(s);
```

Then, under the assumption the y has an even length, we set up a frequency vector. Then, using the fs, or the sampling data that is read from the wav file, we can plot the magnitude using the frequency vector we utilized the subplot function in order create both figures side-by-side. The following 3 lines of code completed this:

```
sdft = sdft(1:length(y)/2+1);
freq = 0:fs/length(y):fs/2;
subplot(2,3,1);
```

Then, utilizing the same frequency vector from earlier, we were able to plot the phase response of the wav file. We utilized the unwrap function in order to do this. The unwrap function essentially corrects the phase angles and makes the phase plot cleaner by altering the phase angles.

```
plot(freq,abs(sdft));
subplot(2,2,2);
plot(freq,unwrap(angle(sdft)));
xlabel('Hz');
```

Another measure we felt would be useful is seeing the amplitude plotted against the time in the frequency. In order to do this we set up the function similar to how we did for the magnitude and the phase response. One thing that was different is that we needed to define the time increment. For this it was the seconds per sample (with that sample coming from the data received from importing the wave file into the matlab workspace).

```
dt = 1/fs;
t = 0:dt:(length(y)*dt)-dt;
```

In the matlab forums where we were researching about different measures of sound analysis, we came upon a great program that we felt might be useful for our project [9]. While some of the measures it creates aren't as useful for audio testing, the code acts as a signal analysis of any given sound file. Within the code we found multiple interesting features that we was able to play around with to have it work with the current version of matlab (the code was originally written on an earlier version on matlab).

The program begins by utilizing the audioread function in order to load in the audiofile. It then defines a couple of variables: the first channel and the length of that channel. Using these variables it creates a time vector which is then used to plot the signal waveform:

```
[x, fs] = audioread('f3.wav');
x = x(:, 1);
N = length(x);
t = (0:N-1)/fs;
```

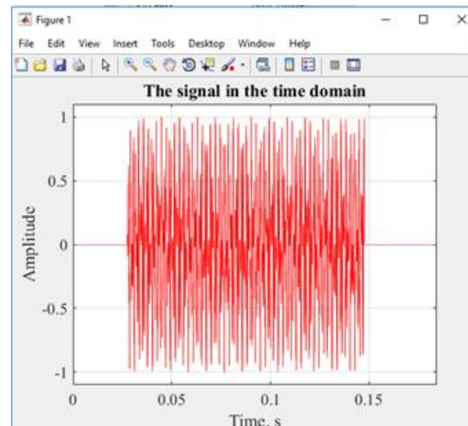


Figure 3: Signal Waveform

The program then plots the signal spectrogram. This is done using the spectrogram function. What the function does is essentially return a short-time domain Fourier transform of the given signal.

```
spectrogram(x,'yaxis')
```

The type of spectrogram function given means that it uses the actual frequency of the signal in hertz and the sampling frequency found from the wave file. In this case the signal is the variable x which was earlier defined as the first channel of the given wav file.

```
spectrogram(x, 1024, 3/4*1024, [], fs, 'yaxis')
```

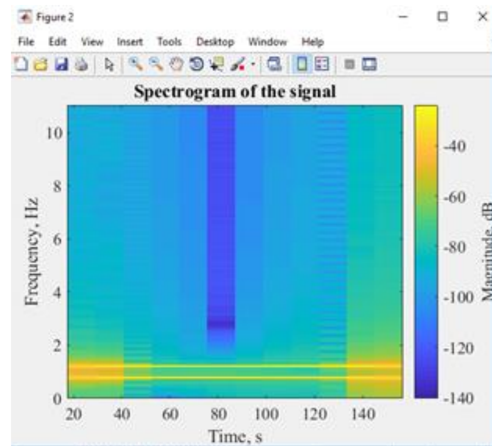


Figure 4: Signal Spectrogram

The next measure the program defines was the signal spectrum. For this spectrum, it plotted the magnitude in dB against the frequency in Hz. To do this the program utilized a function called semilogx. Essentially what the function does is create a plot based on given input values using a base 10 log scale as the x axis, and your choice of a linear scale for the y-axis. Below is the found amplitude spectrum of the wave file:

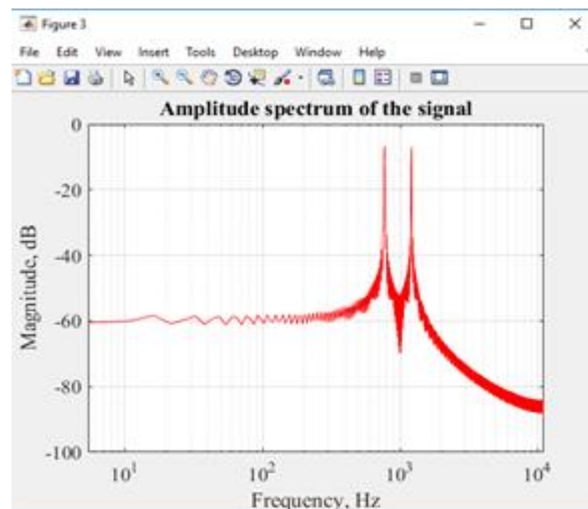


Figure 5: Amplitude Spectrum of Signal

3. Methods

3.1. MATLAB Simulations

The intensity equation which was derived from multiple slit diffraction needed to be tested to ensure that applying only a time delay will cause the main lobe to shift. Using MATLAB, a script was developed that would plot the intensity of a certain frequency wave against the listening angle. The resulting plots would describe the behavior of the wave intensity for the desired time delay. Two scripts were developed in order to observe a single frequency as well as how multiple frequencies are affected.

The first simulation is for applying a time delay to a single frequency and plotting the intensity of that sound wave. The MATLAB function that runs this takes 5 inputs to run. The five inputs are represented by ‘N’, ‘d’, ‘a’, ‘f’, and ‘td’ where ‘N’ is the number of elements, ‘d’ is the distance between adjacent elements, ‘a’ is the width of each element, ‘f’ is the desired frequency, and ‘td’ is the time delay applied. The function definition as well as an example function call are shown in the following two lines

Definition Function: `function I = IntensityPlot(N,d,a,f,td)`

Example Function Call: `IntensityPlot(6,.085,.05,1000,-0.0003)`

Inside the function, the first section is dedicated to declaring variables that will be needed later. The following four lines make up this section in the function:

```
v_s = 343.3
lambda = v_s/f;
theta = linspace(-90,90,10000);
phi = td/lambda*v_s;
```

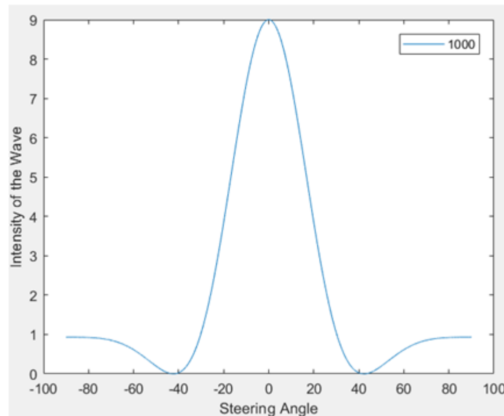
The first line declares ‘v_s’ to be the value of the speed of sound, 343.3 m/s. The second line converts the desired frequency from the function call to its corresponding wavelength. A vector from -90 to 90 called theta is created in the third line to represent the range of angles the wave intensity will be plotted over. Finally, the variable ‘phi’ is defined in the last line as the time delay divided by the wavelength multiplied by the speed of sound.

The function then begins to calculate the intensity of the desired sound wave using the wave intensity equation shown below which is derived from the principles of N-slit diffraction.

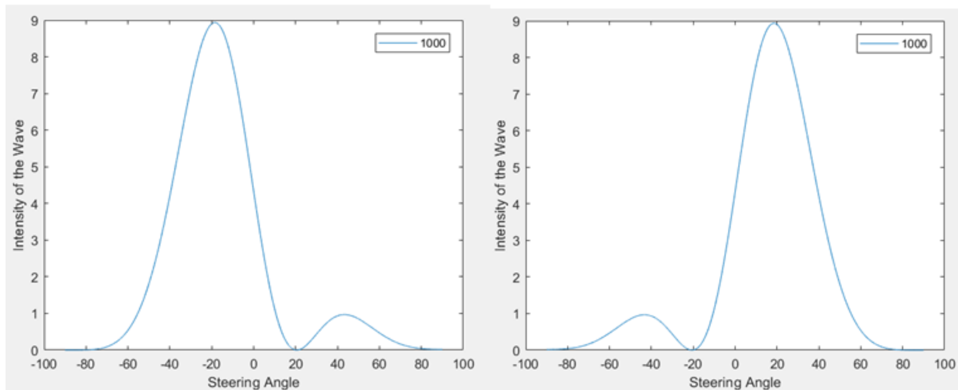
Since this equation is rather detailed, each main term was defined and those definitions were then used to calculate the full equation. Letters A and B represent the numerator and denominator of the first term and letters C, D, and E represent the second term. Using these letters the wave intensity can be calculated and stored in I. The MATLAB code for this implementation is shown below.

```
%Intensity equation -- Each letter equals a certain term
A = sin(pi*a.*sind(theta)/lambda);
B = pi*a.*sind(theta)/lambda;
C = ((2*pi*d.*sind(theta)/lambda)+phi);
D = sin((N/2).*C);
E = sin(C);
%Full equation
I = ((A./B).^2).*((D./E).^2);
```

The results of running this simulation show that a frequency can be steered in different directions due to a time delay. In the plots shown below a 1kHz frequency can be seen centered on zero when there is no time delay and shifted along the axis as a delay is applied. For these simulations the number of element was set to 6, the distance between elements was set to 0.085m, and the element width was 0.05m. As the number of elements is increased, the lobe will become more narrow and greater in amplitude than the side lobes. If the number of elements is set two 2 the frequency does not seem to shift no matter the time delay applied.



1khz wave, no time delay



1kHz wave, 0.0005s delay   *1kHz wave, -0.0005s delay*

The second script was a MATLAB function that modified the single frequency approach to show how the intensity plot of multiple frequencies with the same time delay changes. The function is very similar to the single frequency function as it takes the same 5 input arguments. The changes come in the function where now a lambda vector and a phi vector are created to store the corresponding value for each frequency being observed. A vector of zeros is also defined to store the intensity of each frequency. The vectors defined in MATLAB are shown below.

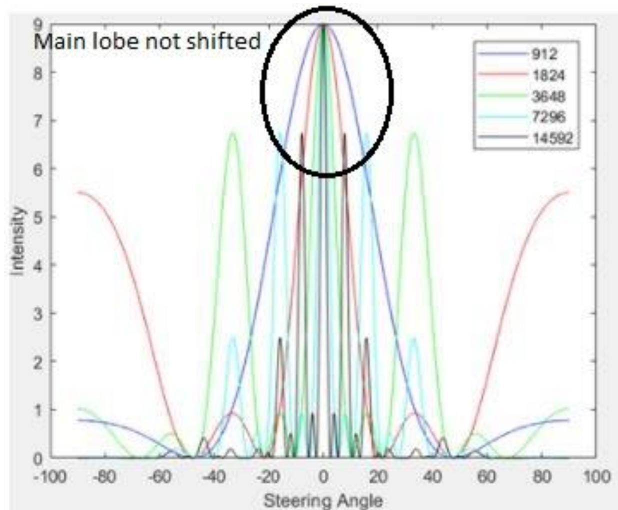
```
lambda_vect = [lambda;lambda/2;lambda/4;lambda/8;lambda/16];
phi =
[td/lambda_vect(1)*v_s;td/lambda_vect(2)*v_s;td/lambda_vect(3)*v_s;td/lambda_
vect(4)*v_s;td/lambda_vect(5)*v_s];
I = zeros(5,length(theta));
```

The lamba vector creates 5 waves to be plotted by dividing the ‘lamda’ value derived from the input frequency by 2, 4, 8, and 16. The phi vector then calculates the corresponding phi value for each wavelength.

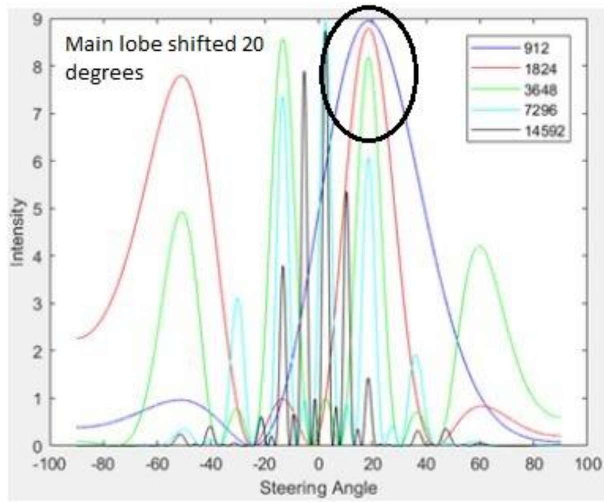
The calculation of the wave intensity is done the same way it is in the prior function except now it loops in order to calculate each frequency. In this case the loop was set to run five times since that is how many frequencies were being plotted. The implementation is shown below.

