FH | JOANNEUM
Elektronik und
Computer Engineering

Sound Laser

Ultrasonic beamforming using a parametric array

Bachelor's Thesis

written by:
Lukas Schüttler

at the bachelor's degree programme Elektronik und Computer Engineering
of the FH JOANNEUM – University of Applied Sciences, Austria

supervised by:
Florian Mayer

Graz, September 8, 2022

Obligatory declaration

I hereby confirm and declare that the present Bachelor's thesis was composed by myself without any help from others and that the work contained herein is my own and that I have only used the specified sources and aids. The uploaded version is identical to any printed version submitted.

I also confirm that I have prepared this thesis in compliance with the principles of the FH JOANNEUM Guideline for Good Scientific Practice and Prevention of Research Misconduct.

I declare in particular that I have marked all content taken verbatim or in substance from third party works or my own works according to the rules of good scientific practice and that I have included clear references to all sources.

The present original thesis has not been submitted to another university in Austria or abroad for the award of an academic degree in this form.

I understand that the provision of incorrect information in this signed declaration may have legal consequences.

Abstract

Parametric speakers are able to transmit audible sound with a beamforming effect using ultrasonic transducers. This work will describe the development of such a speaker as well as the software necessary to operate it.

In order to build this speaker it is explained how the beamforming effect occurs and how the number of transducers as well as distance between them influences its behaviour. Furthermore non-linear acoustics and why they can demodulate amplitude or frequency modulated signals are discussed.

Kurzbeschreibung

Parametrische Lautsprecher sind in der Lage, hörbaren Schall mit richtverhalten unter Verwendung von Ultraschalllautsprechern zu übertragen. In dieser Arbeit wird die Entwicklung eines solchen Lautsprechers sowie die, für seinen Betrieb erforderliche, Software beschrieben.

Um diesen Lautsprecher zu bauen, wird außerdem erklärt, wie der Beamforming-Effekt zustande kommt und wie die Anzahl der Lautsprecher sowie der Abstand zwischen ihnen sein Verhalten beeinflusst. Darüber hinaus wird auf nichtlineare Akustik eingegangen und erklärt, warum durch sie amplituden- oder frequenzmodulierte Signale demodulieren werden können.

Contents

List of Figures

List of Tables

Listings

1 Introduction

In communications technology antenna arrays are used to increase directivity of antennas and therefore amplify their reception power in the main radiation direction.[3] This technology can not only be used for electromagnetic waves but any kind of wave. Parametric arrays achieve a similar beamforming effect in acoustics which is utilised in non invasive medical imaging applications[4][5] or seismic profiling[6].

The aim of this thesis is to develop a speaker which generates a beam of audible sound with the use of a parametric array. In the following chapters the development of the speaker prototype as well as its acoustic behaviour will be explained respectively.

2 Concept

The basic idea of this project is to design a small module which can be combined to a larger speaker-array with improved beamforming characteristics. This will give the opportunity to compare different arrangements and numbers of modules. A single module is hexagon-shaped in order to be arranged in a honeycomb like style. Each module is equipped with an onboard DAC and amplification circuit.

Because of their size and their more focused dispersion characteristics, ultrasonic transducers are used in this project. How an audible sound can played over an ultrasonic transducer will be explained in section 3.2.

The direction of the beam is not controlled by phase shifting because, every transducer would need it's own DAC and amplifier. Alternatively the direction is altered by a mechanical contraption.

3 Theoretical Background

This section will explain the theoretical background of creating a beamforming speaker using ultrasonic transducers. Furthermore the results are used to predict the theoretical behaviour of the speaker in different configurations.

3.1 Beamforming

Beamforming is a concept used for emitting and receiving signals. It describes the technique of bundling a signal so it is only transmitted into or received from one direction. Because in this thesis beamforming is used with a speaker the following explanation will use an emitter as example but the calculations would work in exactly the same way for a receiver.[7]

In order to produce the beamforming effect for one dimension you need multiple emitters arranged in one line. If the receiver is far enough away from the emitters the directional vector of the waves can be viewed as parallel. When the receiver is positioned directly in front of the emitters ($0°$) all waves reach it at the same time (see figure 3.1.1). When the position of the receiver is rotated ($\pm\varphi°$) the waves have to travel different distances (See figure 3.1.2). Therefore, they reach the receiver with a phase shift. Depending oo the position of the receiver and distance of the emitters the waves will cancel each other out.