```
%loop to calculate for 5 frequencies
for k = 1:5
      %Intensity equation -- Each letter equals a certain term
      A = sin(pi*a.*sind(theta)/lambda_vect(k));
      B = pi*a.*sind(theta)/lambda_vect(k);
      C = ((2*pi*d.*sind(theta)/lambda_vect(k))+phi(k));
      D = sin((N/2).*C);
      E = sin(C);
      %Full equation
      I(k,:) = ((A./B).^2).*((D./E).^2);
End
```

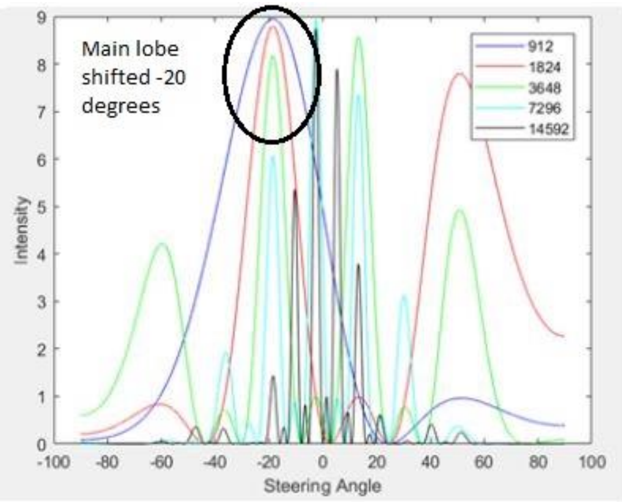
Each wave intensity that is calculated replaces a zero in the I vector defined above. The intensity for each frequency is then plotted against the angle vector on the same figure to show the position of the main lobe of each frequency. For the most part, the main lobe of the frequencies all seem to occur at the same angle, however the lobe location of higher frequencies seems to be more difficult to predict.The following plots show the main lobe of each frequency shift together along the axis when a time delay is applied and center of zero when no delay is present.



Multi-freq., no time delay



Multi-freq., -0.0005s delay



Multi-freq., 0.0005s delay

4. Implementation

4.1 Arduino Generated Sine Wave to One Speaker

The implementation process for this project went through numerous stages before the final product was reached. The first milestone was to construct a circuit using the Arduino Mega 2560, the TLC7528 8-bit DAC, a Sparkfun amplifier, and a speaker to play a sine wave generated in the Arduino out through the speaker. This required minimal work with the Sparkfun amplifier and the speaker driver as they just needed to be connected. However, for the Arduino, code needed to be written to generate the sine wave and also also needed to be compatible with the requirements of the DAC in order for the digital output to be converted to analog. A circuit for connecting the Arduino to the DAC input and then the output to the Sparkfun amplifier also needed to be designed and built. A simple block diagram for this implementation can be seen in figure 4.1.1

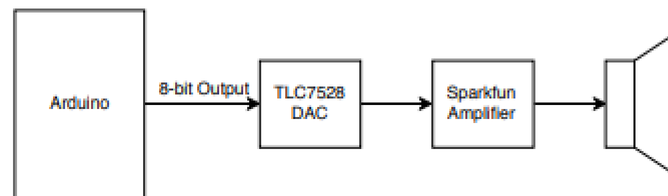


Figure 4.1.1 - Generated Sine Wave Single Speaker Block Diagram

The Arduino code that needed to be written for this stage would serve as a solid foundation that could be built upon later for more complex stages. The first task was to generate a sine wave internally so the DAC circuit could be tested without having to simultaneously test an audio input circuit. To do this, 100 values ranging from 0-255 were chosen in an order that started at 127 and progressively climbed to 255 and then progressively declined back down to 0 and back up again to 127. The buffer starts at 127 because a sine wave starts at 0 then oscillated between 1 and -1 so 127 is the middle point representing 0, 255 will represent +1, and 0 will represent -1. The reason this range of values of chosen is because our DAC is 8-bits meaning it is able providing a corresponding analog voltage for values up to 2^n where *n* represents the number of bits. With *n* being 8, that means 2^8 values can be converted or 256 so therefore values 0-255. The code for generating the sine wave is shown below

```
byte sine[] = {127, 134, 142, 150, 158, 166, 173,181, 188, 195, 201, 207,
213, 219, 224, 229, 234,238, 241, 245, 247, 250, 251, 252, 253, 254, 253,
252, 251, 250, 247, 245, 241, 238,234, 229, 224, 219, 213, 207, 201, 195,
188,181, 173, 166, 158, 150, 142, 134, 127, 119, 111, 103, 95, 87, 80, 72,
65, 58, 52, 46, 40, 34, 29, 24, 19, 15, 12, 8, 6, 3, 2, 1, 0, 0, 0, 1, 2, 3,
6, 8, 12, 15, 19, 24, 29, 34, 40, 46, 52, 58, 65, 72, 80, 87,95, 103, 111,
119, 126, 134, 142, 150, 158, 166, 173, 181, 188, 195, 201, 207, 213, 219,
224, 229, 234, 238, 241, 245, 247, 250, 251, 252, 253, 254, 253, 252,
               };
```

The next task was to configure a timer interrupt to grab a value from the sine buffer and output it at the proper rate. Since the common Nyquist rate for sampling audio is 44kHz, the timer interrupts were configured at a 40kHz frequency since that would align better with our ADC

configuration in the future. This results in a new sample or value from the internally generated sine buffer to be outputted every 22.7μs. This was done using the following block of code:

```
cli();
 // initialize Timer1 ~ 40kHz
 TCCR1A = 0;     // set entire TCCR1A register to 0
 TCCR1B = 0;     // same for TCCR1B
 // set compare match register to desired timer count:
 OCR1A = 399;
 // CTC mode on:
 TCCR1B |= (1 << WGM12);
 // Set CS10 bit for 1 prescaler:
 TCCR1B |= (1 << CS10);
 // enable timer compare interrupt:
 TIMSK1 |= (1 << OCIE1A);
 //enable global interrupts
 sei();
```

The first and last lines disable and enable global interrupts, respectively, in order for the interrupt to be configured. The timer interrupt in this program is configured to CTC mode or Clear Timer on Compare Match. This means that one of the Arduino Mega's internal timers is incremented until it reaches a specified value where it well then trigger an interrupt, reset back to 0 and begin counting again. By default, the arduino clock runs at 16MHz meaning each clock tick occurs every 63ns. However, using prescaler values, the clock ticks can be configured to happen slower therefore allowing the user to set slower interrupt frequencies. When setting a 40kHz timer interrupt, a prescaler of 1 was used meaning the clock would still tick every 63ns and a compare match register value of 399 was used to set an interrupt to trigger every 40kHz. The following equation was used to find the compare match register value.

*interrupt frequency (Hz) = (16,000,000Hz) / (prescaler * (compare match register + 1))*

Setting the prescaler is done in the line "`TCCR1B |= (1 << CS10);`" which in this case turns tells the arduino to use timer one with no prescaler or a value of 1. If a higher prescale value was desired, a combination of other bits such as `CS11` or `CS12` can be set to 1 in order to achieve this. The match value is set in the line "`OCR1A = 399;`" telling the timer to count 400 ticks then trigger an interrupt.

The next aspect of the program was setting the pins for chip select, write, and DACA as well as the pins which would output the 8-bit number to the DAC. The chip select pin used to signal when the DAC should start operating. When this is pulled low, the DAC will then look for which specific DAC, A or B, is being used since the chip in this design requires that. Lastly, the write pin is pulled low to allow data to be written to the DAC, in this case the values in the sine buffer. Each digital output pin on an arduino is capable of outputting either a '1'(high) or a '0'(low) so one 8 bit DAC will use 8 pins from the arduino to essentially have an 8-bit binary number given to it. The 8 pins on the arduino will individually be set to high or low to create the binary representation of the number between 0 and 255. The Arduino Mega has convenient feature

where specific groups of 8 pins can be referenced as a port when writing a program. Using this feature, the sine value can be outputted to a port as it indexes through the buffer as shown in the following line of code.

```
PORTK = sine[t&0x7F];
```

The variable ‘t’ is used to index through the buffer and is incremented after each value is outputted. The “`&0x7F`” is the modulo addressing used to prevent the index value ‘t’ from exceeding the total number of elements in the buffer which would result in an error. The sine buffer used in this program contains 128 elements and the value of 7F converted from hex to decimal is 127. When current value of the index t (0-127) is anded with the 7F value it will give the current index value as a result until the value of ‘t’ reaches 128 where it will not result in 0 when anded with the 7F. The pattern continues as the result of 129 & 0x7F is 1 and this is how the buffer will wrap around and avoid overflow. The full program for this stage can be found in Appendix 4.1.

After the code was written, the DAC circuit needed to be constructed. The DAC in this design contains 20 pins, in which pins 7 through 14 were dedicated to the 8-bit value being inputted. The DAC’s power supply was +5 volts and ground (GND). Pins 2, 3, and 17 were given +5 volts and pins 5 and 1 were connected to ground. Chip select(CS), write(WR), and DACA corresponded to pins 15, 16, and 6, respectively, and the DAC output to the sparkfun amplifier came from pin 4. A circuit diagram of the DAC circuit is shown in figure 4.1.2 below.

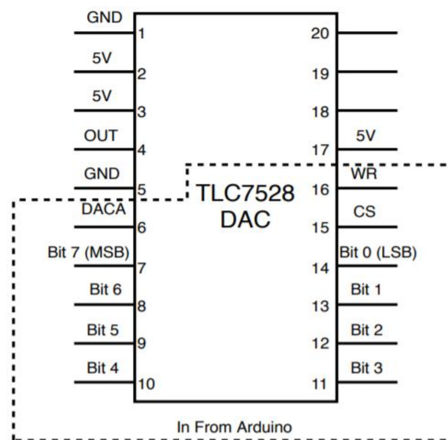


Figure 4.1.2 - DAC Circuit Diagram

<u>4.2 Arduino Generated Sine Wave to Six Speakers with Phasing</u>

After the generated sine wave was able to be played cleanly through one speaker, five more needed to be added in order to test the effectiveness of steering sound with time delays. In order to do this, 5 more sparkfun amplifiers and 5 more dac circuits needed to be built as well as modifications to the arduino program. The block diagram for this stage is shown in figure 4.2.1. It is essentially the same as the previous stage but has 5 more speakers and DACs as well as more processing in the Arduino.

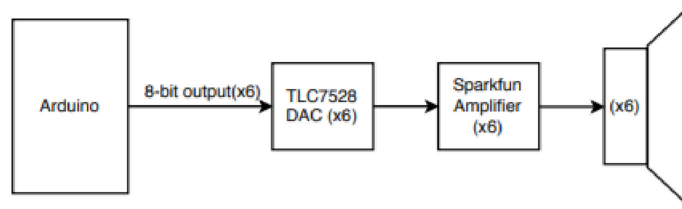
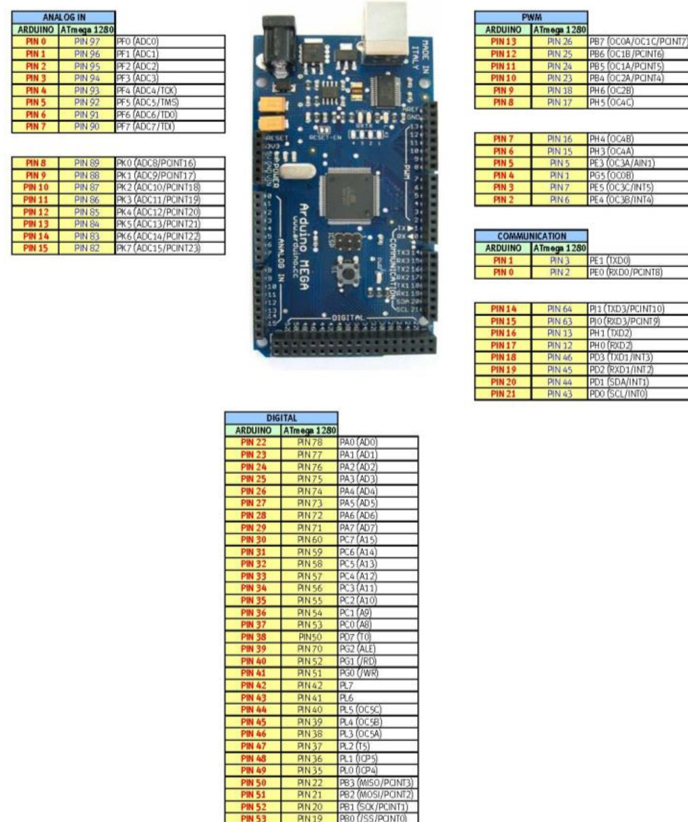


Figure 4.2.1 - Generated Sine Wave to Six Speakers Block Diagram

The main modification that needed to be added to the arduino code was adding five other ports to output to. Using figure 4.2.2 below, which detailed which 8 pins each port consisted of, the remaining 40 output pins were set. This was done by writing a for loop to iterate through the desired pins and set them to “OUTPUT”. An example of this is as following:

```
for (int i = 22; i < 54; i++)
{
  pinMode(i, OUTPUT);
}
```

The figure shows each pin number in yellow and which port it belongs to as well as what bit



ANALOG IN		
ARDUINO	ATmega 1280	
PIN 0	PIN 97	PF0 (ADC0)
PIN 1	PIN 96	PF1 (ADC1)
PIN 2	PIN 95	PF2 (ADC2)
PIN 3	PIN 94	PF3 (ADC3)
PIN 4	PIN 93	PF4 (ADC4/TCK)
PIN 5	PIN 92	PF5 (ADC5/TMS)
PIN 6	PIN 91	PF6 (ADC6/TDO)
PIN 7	PIN 90	PF7 (ADC7/TDI)
PIN 8	PIN 89	PK0 (ADC8/PCINT16)
PIN 9	PIN 88	PK1 (ADC9/PCINT17)
PIN 10	PIN 87	PK2 (ADC10/PCINT18)
PIN 11	PIN 86	PK3 (ADC11/PCINT19)
PIN 12	PIN 85	PK4 (ADC12/PCINT20)
PIN 13	PIN 84	PK5 (ADC13/PCINT21)
PIN 14	PIN 83	PK6 (ADC14/PCINT22)
PIN 15	PIN 82	PK7 (ADC15/PCINT23)

PWM		
ARDUINO	ATmega 1280	
PIN 13	PIN 26	PB7 (OC0A/OC1C/PCINT7)
PIN 12	PIN 25	PB6 (OC1B/PCINT6)
PIN 11	PIN 24	PB5 (OC1A/PCINT5)
PIN 10	PIN 23	PB4 (OC2A/PCINT4)
PIN 9	PIN 18	PH6 (OC2B)
PIN 8	PIN 17	PH5 (OC4C)
PIN 7	PIN 16	PH4 (OC4B)
PIN 6	PIN 15	PH3 (OC4A)
PIN 5	PIN 5	PE3 (OC3A/AIN1)
PIN 4	PIN 1	PG5 (OC0B)
PIN 3	PIN 7	PE5 (OC3C/INT5)
PIN 2	PIN 6	PE4 (OC3B/INT4)

COMMUNICATION		
ARDUINO	ATmega 1280	
PIN 1	PIN 3	PE1 (TXD0)
PIN 0	PIN 2	PE0 (RXD0/PCINT8)
PIN 14	PIN 64	PJ1 (TXD3/PCINT10)
PIN 15	PIN 63	PJ0 (RXD3/PCINT9)
PIN 16	PIN 13	PH1 (TXD2)
PIN 17	PIN 12	PH0 (RXD2)
PIN 18	PIN 46	PD3 (TXD1/INT3)
PIN 19	PIN 45	PD2 (RXD1/INT2)
PIN 20	PIN 44	PD1 (SDA/INT1)
PIN 21	PIN 43	PD0 (SCL/INT0)

DIGITAL		
ARDUINO	ATmega 1280	
PIN 22	PIN 78	PA0 (AD0)
PIN 23	PIN 77	PA1 (AD1)
PIN 24	PIN 76	PA2 (AD2)
PIN 25	PIN 75	PA3 (AD3)
PIN 26	PIN 74	PA4 (AD4)
PIN 27	PIN 73	PA5 (AD5)
PIN 28	PIN 72	PA6 (AD6)
PIN 29	PIN 71	PA7 (AD7)
PIN 30	PIN 60	PC7 (A15)
PIN 31	PIN 59	PC6 (A14)
PIN 32	PIN 58	PC5 (A13)
PIN 33	PIN 57	PC4 (A12)
PIN 34	PIN 56	PC3 (A11)
PIN 35	PIN 55	PC2 (A10)
PIN 36	PIN 54	PC1 (A9)
PIN 37	PIN 53	PC0 (A8)
PIN 38	PIN50	PD7 (T0)
PIN 39	PIN 70	PG2 (ALE)
PIN 40	PIN 52	PG1 (/RD)
PIN 41	PIN 51	PG0 (/WR)
PIN 42	PIN 42	PL7
PIN 43	PIN 41	PL6
PIN 44	PIN 40	PL5 (OC5C)
PIN 45	PIN 39	PL4 (OC5B)
PIN 46	PIN 38	PL3 (OC5A)
PIN 47	PIN 37	PL2 (T5)
PIN 48	PIN 36	PL1 (ICP5)
PIN 49	PIN 35	PL0 (ICP4)
PIN 50	PIN 22	PB3 (MISO/PCINT3)
PIN 51	PIN 21	PB2 (MOSI/PCINT2)
PIN 52	PIN 20	PB1 (SCK/PCINT1)
PIN 53	PIN 19	PB0 (/SS/PCINT0)

Figure 4.2.2 - Arduino Mega 2560 Pin/Port Layout

(0-7) in white. For simplicity sake, ports that were made up of consecutive pins were chosen when possible. The final ports selected were Port K, B, L, C and A as well as a port consisting of portions of E, G, and H using a bitmask in an effort to simplify the wiring. This port was referred to as Port EGH and the mask used to create it is shown in the code below.

```
PORTE = ((sine[(t)&0x7F] & 0x03) << 3) | (PORTE & (~0x38));
```

```
PORTG = ((sine[(t)&0x7F] & 0x08) << 2) | (PORTG & (~0x20));
PORTH = ((sine[(t)&0x7F] & 0xF0) >> 1) | (PORTH & (~0x78));
```

The mask works by using the and operator to only keep a specific section of the 8-bit value and then shifting it either left or right. The and operator works by comparing two bits from binary representations of the value in the buffer and the hexadecimal value 0x7F. If the bits are both "1" then the resulting bit will be a "1" as well. Any other combinations will result in a "0" being the output for that bit. A simple example of this can be seen below.

1 0 1 1 0 1
& 1 1 1 0 1 1
1 0 1 0 0 1

The values chose for the mask will set only the wanted pins to "1" and the rest to "0". The high pins will then be shifted using the << operator to put them in the proper place of the 8-bit output. When the three are combined they will form the 8-bit value which will be fed into the DAC using part of each port.

The next aspect of modifying the arduino code involved introducing the phasing. The basic concept behind this was introducing a time delay that became progressively longer as it went down the speaker array. Which speaker had the longest or shortest delay was dependent upon at which angle the sound was being steered. The initial delay ranged anywhere from -0.3ms to +0.3ms which would then get larger for the elements with the longer delays. This concept was implemented by controlling the sample each speaker would output. Since our system had an interrupt frequency of 40kHz, a new sample was played every 25µs. A port can then be told to output sample [t+4] while another port is told to just output sample [t] meaning the former port will be 4 samples or 0.1ms ahead of the latter port. Using simple multiplication, the specific number of samples to shift can be found for the full range of delay values. This value will be stored in a variable "td" which can be updated by changing the code, but will be capable of changing in real time in later implementations. The "td" value will be doubled, tripled and so on as it moves down the speaker array. The code implementing this can be seen below and the full code can be found in Appendix 4.2.