The difference in travel distance of two waves depending on the angle φ can be described as follows:

$$l(\varphi) = \sin\varphi \cdot d \tag{3.1.1}$$

Where d describes the spacing between the emitters.

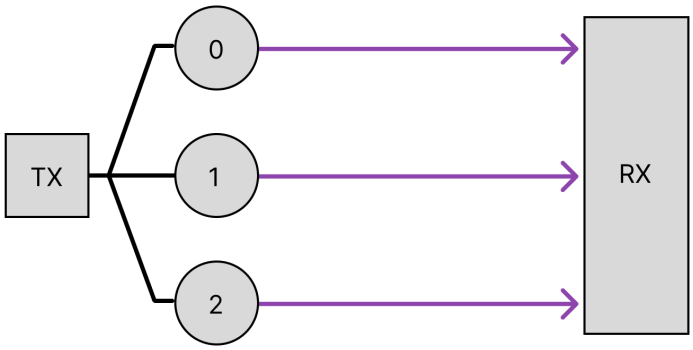


Figure 3.1.1: Phased array with the receiver positioned at 0°

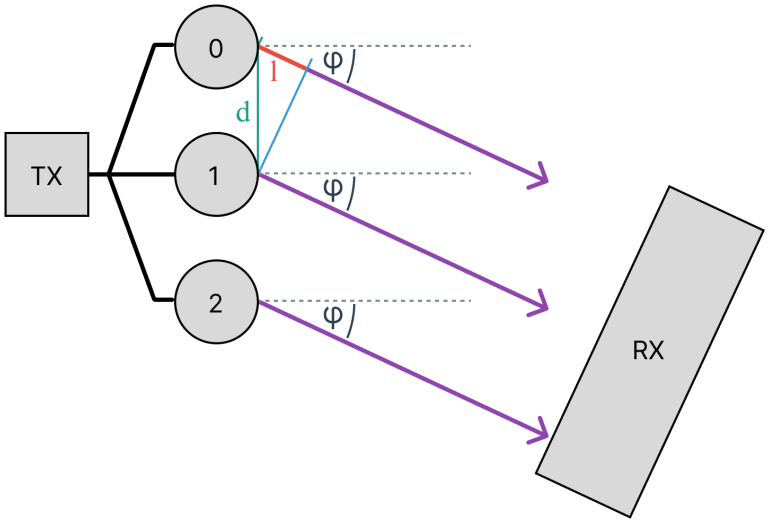


Figure 3.1.2: Phased array with the receiver positioned at 25°

Given the connection between distance d, time t, velocity v, wavelength λ and frequency f, this equation can be used to calculate the phaseshift Φ between those two waves.[8][9]

$$t(\varphi) = \frac{l}{v} = \frac{d}{v}\sin\varphi \tag{3.1.2}$$

$$\Phi(\varphi) = \omega \cdot t = \frac{\omega}{v} d \cdot \sin\varphi \tag{3.1.3}$$

$$\text{with } \lambda = \frac{v}{f} \implies \frac{\omega}{v} = \frac{2\pi \cdot f}{v} = \frac{2\pi}{\lambda} \tag{3.1.4}$$

$$\Phi(\varphi) = \frac{2\pi}{\lambda} d \cdot \sin\varphi \tag{3.1.5}$$

Equation 3.1.5 can also be used for a whole array of emitters, where $\Phi(\varphi)$ describes the phaseshift between two adjacent emitters. To get the phaseshift of the nth speaker relative to the first one $(n = 0)$ the phaseshifts $\Phi(\varphi)$ just need to be summed up.

$$\Phi(\varphi, n) = n \cdot \frac{2\pi}{\lambda} d \cdot \sin\varphi \tag{3.1.6}$$

Given a set of two spherical emitter generating a cosine wave $(s_e(t))$, the received signal $(s(t))$ would look as shown in figure 3.1.3. Figure 3.1.4 displays the result in a polar diagram.[10]

It can be observed, that the amplitude of the resulting signal is reduced the larger φ gets. Until both waves cancel each other out at 90°.