```
PORTK = sine[t&0x7F];
PORTB = sine[(t + td)&0x7F];
PORTL = sine[(t + td*2)&0x7F];
PORTC = sine[(t + td*3)&0x7F];
PORTE = ((sine[(t + td*4)&0x7F] & 0x03) << 3) | (PORTE & (~0x38));
PORTG = ((sine[(t + td*4)&0x7F] & 0x08) << 2) | (PORTG & (~0x20));
PORTH = ((sine[(t + td*4)&0x7F] & 0xF0) >> 1) | (PORTH & (~0x78));
PORTA = sine[(t + td*5)&0x7F];
```

The DAC circuit remained the same but just needed to be recreated 5 more times for the additional 5 speakers. The +5 volts and ground were run to a common rail and then connected to each DAC. Chip select, write, and DACA were daisy chained through all 6 DACs to allow them to share the same three pins as opposed to each DAC getting an individual set. When originally testing the six speaker output, three of the six channels sounded clean while the other three sounded very distorted. After debugging and troubleshooting both the code and the circuitry, a voltage leak was found at the DAC. The DAC should have been getting +5 volts however it was only measuring around 2.5 to 3 volts. The problem stemmed from loose connections that just needed to be found and adjusted due to the significant amount of wires in

the system. The next stage is to construct a circuit to input an analog waveform, either a simple sine wave or audio, into the Arduino.

4.3 Audio Input Circuit

The audio input circuit was the next feature that needed to be added in order to process actual analog waveforms that were being inputted as opposed to a digitally generated sine wave. The circuit shown in figure 4.3.1 depicts the circuit constructed for this stage. The op amp used is the TLC072CP which takes +9 volts and -9 volts. The input goes in to the non inverting pin of the op amp as well as a 10kΩ potentiometer to control the gain of the amplifier by varying the resistance. This can be adjusted up to increase the volume until the signal begins clipping. A 100kΩ is put in the feedback loop between the output pin and the inverting pin to control the output voltage based upon the equation $V_{out} = V_{in} * (R_2 / R_1)$ where R_2 is the 100kΩ resistor and the variable 10kΩ potentiometer is R_1. The next stage of the input circuit is the DC offset to

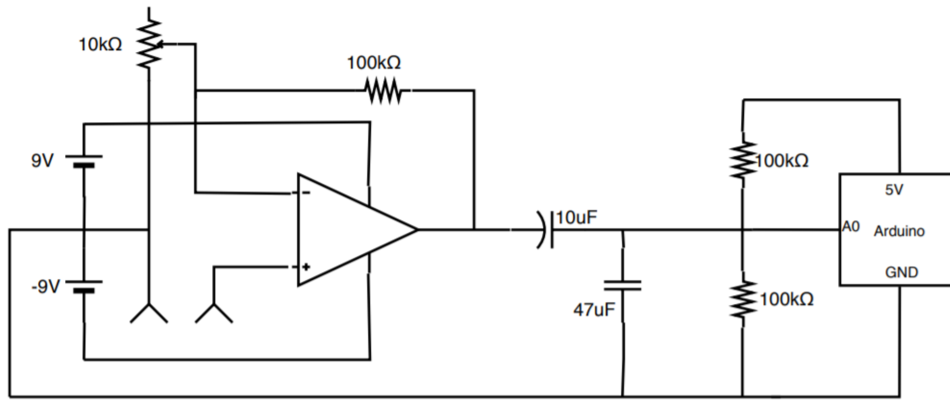


Figure 4.3.1 - Audio Input Circuit

shift the voltage up to be centered around 2.5V therefore giving the range of voltages to be 0 to 5V. The main components of the DC offset section are the capacitor and the voltage divider. The voltage divider is made up of two 100kΩ resistors connected in series from a 5V source meaing the voltage at the junction will be half of the input voltage. In this circuit, 5V comes in to the voltage divider and 2.5V is left at the junction. The 10μF capacitor that is connected to the junction and the output of the amplifier will have charge accumulate in it depending on the amplifier output. When this charge varies on one end of the capacitor it will repel from the side connected to the 2.5V junction causing it to vary up and down centered around 2.5V. A 47nF capacitor is also added between 2.5V and ground to reduce noise.

4.4 Configuring Arduino ADC

After the audio input circuit was constructed and working properly, the next step was to configure the ADC in the Arduino Mega to sample analog waveform and store the samples in a buffer. The samples need to be stored in a buffer in order for the same sample shifting concept discussed in section 4.2 to be utilized. The resulting block diagram for this section is shown in figure 4.4.1.

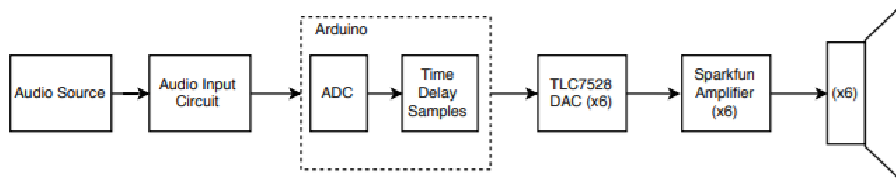


Figure 4.4.1 - Analog Audio Input with Six Speaker Output Block Diagram

The Arduino library contains a built in function called analogRead() which takes a pin number as the input and will sample the analog waveform connected to that pin. This function is acceptable for many applications however with the very high sampling rate required for audio, this was not sufficient. The analogRead function only has a sampling rate of 8kHz meaning it takes a sample every 125μs which is far too slow for the 25μs achieved using a 40kHz interrupt. To increase this sampling rate the analogRead function needed to be bypassed and the ADC needed to be configured manually. This was done in a similar way as the interrupts done in sections 4.1 and 4.2 when the ADC was not being used. The internal ADC counter was set to 500kHz and set to read off of analog input pin 0 (pin A0). The ADC takes 13 clock cycles to fully sample a new analog value so 500kHz clock speed was chosen because (*500kHz / 13) = 38.5kHz* which is close to our target of 40kHz. A sample rate of 38.5kHz will result in a sample being taken every 26μs. The reading from pin A0 would be converted into a 10-bit value as the Arduino's ADC is 10-bits. Since the DACs in this system are only 8-bit, only the most significant 8-bits of the 10-bit ADC reading are stored in the ADCH register and the last two are ignored. The code to configure the ADC is shown below. It contains similar concepts to configuring a timer interrupt as clears registers first and it uses prescalers to set the counter speed. It also left aligns the ADC value to ensure the most significant 8-bits are taken.

```
ADCSRA = 0;  //clear ADCSRA and ADCSRB registers
ADCSRB = 0;
ADMUX |= (1 << REFS0); //set reference voltage
ADMUX |= (1 << ADLAR); //left align the ADC value- read highest 8-bits
ADCSRA |= (1 << ADPS2) | (1 << ADPS0); //set ADC clock with 32 prescaler
ADCSRA |= (1 << ADATE); //enable auto trigger
ADCSRA |= (1 << ADIE); //enable interrupts when measurement complete
ADCSRA |= (1 << ADEN); //enable ADC
ADCSRA |= (1 << ADSC); //start ADC measurements
```

After the ADC is configured the output value will be contained in “ADCH”. In order to output this straight to the DAC, the code would simply be `PORTK = ADCH;` using port k as an example. However, the ADCH value will need to be stored in a buffer in order to use the sample shifting for phasing the speaker array. This is done by creating an index variable “data” similar to the index “t” described in section 4.1 used to iterate through the buffer. The index “data” will also iterate through the buffer, but will place the current ADCH value in that spot. The index value “t” is used to iterate through the buffer as well but it instead takes the value in that place of the buffer and outputs it through a port to the DAC. The code for this is only four lines and is shown below.

```
PORTC = sample[(t) & 0x7FF];
t++;
sample[data & 0x7FF] = ADCH;
Data++;
```

This code, which will also include five more port outputs, is all encompassed in an ADC interrupt. The code inside the ADC interrupt will run every time the ADC samples which is configured to 26μs. The other commands included in this interrupt is pulling the chip select, write, and DACA low to activate the DACs, then setting them back high after all the other commands have been done. A flow chart of the program is depicted in figure 4.4.2 and the full program code for this stage can be found in Appendix 4.4.

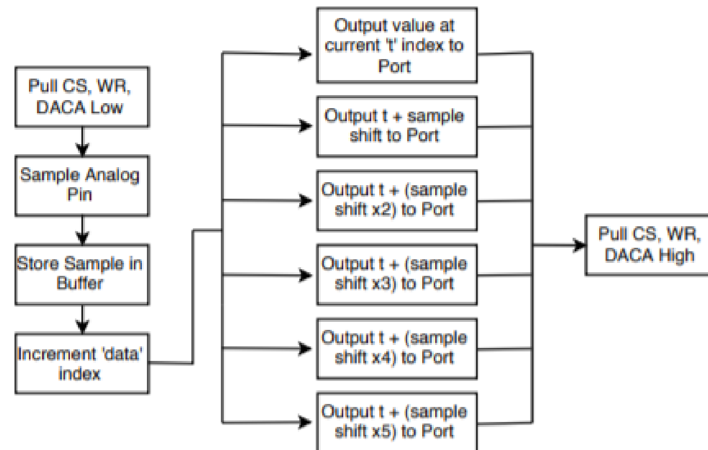


Figure 4.4.2 - Audio Processing Flow Chart

<u>4.5 Addition of Low Pass Filter After DAC</u>

The low pass filter addition was crucial as without it the speaker output was severely distorted and most of the time incomprehensible. Originally, this problem was thought to be related to the ADC not being able to sample fast enough for the audio. Then the idea of adding a simple filter to the output would help fix the issue. After some testing and looking at the fast Fourier transform (FFT) of both the clean audio from the source and the distorted signal out of the DAC, it could be seen that there was a significant amount of unwanted frequencies being outputted. The FFT oscilloscope plots from both the source and the DAC output can be seen in figures 4.5.1(a) and (b).

4.5.1(a) - Source Audio FFT *4.5.1(a) - DAC Output FFT*

When examining the two plots it can be seen that there is much more activity in the higher frequencies of the plot as opposed to at the source audio waveform. The low pass filter circuit shown in figure 4.5.1 was then implemented in the system after the DAC output. The filter

consists of the LT1115 Audio Op Amp and a basic RC filter where the resistor value is 3kΩ and the capacitor value is 0.01μF. The output of the filter will go into the Sparkfun amplifier which will then be played through the speaker.

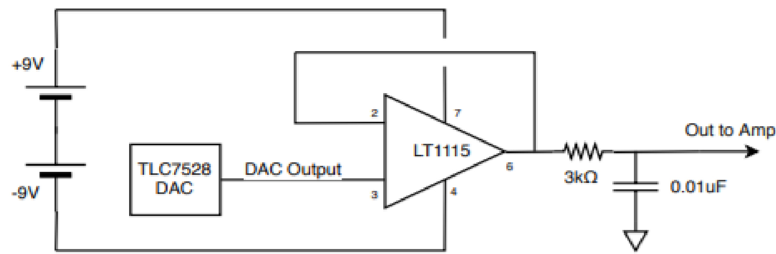


Figure 4.5.1 - Low Pass Filter Circuit

Applying the low pass filter greatly improved the sound output as there was no longer distortion present. The audio came out very clear with only a small amount of white noise in the background. A full block diagram of the entire audio system is provided in figure 4.5.2.

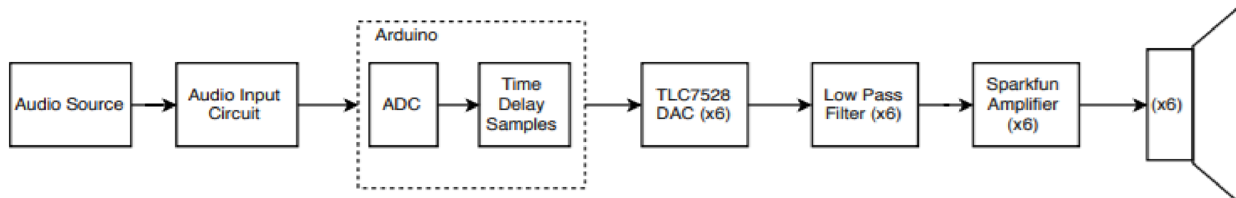


Figure 4.5.2 - Full Audio System Block Diagram

4.6 PIR Sensors

The PIR sensors are a crucial aspect of the project as this is how the time delay values for the phased array will be chosen. There will be 5 PIR sensors mounted on the top of the unit that will all be connected to an Arduino. An LED light will be mounted below them which will turn off when the sensor detects something in front of it. The LED light is not necessarily needed in the system however they are a useful visual to show when a sensor is turned on. The PIR sensor is powered by 5 volts, connects to ground through a 22kΩ resistor, and outputs to a TM082 op amp. The feedback loop contains a voltage divider comprised of two 1kΩ resistors effectively halving the output voltage that is fed back into the op amp to stabilize the gain. The output if the op amp is connected to the Arduino as well as the simple LED circuit to illuminate it when needed. The full PIR sensor circuit can be seen in figure 4.6.1 below.

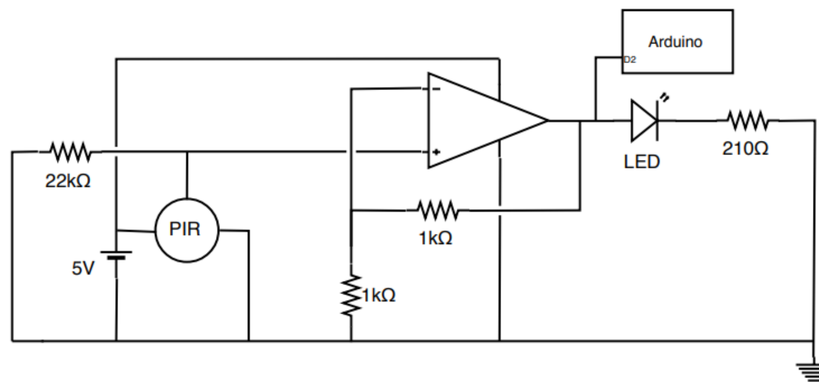


Figure 4.6.1 - PIR Sensor Circuit

Each sensor is connected to the master Arduino through a single digital pin, in this case digital pins 2 through 6. When the sensor detects a person in front of it, it will send a “low” signal to the Arduino. The Arduino will be reading each pin and when it receives a low signal from a pin it will set a variable “x” to the appropriate value of 0 through 5. This is done with a series of if statements that each compare the current state of one digital pins. When a pin is low, it enters into that statement which will tell the motors to move to a specific position and set the value of “x” to the appropriate value. This x value will be sent to the slave Arduino, using the built in wire library that allows connected Arduinos to communicate with each other. The two Arduinos are connected using 3 wires which connect the SDA pin (Communication pin 20) on each board, the SCL pin (Communication pin 21), and the ground.

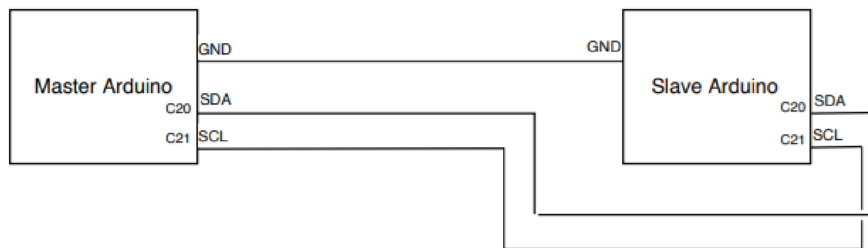


Figure 4.6.2 - Arduino Connection Diagram

Figure 4.6.2 above shows a simple diagram of how the Arduinos are connected physically. The SDA and SCL pins are where the information is sent and received between the slave and master. In the program, the master sends information to the slave using the following lines of code.

```
  Wire.beginTransmission(8); // transmit to device #8
  Wire.write(x);              // sends one byte
  Wire.endTransmission();    // stop transmitting
```

When this value is received at the slave Arduino, each x value will have its own corresponding number of sample shifts that will be applied to the output to create the time delay. A table describing the relationship between each sensor pin, x value, and the number of samples shifted in the buffer to create the time delay value is shown below in figure 4.6.2.

Sensor Pin	X	Samples
2	0	-40
3	1	-20
4	2	0
5	4	20
6	5	40

Figure 4.6.3 - Sensor Pin Table

In the table, there is no condition where x is equal to 3. This value is reserved for when none of the sensors are set to high so the speaker array will play the sound straight with no steering. The slave Arduino will also need code modifications as it will now have to read in the x value from the master Arduino. As stated above, this is done using the wire library that Arduino provides. The wire communication is initialized in the set up by choosing the bus and stating

what function should be called when the slave receives information from the master. The code is shown below.

```
 //Communication Setup
  Wire.begin(8);                // join i2c bus with address #8
  Wire.onReceive(receiveEvent); // register event
```

When the slave receives an event from the master it calls the receiveEvent function, which is shown below. This function accepts the number of how many sensors were on and then receives the x values as integers.

```
void receiveEvent(int howMany) {
  x = Wire.read();    // receive byte as an integer
  Serial.println(x);         // print the integer
}
```

The last step is to convert the received x value into the corresponding number of samples to shift in the buffer. This is done by comparing the current x value with the values in the table above using an if statement until the correct one is found. Once the value is found, the sample shift value “td” is then set with the corresponding value from the table. The if statements that perform this can be seen in the code below.

```
  if (x == 0)
    td = 0;
  else if (x == 1)
    td = -40;
  else if (x == 2)
    td = -20;
  else if (x == 3)
    td = 0;
  else if (x == 4)
    td = 20;
  else if (x == 5)
    td = 40;
  else
      td = 0;}
```

This value is then applied to the buffer being outputted to the DACs in order to properly delay the audio output. Both the master and the slave code can be found in their entirety in Appendix 4.6 and a simple diagram of how the Arduinos are connected can be seen below.

<u>4.7 Stepper Motors</u>

In order to manually steer two of the speakers we decided to utilize stepper motors to manually shift the speakers direction to point in the direction where motion was detected. For this we utilized the Unipolar Stepper Motor (28-BYJ48) which was a relatively inexpensive option that used only 5 volts. We also utilized the ULN2003 stepper motor driver which was compatible with out Arduino board. We wired this up to the BYJ48 Stepper and the Arduino in order to easily control the motor from the Mega. The following diagram helped us with the wiring of the stepper motor driver:

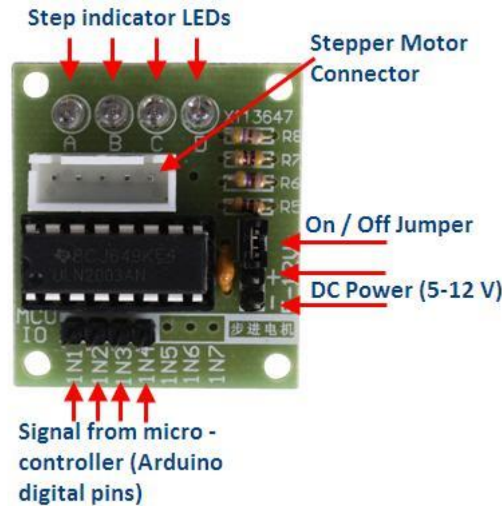
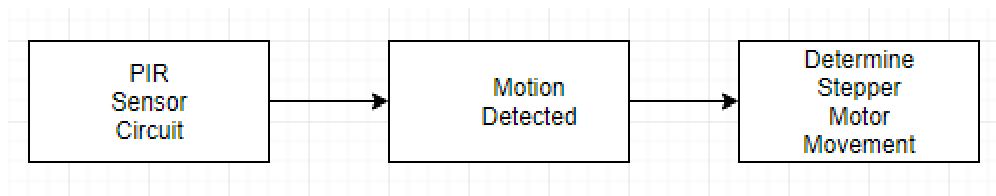


Figure 4.7.1 - ULN2003 Stepper Motor Driver board set-up

First we hooked up the stepper motor to the ULN2003 Driver on the side that also has the 4 LED's (top of the image). The LED's are used in order to indicate which of the coils are powered at the time. There were also a couple resistors and the side which takes the stepper motor on/off jumper which is for allowing power to the stepper motor. Our research found that it was not recommended to power the stepper from the Arduino itself so we used a battery pack instead for the power to the motor. The Arduino code we used, which can be found Appendix 4.6, first sets up and initializes the pin sequence for use with the stepper motor. The library we used for this was the AccelStepper library which runs the stepper motors most efficiently and had the least issues from most. It also moved the stepper motors in a reasonable pace, not jolting them back and forth or moving them too slowly. The following lines of code show how we incorporated the motion sensing into the stepper motor movement:

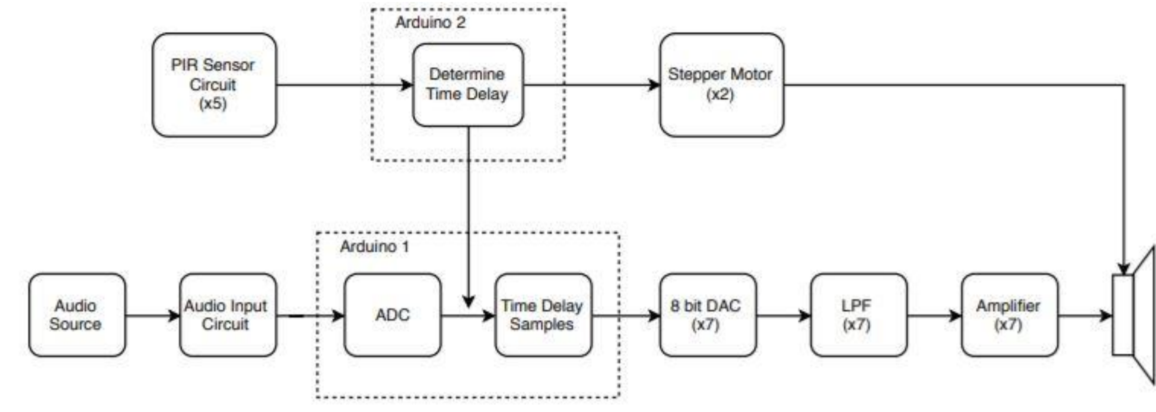
```
if (digitalRead(2) == HIGH)
 {
   stepper1.moveTo(-150);
   stepper2.moveTo(-150);
  /* while (stepper1.distanceToGo() != 0 && stepper2.distanceToGo() != 0)
    {
      stepper1.run();
      stepper2.run();
    }
```



4.7.2 - Block Diagram of Stepper Motor System

The stepper motors were the last feature to be added to the product. The full system now has an audio source and an input circuit to input an analog waveform the Arduino ADC. The

waveform is then sampled and a time delay is applied determined upon the output of the PIR sensor circuit into a second Arduino which then converts that value into the corresponding delay. The second Arduino also uses that delay to steer the stepper motors accordingly and the first Arduino delays the audio signal and outputs it to the six DACS which are then filtered, amplified, and played through the speakers. Figure 4.7.2 depicts the full system in a block diagram.



4.7.3
- Full System Block Diagram

5. Recommendations/Conclusions

This project as whole provided a good foundation for the signal processing needed to produce a phased array speaker. While many aspects of the project performed properly and reliably, there were many areas that could be improved in future design iterations that would result in a better final product.

The most notable feature that could be enhanced is the beam steering of the array. While the beam steering design that was implemented in this project worked sufficiently, more intricate and computationally heavy concepts could certainly improve the effect. As described in section 2.3, the Hawksford loudspeaker incorporates FIR filters that our designed depending on how many speakers are being used as well as frequency unique time delays as opposed to a time delay that encompasses a broad range of frequencies. The system uses FFTs to derive the filters for a discrete set of frequency values and then uses interpolation to approximate the values in between. An design like this proved to be too complex and also needed more processing power than an Arduino was capable of so it was not used in this iteration. A successful implementation of this could provide the user with much more control over the angle the sound beam is steered and can also provide control over the width of the main beam by applying advanced trigonometry principles.

The current beam steering implementation which is given variable time delays based upon the output of the PIR sensor circuit could also be improved by interpolating from the current delay value to the new one. Currently, when the time delay value being applied to the output signal changes there is a noticeable jump in the audio being played. While the main scope of this project was proving a phased array speaker system could be constructed and programed, it was also to provide a more enjoyable and immersive listening experience for the user. An audio system that has noticeable jumps as a person moves about the room is not ideal for a pleasant listening experience. Improving this would not be overly complex and could be done in multiple ways. One possible solution to this problem would simply be adding a function into the program to increment or decrement the delay value from the current value to the new updated value. The speed the delay value would be incremented would have to be determined using some trial and error and observing the quality of the audio as the delay gets changed. However, if this was implemented successfully the sound would gradually follow the listener ideally without them even noticing.

Other changes that would improve the project involve replacing the some of the hardware components with more powerful ones. The Arduino Mega could easily be replaced with a microcontroller that is designed to handle audio signals and has a much more processing power in order to manipulate the input signals more efficiently and effectively. A microcontroller geared for audio would allow a implementation similar the the Hawksford loudspeaker to become a possibility in future designs. Along with the microcontroller, and external ADC could be implemented using SPI in order to increase the sampling rate up to 44.1 kHz and the resolution on both the ADC and DACs could be increased to 12-bit or even 16-bit. This would allow for a much cleaner audio signal as the difference between quantization levels on a 0-5V scale would decrease dramatically. A 16-bit DAC would have 65536 quantization levels where an 8-bit DAC has only 256. Lastly, a printed circuit board would decrease the amount of noise

generated from wire connections and also eliminate the risk of a connection coming loose. This would be the next step after having perfboarded circuit work properly and consistently.

This project required research into phased arrays and signal processing in order to transfer that into the home speaker field and successfully steer sound. Sensors also needed to be studied as well as basic audio concepts in order to design circuits to identify listeners in the room and provide the proper time delay to the audio circuit. The final product this project produced has many improvements that can be made in future iterations but provides a framework and starting point for them.

6. References

1. Bouwkamp, Christoffel Jacob. "Diffraction theory." *Reports on progress in physics* 17, no. 1 (1954): 35.

2. Ghassaei, Amanda. “Arduino Audio Input.” *Instructables.com*, Instructables, 27 Oct. 2017, www.instructables.com/id/Arduino-Audio-Input/.

3. Ghassaei, Amanda. “Arduino Audio Output.” *Instructables.com*, Instructables, 26 Oct. 2017, www.instructables.com/id/Arduino-Audio-Output/.

4. Hawksford, Malcolm J. "Smart digital loudspeaker arrays." *Journal of the Audio Engineering Society* 51, no. 12 (2003): 1133-1162.

5. Mauyra, Amita. “Diffraction Due to N-Slits (Grating).” *Engineering Physics*, Engineering Physics, sites.google.com/site/puenggphysics/home/Unit-II/diffraction-due-to-n-slits.

6. Mert. “Arduino : How to Control a Stepper Motor With L293D Motor Driver.” *Instructables.com*, Instructables, 24 Sept. 2017, www.instructables.com/id/Arduino-How-to-Control-a-Stepper-Motor-With-L293D-/.

7. Pao, Yih - Hsing, and Vasundara Varatharajulu. "Huygens’ principle, radiation conditions, and integral formulas for the scattering of elastic waves." *The Journal of the Acoustical Society of America* 59, no. 6 (1976): 1361-1371.

8. Szoka, Edward, and Tom Jackson. “Phased Array Speaker System.” *Phased Array Speaker System*, Cornell University, people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/tcj26_ecs227/tcj26_ecs227/index.html.

9. Zhivomirov, Hristo. "Sound Analysis with Matlab Implementation." Sound Analysis with Matlab Implementation - File Exchange - MATLAB Central. February 18, 2017. Accessed April 24, 2018. https://www.mathworks.com/matlabcentral/fileexchange/38837-sound-analysis-with-matlab-implementation.

7. Appendix

<u>2.4 Audio Testing Matlab Code</u>