$$s_e(t, \varphi, n) = \cos(\omega t + n \cdot \frac{2\pi}{\lambda} d \cdot \sin\varphi) \tag{3.1.7}$$

$$s(t, \varphi) = \sum_{n=0}^{N} s_e(t, \varphi, n) \tag{3.1.8}$$

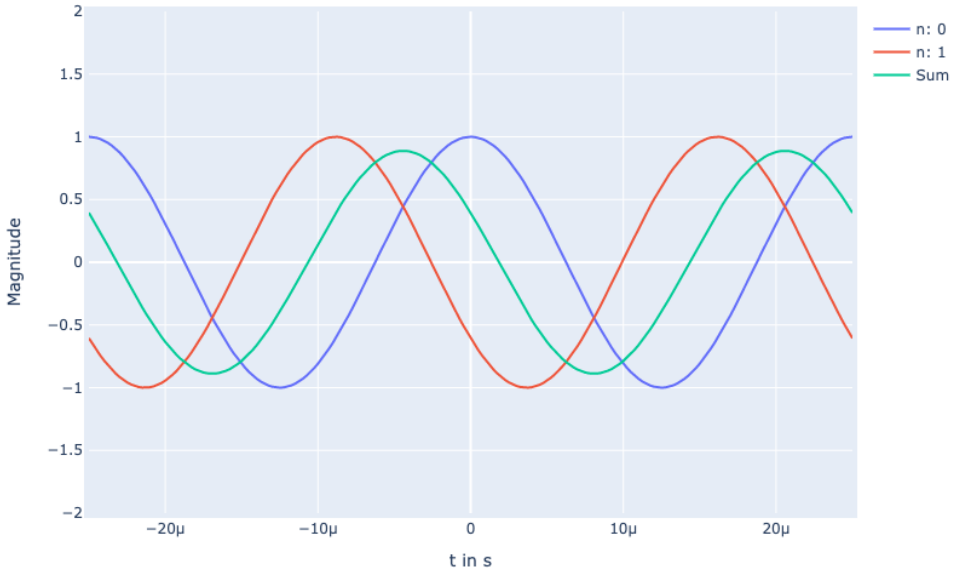


Figure 3.1.3: Signals of two emitters in a phased array ($\varphi = 45°$)

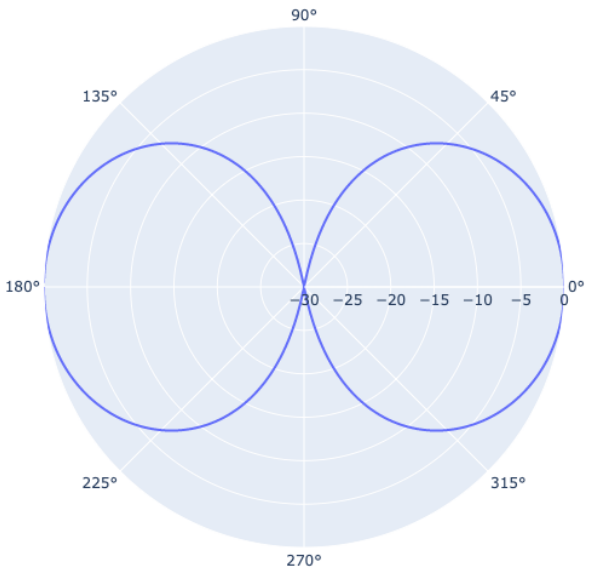
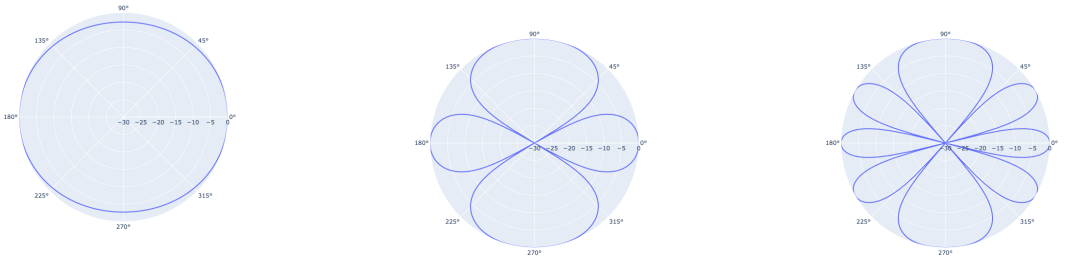


Figure 3.1.4: Phased array distribution characteristics ($d = \frac{1}{2}\lambda$, 2 emitters)

3.1.1 Distance between Emitters

The beamforming characteristics can easily be influenced by the distance between the emitters, hence any changes in the phaseshift. The ideal distance is half of the wavelength of the transmitted signal. With this spacing the two waves cancel each other out at exactly 90° as shown in figure 3.1.4.