```
clear all

timing=5;
fs=48000; %sampling frequency
startfreq=20; %starting frequency
endfreq=1000;    %ending frequency

t=1/fs:1/fs:timing;
f=startfreq:(endfreq-startfreq)/length(t):endfreq-(endfreq-
startfreq)/length(t);

y = sin(2*pi*f.*t);

audiowrite('test100.wav', y, fs); %writesfile to the folder

info = audiodevinfo; %gets information about the input and output audio
devices (allows for program to play and record audio)

recorder1 = audiorecorder(48000,16,2); %creates a recording object, sample
rate of 48000, 16 bit and 2 channels

record(recorder1); %begins recording
sound(y,fs)%begins to play sound out of speaker
pause(timing)%recording stops after the variable timing - set earlier and
depends on how long the sweep signal is
stop(recorder1);%stops recording
myRecObj = getaudiodata(recorder1); %stores the recording and returns the
recorded audio data as an array which could be analyzed

filename = 'sweepaudiofile2.wav'; %sets file name for the recording of the
sound coming from the speaker
audiowrite( 'sweepaudiofile2.wav', myRecObj, fs); %writes the audiofile to
the current file

[y,fs] = audioread('sweepaudiofile2.wav'); %reads data from wave file and
returns that data as y array

%each of these segments calculates the range for samples at 10 different
%points throughout the sweep. Each range is for 20 cycles at every point
so
%the ending sample corresponds to whenever 20 cycles is done (roughly)

%first section of the audio signal
out = 1; %begin at sample 1
Tsamp = out*0.0002; %time is takes for 1
f2 = (Tsamp/0.0222)+ 20; %the 20 represents the number of cycles we want
to see, f2 is the starting frequency
out2 = out + (1/f2)*20*(4800); %ending sample number (calculated variable)
y1 = y(out:out2); %sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done
```

```
out3 = 5500;
Tsamp1 = out3*0.0002;
f3 = (Tsamp1/0.0222)+ 20; %starting frequency calculation
out4 = out3 + (1/f3)*20*(4800);%ending sample number (calculated variable)
y2 = y(out3:out4);%sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done

out5 = 11000;
Tsamp2 = out5*0.0002;
f4 = (Tsamp2/0.0222)+ 20;%starting frequency calculation
out6 = out5 + (1/f4)*20*(48000);%ending sample number (calculated
variable)
y3 = y(out5:out6);%sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done

out7 = 16500;
Tsamp3 = out7*0.0002;
f5 = (Tsamp3/0.0222)+ 20;%starting frequency calculation
out8 = out7 + (1/f5)*20*(48000);%ending sample number (calculated
variable)
y4 = y(out7:out8);%sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done

out9 = 23100;
Tsamp4 = out9*0.0002;
f6 = (Tsamp4/0.0222)+ 20;%starting frequency calculation
out_a = out9 + (1/f6)*20*(48000);%ending sample number (calculated
variable)
y5 = y(out9:out_a);%sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done

out_b = 28600;
Tsamp5 = out_b*0.0002;
f7 = (Tsamp4/0.0222)+ 20;
out_c = out_b + (1/f7)*20*(48000);%ending sample number (calculated
variable)
y6 = y(out_b:out_c);%sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done

out_d = 34100;
Tsamp6 = out_d*0.0002;
f8 = (Tsamp6/0.0222)+ 20;%starting frequency calculation
out_e = out_d + (1/f8)*40*(48000);%ending sample number (calculated
variable)
y7 = y(out_d:out_e);%sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done

out_f = 39600;
Tsamp7 = out_f*0.0002;
f9 = (Tsamp7/0.0222)+ 20;%starting frequency calculation
out_g = out_f + (1/f9)*40*(48000);%ending sample number (calculated
variable)
y8 = y(out_f:out_g);%sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done
```

```
out_h = 45100;
Tsamp8 = out_h*0.0002;
f_a = (Tsamp8/0.0222)+ 20;%starting frequency calculation
out_i = out_h + (1/f_a)*40*(48000);%ending sample number (calculated
variable)
y9 = y(out_h:out_i);%sets the range for the y, it goes from sample 1 to
whenever 20 cycles is done

%vectors created with the data for each of the 10 segments
yb = y(out:out2);
L = length(y1);
L2 = length(y2);
L3 = length(y3);
L4 = length(y4);
L5 = length(y5);
L6 = length(y6);
L7 = length(y7);
L8 = length(y8);
L9 = length(y9);
Lb = length(yb);
T = 1/fs;

%time vectors for each of the 10 segments
t = ((0:L-1)/fs);
t2 = ((0:L2-1)/fs);
t3 = ((0:L3-1)/fs);
t4 = ((0:L4-1)/fs);
t5 = ((0:L5-1)/fs);
t6 = ((0:L6-1)/fs);
t7 = ((0:L7-1)/fs);
t8 = ((0:L8-1)/fs);
t9 = ((0:L9-1)/fs);
tb = ((0:Lb-1)/fs);

%each of these calculate the ending frequency of the section that is
%displayed on the graphs/figures which pop up
Tsamp = out*0.0002;
f2 = (Tsamp/0.0222)+ 20

Tsampf = out2*0.0002;
ff = (Tsampf/0.0222)+ 20 ;

Tsampf2 = out4*0.0002;
ff2 = (Tsampf2/0.0222)+ 20 ;

Tsampf3 = out6*0.0002;
ff3 = (Tsampf3/0.0222)+ 20;

Tsampf4 = out8*0.0002;
ff4 = (Tsampf4/0.0222)+ 20;

Tsampf5 = out_a*0.0002;
ff5 = (Tsampf5/0.0222)+ 20;
```

```
Tsampf6 = out_c*0.0002;
ff6 = (Tsampf6/0.0222)+ 20;

Tsampf7 = out_e*0.0002;
ff7 = (Tsampf7/0.0222)+ 20;

Tsampf8 = out_g*0.0002;
ff8 = (Tsampf8/0.0222)+ 20;

Tsampf9 = out_i*0.0002;
ff9 = (Tsampf9/0.0222)+ 20;

%Filtering

NFFT=2^16;
fastf = fft(y1,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
Z=2*abs(fastf(1:NFFT/2));
[b,a]=butter(3,[f2,ff]/(fs/2),'bandpass'); %bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP = filter(b,a,y1);%implementation of filter and assigning it to seperate
array to be displayed as daya

NFFT=2^16;
fastf = fft(y2,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
Z=2*abs(fastf(1:NFFT/2));
[b2,a2]=butter(3,[f3,ff2]/(fs/2),'bandpass');%bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP2 = filter(b2,a2,y2);%implementation of filter and assigning it to
seperate array to be displayed as daya

NFFT=2^16;
fastf = fft(y3,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
Z=2*abs(fastf(1:NFFT/2));
[b3,a3]=butter(3,[f4,ff3]/(fs/2),'bandpass');%bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP3 = filter(b3,a3,y3);%implementation of filter and assigning it to
seperate array to be displayed as daya

NFFT=2^16;
fastf = fft(y4,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
Z=2*abs(fastf(1:NFFT/2));
```

```
[b4,a4]=butter(3,[f5,ff4]/(fs/2),'bandpass');%bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP4 = filter(b4,a4,y4);%implementation of filter and assigning it to
seperate array to be displayed as daya

NFFT=2^16;
fastf = fft(y5,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
Z=2*abs(fastf(1:NFFT/2));
[b5,a5]=butter(3,[f6,ff5]/(fs/2),'bandpass');%bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP5 = filter(b5,a5,y5);%implementation of filter and assigning it to
seperate array to be displayed as daya

NFFT=2^16;
fastf = fft(y6,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
Z=2*abs(fastf(1:NFFT/2));
[b6,a6]=butter(3,[f7,ff6]/(fs/2),'bandpass');%bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP6 = filter(b6,a6,y6);%implementation of filter and assigning it to
seperate array to be displayed as daya

NFFT=2^16;
fastf = fft(y7,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
Z=2*abs(fastf(1:NFFT/2));
[b7,a7]=butter(3,[f8,ff7]/(fs/2),'bandpass');%bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP7 = filter(b7,a7,y7);%implementation of filter and assigning it to
seperate array to be displayed as daya

NFFT=2^16;
fastf = fft(y8,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
Z=2*abs(fastf(1:NFFT/2));
[b8,a8]=butter(3,[f9,ff8]/(fs/2),'bandpass');%bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP8 = filter(b8,a8,y8);%implementation of filter and assigning it to
seperate array to be displayed as daya

NFFT=2^16;
fastf = fft(y9,NFFT)/NFFT;
f =fs/2*linspace(0,1,NFFT/2);
```

```
Z=2*abs(fastf(1:NFFT/2));
[b9,a9]=butter(3,[f_a,ff9]/(fs/2),'bandpass');%bandpass filter is designed
with a high and low cutoff frequency depending on the sweep range at
specific point

LP9 = filter(b9,a9,y9);%implementation of filter and assigning it to
seperate array to be displayed as daya

%each of these sections is a graph for both the input and output amplitude
%spectrum
figure(1)
subplot(2,1,1); % allows both graphs to be shown
plot(t,LP); grid; %plots the data from the post-filtered signal
title('Output Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f2),'   endfreq =' num2str(ff)]); %displays
starting and ending frequency

subplot(2,1,2);  % allows both graphs to be shown
plot(t,y1); grid;%plots the data from the pre-filtered signal
title('Input Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f2),'   endfreq =' num2str(ff)]);%displays
starting and ending frequency


figure(2)
subplot(2,1,1);
plot(t2,LP2); grid;
title('Output Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f3),'   endfreq =' num2str(ff2)]);

subplot(2,1,2);
plot(t2,y2); grid;
title('Input Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f3),'   endfreq =' num2str(ff2)]);

figure(3)
subplot(2,1,1);
plot(t3,LP3); grid;
title('Output Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f4),'   endfreq =' num2str(ff3)]);

subplot(2,1,2);
plot(t3,y3); grid;
title('Input Amplitude Spectrum')
```

```
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f4),'   endfreq =' num2str(ff3)]);

figure(4)
subplot(2,1,1);
plot(t4,LP4); grid;
title('Output Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f5),'   endfreq =' num2str(ff4)]);

subplot(2,1,2);
plot(t4,y4); grid;
title('Input Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f5),'   endfreq =' num2str(ff4)]);

figure(5)
subplot(2,1,1);
plot(t5,LP5);grid;
title('Output Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f6),'   endfreq =' num2str(ff5)]);

subplot(2,1,2);
plot(t5,y5);grid;
title('Input Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f6),'   endfreq =' num2str(ff5)]);

figure(6)
subplot(2,1,1);
plot(t6,LP6);grid;
title('Output Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f7),'   endfreq =' num2str(ff6)]);

subplot(2,1,2);
plot(t6,y6);grid;
title('Input Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f7),'   endfreq =' num2str(ff6)]);

figure(7)
subplot(2,1,1);
plot(t7,LP7);grid;
title('Output Amplitude Spectrum')
xlabel('Time (s)')
```

```
ylabel('Amplitude')
legend(['startfreq =' num2str(f8),'   endfreq =' num2str(ff7)]);

subplot(2,1,2);
plot(t7,y7);grid;
title('Input Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f8),'   endfreq =' num2str(ff7)]);

figure(8)
subplot(2,1,1);
plot(t8,LP8);grid;
title('Output Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f9),'   endfreq =' num2str(ff8)]);

subplot(2,1,2);
plot(t8,y8);grid;
title('Input Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f9),'   endfreq =' num2str(ff8)]);

figure(9)
subplot(2,1,1);
plot(t9,LP9); grid;
title('Output Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f_a),'   endfreq =' num2str(ff9)]);

subplot(2,1,2);
plot(t9,y9); grid;
title('Input Amplitude Spectrum')
xlabel('Time (s)')
ylabel('Amplitude')
legend(['startfreq =' num2str(f_a),'   endfreq =' num2str(ff9)]);


%zci = @(v) find(v(:).*circshift(v(:), [-1 0]) <= 0);                %
Returns Zero-Crossing Indices Of Argument Vector
%zx = zci(LP);
% Approximate Zero-Crossing Indices
%figure(5)
%plot(t, LP, '-r')
%hold on
%plot(t(zx), LP(zx), 'bp')
%hold off
%grid
%legend('Signal', 'Approximate Zero-Crossings')
%figure(3)
%plot(f,2*abs(LP(1:NFFT/2)))
```

```
%functions that find the peak of each of the values

pksb = findpeaks(yb,fs,'MinPeakDistance',0.005)
Mb = mean(pksb)

pks = findpeaks(LP,fs,'MinPeakDistance',0.005)
M = mean(pks)

pks2 = findpeaks(LP2,fs,'MinPeakDistance',0.005)
M2 = mean(pks2)

pks3 = findpeaks(LP3,fs,'MinPeakDistance',0.005)
M3 = mean(pks3)

pks4 = findpeaks(LP4,fs,'MinPeakDistance',0.005)
M4 = mean(pks4)

pks5 = findpeaks(LP5,fs,'MinPeakDistance',0.005)
M5 = mean(pks5)

pks6 = findpeaks(LP6,fs,'MinPeakDistance',0.005)
M6 = mean(pks6)

pks7 = findpeaks(LP7,fs,'MinPeakDistance',0.005)
M7 = mean(pks7)

pks8 = findpeaks(LP8,fs,'MinPeakDistance',0.005)
M8 = mean(pks8)

pks9 = findpeaks(LP9,fs,'MinPeakDistance',0.005)
M9 = mean(pks9)

%plots the peak values vs the frequency
x = [f2, f3, f4, f5, f6, f7, f8, f9, f_a]; %x values are the frequency
y = [M, M2, M3, M4, M5, M6, M7, M8, M9]; %y values are the peak values
figure(10);
plot(x,y,'x'); %plots the point with a 'x' as the point
title('Average Peak Value vs. Frequency')
xlabel('Frequency')
ylabel('Average Peak Value')
```

4.1 One Speaker Full Code

```
// avr-libc library includes
#include <avr/io.h>
#include <avr/interrupt.h>

//global variables
int t = 0;

//sine wave to output
byte sine[] = {127, 134, 142, 150, 158, 166, 173, 181, 188, 195, 201, 207,
```

```
213, 219, 224, 229, 234, 238, 241, 245, 247, 250, 251, 252, 253, 254, 253,
252, 251, 250, 247, 245, 241, 238, 234, 229, 224, 219, 213, 207, 201, 195,
188, 181, 173, 166, 158, 150, 142, 134, 127, 119, 111, 103, 95, 87, 80,
72, 65, 58, 52, 46, 40, 34, 29, 24, 19, 15, 12, 8, 6, 3, 2, 1, 0, 0, 0, 1,
2, 3, 6, 8, 12, 15, 19, 24, 29, 34, 40, 46, 52, 58, 65, 72, 80, 87, 95,
103, 111, 119, 126, 134, 142, 150, 158, 166, 173, 181, 188, 195, 201, 207,
213, 219, 224, 229, 234, 238, 241, 245, 247, 250, 251, 252, 253, 254, 253,
252,
               };
void setup()
{
  Serial.begin(115200);
  //set CS, WR, DACA to output
  pinMode(A5, OUTPUT);
  pinMode(A6, OUTPUT);
  pinMode(A7, OUTPUT);
  //set PORTK to ouput
  pinMode(A8, OUTPUT);
  pinMode(A9, OUTPUT);
  pinMode(A10, OUTPUT);
  pinMode(A11, OUTPUT);
  pinMode(A12, OUTPUT);
  pinMode(A13, OUTPUT);
  pinMode(A14, OUTPUT);
  pinMode(A15, OUTPUT);

 //disable global interrupts
  cli();
  // initialize Timer1 ~ 40kHz
  TCCR1A = 0;     // set entire TCCR1A register to 0
  TCCR1B = 0;     // same for TCCR1B
  // set compare match register to desired timer count:
  OCR1A = 400;
  // CTC mode on:
  TCCR1B |= (1 << WGM12);
  // Set CS10 and CS12 bits for 1024 prescaler:
  TCCR1B |= (1 << CS10);
  //TCCR1B |= (1 << CS12);
  // enable timer compare interrupt:
  TIMSK1 |= (1 << OCIE1A);
  //enable global interrupts
  sei();

ISR(TIMER1_COMPA_vect)
{
  //set CS, WR, DACA low
  digitalWrite(A5, LOW);
```

```
  digitalWrite(A6, LOW);
  digitalWrite(A7, LOW);
  //send sine wave to DAC, centered around (127/255)*5 = 2.5V
  PORTK = sine[t&0x7F];
  //set CS, WR, DACA high
  digitalWrite(A5, HIGH);
  digitalWrite(A6, HIGH);
  digitalWrite(A7, HIGH);
  //increment t
  t++;
}
```

<u>4.2 Full 6 Speaker Program</u>

```
// avr-libc library includes
#include <avr/io.h>
#include <avr/interrupt.h>

//global variables
int t = 0;
int td = 25;

//sine wave to output
byte sine[] = {127, 134, 142, 150, 158, 166, 173, 181, 188, 195, 201, 207,
213, 219, 224, 229, 234, 238, 241, 245, 247, 250, 251, 252, 253, 254, 253,
252, 251, 250, 247, 245, 241, 238, 234, 229, 224, 219, 213, 207, 201, 195,
188, 181, 173, 166, 158, 150, 142, 134, 127, 119, 111, 103, 95, 87, 80,
72, 65, 58, 52, 46, 40, 34, 29, 24, 19, 15, 12, 8, 6, 3, 2, 1, 0, 0, 0, 1,
2, 3, 6, 8, 12, 15, 19, 24, 29, 34, 40, 46, 52, 58, 65, 72, 80, 87, 95,
103, 111, 119, 126, 134, 142, 150, 158, 166, 173, 181, 188, 195, 201, 207,
213, 219, 224, 229, 234, 238, 241, 245, 247, 250, 251, 252, 253, 254, 253,
252,
               };
void setup()
{
  Serial.begin(115200);
  //set CS, WR, DACA to output
  pinMode(A5, OUTPUT);
  pinMode(A6, OUTPUT);
  pinMode(A7, OUTPUT);
  //set PORTK to ouput
  pinMode(A8, OUTPUT);
  pinMode(A9, OUTPUT);
  pinMode(A10, OUTPUT);
  pinMode(A11, OUTPUT);
  pinMode(A12, OUTPUT);
  pinMode(A13, OUTPUT);
```

```
  pinMode(A14, OUTPUT);
  pinMode(A15, OUTPUT);
  //set PORTE,G,H,B to output
  for (int i = 2; i < 14; i++)
  {
    pinMode(i, OUTPUT);
  }
  //set PORTA,B,C,L to output
  for (int i = 22; i < 54; i++)
  {
    pinMode(i, OUTPUT);
  }


 //disable global interrupts
  cli();
  // initialize Timer1 ~ 40kHz
  TCCR1A = 0;     // set entire TCCR1A register to 0
  TCCR1B = 0;     // same for TCCR1B
  // set compare match register to desired timer count:
  OCR1A = 400;
  // CTC mode on:
  TCCR1B |= (1 << WGM12);
  // Set CS10 and CS12 bits for 1024 prescaler:
  TCCR1B |= (1 << CS10);
  //TCCR1B |= (1 << CS12);
  // enable timer compare interrupt:
  TIMSK1 |= (1 << OCIE1A);
  //enable global interrupts
  sei();

ISR(TIMER1_COMPA_vect)
{
  //set CS, WR, DACA low
  digitalWrite(A5, LOW);
  digitalWrite(A6, LOW);
  digitalWrite(A7, LOW);
  //send sine wave to DAC, centered around (127/255)*5 = 2.5V
  PORTK = sine[t&0x7F];
  PORTB = sine[(t + td)&0x7F];
  PORTA = sine[(t + td*2)&0x7F];
  PORTL = sine[(t + td*3)&0x7F];
  PORTC = sine[(t + td*4)&0x7F];
  PORTE = ((sine[(t + td*5)&0x7F] & 0x03) << 3) | (PORTE & (~0x38));
  PORTG = ((sine[(t + td*5)&0x7F] & 0x08) << 2) | (PORTG & (~0x20));
  PORTH = ((sine[(t + td*5)&0x7F] & 0xF0) >> 1) | (PORTH & (~0x78));
```

```
  //set CS, WR, DACA high
  digitalWrite(A5, HIGH);
  digitalWrite(A6, HIGH);
  digitalWrite(A7, HIGH);
  //increment t
  t++;
}
```

4.4 Program with using ADC

```
//global variables
uint16_t t = 0;
uint16_t data = 0;
byte sample[2048];
int td = 25;


void setup() {

  Serial.begin(9600);
  //set WR, CS, DACA to output
  pinMode(A5, OUTPUT);
  pinMode(A6, OUTPUT);
  pinMode(A7, OUTPUT);

  //set PORTK to ouput
  pinMode(A8, OUTPUT);
  pinMode(A9, OUTPUT);
  pinMode(A10, OUTPUT);
  pinMode(A11, OUTPUT);
  pinMode(A12, OUTPUT);
  pinMode(A13, OUTPUT);
  pinMode(A14, OUTPUT);
  pinMode(A15, OUTPUT);

  //set PORTE,G,H,B to output
  for (int i = 2; i < 14; i++)
  {
    pinMode(i, OUTPUT);
  }

  //set PORTA,B,C,L to output
  for (int i = 22; i < 54; i++)
  {
    pinMode(i, OUTPUT);
  }

  cli();//disable interrupts
  //set up continuous sampling of analog pin 0
```

```
  //clear ADCSRA and ADCSRB registers
  ADCSRA = 0;
  ADCSRB = 0;
  ADMUX |= (1 << REFS0); //set reference voltage
  ADMUX |= (1 << ADLAR); //left align the ADC value
  ADCSRA |= (1 << ADPS2) | (1 << ADPS0); //set ADC clock with 32 prescaler
  ADCSRA |= (1 << ADATE); //enabble auto trigger
  ADCSRA |= (1 << ADIE); //enable interrupts when measurement complete
  ADCSRA |= (1 << ADEN); //enable ADC
  ADCSRA |= (1 << ADSC); //start ADC measurements
  sei();//enable interrupts
}

ISR(ADC_vect) {//when new ADC value ready
  //set CS, WR, DACA low
  digitalWrite(A5, LOW);
  digitalWrite(A6, LOW);
  digitalWrite(A7, LOW);

  PORTK = sample[t&0x7F];
  PORTB = sample[(t + td)&0x7F];
  PORTA = sample[(t + td*2)&0x7F];
  PORTL = sample[(t + td*3)&0x7F];
  PORTC = sample[(t + td*4)&0x7F];
  PORTE = ((sample[(t + td*5)&0x7F] & 0x03) << 3) | (PORTE & (~0x38));
  PORTG = ((sample[(t + td*5)&0x7F] & 0x08) << 2) | (PORTG & (~0x20));
  PORTH = ((sample[(t + td*5)&0x7F] & 0xF0) >> 1) | (PORTH & (~0x78));
  t++;//update the variable incomingAudio with new value from A0 (between 0
and 255)
  sample[data & 0x7FF] = ADCH;
  data++;

   digitalWrite(A5, HIGH);
   digitalWrite(A6, HIGH);
   digitalWrite(A7, HIGH);
}
```

4.6 PIR Sensor and Stepper Motor Code

Master

```
/*
   Team Epsilon:
   Mark, Corey, Avik, Suilio
   ----------------------------
   Sources:

http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/tcj26_
ecs227/tcj26_ecs227/index.html
   http://www.instructables.com/id/Arduino-Audio-Output/
```

```
   http://www.instructables.com/id/Arduino-Audio-Input/
   https://playground.arduino.cc/Code/Filters
   https://www.arduino.cc/en/Tutorial/MasterWriter
   http://42bots.com/tutorials/28byj-48-stepper-motor-with-uln2003-driver-
and-arduino-uno/
*/

#include <Wire.h>
#include <AccelStepper.h>
#define HALFSTEP 8

// Motor pin definitions
#define motorPin1  11     // IN1 on the ULN2003 driver 1
#define motorPin2  10     // IN2 on the ULN2003 driver 1
#define motorPin3  9     // IN3 on the ULN2003 driver 1
#define motorPin4  8     // IN4 on the ULN2003 driver 1

#define motorPin5  22     // IN1 on the ULN2003 driver 2
#define motorPin6  23     // IN2 on the ULN2003 driver 2
#define motorPin7  24    // IN3 on the ULN2003 driver 2
#define motorPin8  25    // IN4 on the ULN2003 driver 2

// Initialize with pin sequence IN1-IN3-IN2-IN4 for using the AccelStepper
with 28BYJ-48
AccelStepper stepper1(HALFSTEP, motorPin1, motorPin3, motorPin2, motorPin4);
AccelStepper stepper2(HALFSTEP, motorPin5, motorPin7, motorPin6, motorPin8);

// Transmission Variable
byte x = 0;

void setup() {
  Serial.begin(9600);

  // PIR pins
  for (int i = 2; i <= 6; i++)
  {
    pinMode(i, INPUT);
  }

  // Communication Setup
  Wire.begin(); // join i2c bus (address optional for master)
  stepper1.setMaxSpeed(1000.0);
  stepper1.setAcceleration(200.0);
  stepper1.setSpeed(200);
  stepper1.moveTo(0);

  stepper2.setMaxSpeed(1000.0);
  stepper2.setAcceleration(200.0);
  stepper2.setSpeed(200);
```

```
  stepper2.moveTo(0);

  while (stepper1.distanceToGo() != 0)
  {
    stepper1.run();
  }
  while (stepper2.distanceToGo() != 0)
  {
    stepper2.run();
  }
}

void loop() {

  Wire.beginTransmission(8); // transmit to device #8
  Wire.write(x);              // sends one byte
  Wire.endTransmission();    // stop transmitting
  Serial.println(x);

  delay(1000);

  if (digitalRead(2) == LOW)
  {
    stepper1.moveTo(-150);
    stepper2.moveTo(-150);
    x = 0;
  }
  else if (digitalRead(3) == LOW)
  {
    stepper1.moveTo(-90);
    stepper2.moveTo(-90);
    x = 1;
  }
  else if (digitalRead(4) == LOW)
  {
    stepper1.moveTo(-30);
    stepper2.moveTo(-30);
    x = 2;
  }

  else if (digitalRead(5) == LOW)
  {
    stepper1.moveTo(90);
    stepper2.moveTo(90);
    x = 4;
  }

  else if (digitalRead(6) == LOW)
  {
```

```
    stepper1.moveTo(150);
    stepper2.moveTo(150);
    x = 5;
  }

  else
  {
    stepper1.moveTo(30);
    stepper2.moveTo(30);
    x = 3;
  }
}
```

Slave

```
/*
   Team Epsilon:
   Mark, Corey, Avik, Suilio
   ----------------------------
   Sources:

http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/tcj26_ecs227/tcj26_ecs227/index.html
   http://www.instructables.com/id/Arduino-Audio-Output/
   http://www.instructables.com/id/Arduino-Audio-Input/
   https://playground.arduino.cc/Code/Filters
   https://www.arduino.cc/en/Tutorial/MasterWriter
   http://42bots.com/tutorials/28byj-48-stepper-motor-with-uln2003-driver-and-arduino-uno/
*/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <Wire.h>
//#include <Filters.h>

//global variables + buffers
int incomingAudio;
uint16_t t = 0;
uint16_t data = 0;
byte sample[2048];
int td = 15;
int x = 0;

void setup() {

  Serial.begin(9600);
  //set WR, CS, DACA to output
  pinMode(A5, OUTPUT);
  pinMode(A6, OUTPUT);
```

```
pinMode(A7, OUTPUT);

//set PORTK to ouput
pinMode(A8, OUTPUT);
pinMode(A9, OUTPUT);
pinMode(A10, OUTPUT);
pinMode(A11, OUTPUT);
pinMode(A12, OUTPUT);
pinMode(A13, OUTPUT);
pinMode(A14, OUTPUT);
pinMode(A15, OUTPUT);

//set PORTE,G,H,B to output
for (int i = 2; i < 14; i++)
{
  pinMode(i, OUTPUT);
}

//set PORTA,B,C,L to output
for (int i = 22; i < 54; i++)
{
  pinMode(i, OUTPUT);
}

//set CS, DACA, WR low
digitalWrite(A5, LOW);
digitalWrite(A6, LOW);
digitalWrite(A7, LOW);

cli();//disable interrupts

//set up continuous sampling of analog pin 0

//clear ADCSRA and ADCSRB registers
ADCSRA = 0;
ADCSRB = 0;

ADMUX |= (1 << REFS0); //set reference voltage
ADMUX |= (1 << ADLAR); //left align the ADC value
ADCSRA |= (1 << ADPS2) | (1 << ADPS0); //set ADC clock with 32 prescaler
ADCSRA |= (1 << ADATE); //enabble auto trigger
ADCSRA |= (1 << ADIE); //enable interrupts when measurement complete
ADCSRA |= (1 << ADEN); //enable ADC
ADCSRA |= (1 << ADSC); //start ADC measurements

sei();//enable interrupts

//Communication Setup
Wire.begin(8);                // join i2c bus with address #8
```

```
  Wire.onReceive(receiveEvent); // register event
  Serial.begin(9600);           // start serial for output
}

ISR(ADC_vect) {//when new ADC value ready

  PORTK = sample[t & 0x7FF];
  PORTB = sample[(t + td) & 0x7FF];
  PORTL = sample[(t + td * 2) & 0x7FF];
  PORTC = sample[(t + td * 3) & 0x7FF];
  PORTE = ((sample[(t + td * 4) & 0x7FF] & 0x03) << 3) | (PORTE & (~0x38));
  PORTG = ((sample[(t + td * 4) & 0x7FF] & 0x08) << 2) | (PORTG & (~0x20));
  PORTH = ((sample[(t + td * 4) & 0x7FF] & 0xF0) >> 1) | (PORTH & (~0x78));
  PORTA = sample[(t + td * 5) & 0x7FF];

  t++;
  sample[data & 0x7FF] = ADCH;
  data++;
}

void loop() {
  if (x == 0)
    td = 0;
  else if (x == 1)
    td = -40;
  else if (x == 2)
    td = -20;
  else if (x == 3)
    td = 0;
  else if (x == 4)
    td = 20;
  else if (x == 5)
    td = 40;
  else
      td = 0;
}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void receiveEvent(int howMany) {
  x = Wire.read();    // receive byte as an integer
  Serial.println(x);         // print the integer
}
```