Figures 3.1.5 and 3.1.4 show the beamforming characteristics of a phased array with 2 emitters and a distance of $\frac{1}{4}\lambda$, $\frac{1}{2}\lambda$, λ, and 2λ. The graph shows that using a smaller distance than $\frac{1}{2}\lambda$ makes the beamforming effect slowly disappear while a higher distance introduces a side lobe at ±90°. If the distance is further increased, the side lobes split up and a new one emerges at ±90°.



(a) $d = \frac{1}{4}\lambda$ **(b)** $d = \lambda$ **(c)** $d = 2\lambda$

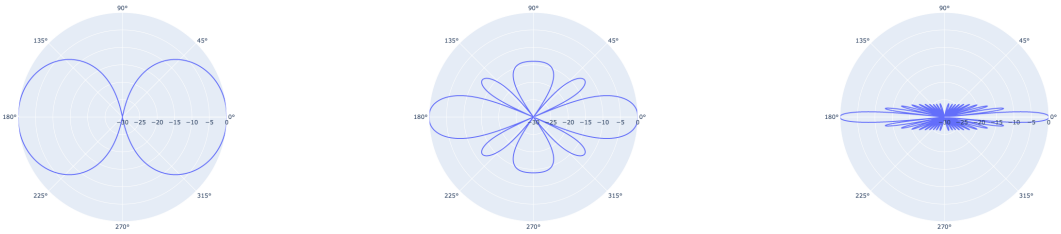
Figure 3.1.5: Beamforming characteristics depending on the distance between emitters

3.1.2 Number of Emitters

Figure 3.1.6 shows the beamforming characteristics of a phased array with 2, 5 and 20 emitters. The distance between the emitters is set to $\frac{1}{2}\lambda$. From the graphs it can be observed, that by increasing the number of emitters the generated beam gets narrower. However a higher number of emitters also introduces new side lobes with a smaller amplitude. The amplitude of the side lobes decreases the larger the number of emitters gets.

3.1.3 Dispersion Characteristics of Emitters

For the calculations above, the emitters are assumed to be isotropic radiators. In a real world scenario a speaker has some form of direction characteristic. The exact behaviour is highly dependent of the signal frequency. A higher frequency usually leads to more directivity.[11][12]



(a) 2 emitters **(b)** 5 emitters **(c)** 20 emitters

Figure 3.1.6: Beamforming characteristics depending on the number of emitters

3.2 Modulation

Modulation in electronics describes the process of altering parameters of a carrier wave depending on an input signal. This will transfer the frequency of the input signal so it can be transmitted for example via radio wave. The receiver further demodulates the signal to retrieve the original one.[13]

3.2.1 Modulation Techniques

AM

With the amplitude modulation the amplitude of the carrier signal $c(t)$ is changed depending on the modulating signal $s(t)$. The modulation depth m defines to which grade the amplitude is modified. With the parameter U_0 the offset of the modulating signal can be changed.

The AM is generally defined as follows:[13]

$$AM(t) = U_0 \cdot [1 + m \cdot s(t)] \cdot c(t) \tag{3.2.1}$$

Usually the carrier signal is a sine or cosine wave with the carrier frequency f_c.

$$AM(t) = U_0 \cdot [1 + m \cdot s(t)] \cdot \cos(2\pi \cdot f_c \cdot t) \tag{3.2.2}$$

FM

Using frequency modulation the frequency of the carrier signal $c(t, f)$ is changed by the modulating signal $s(t)$. The modulation depth m defines to which grade the frequency is modified. With the parameter U_0 the amplitude of the carrier is defined.

The FM is generally defined as follows:[13]

$$FM(t) = U_0 \cdot c(t, 2\pi \cdot f_c \cdot [1 + m \cdot s(t)]) \tag{3.2.3}$$

$$FM(t) = U_0 \cdot \cos(2\pi \cdot f_c \cdot [1 + m \cdot s(t)] \cdot t) \tag{3.2.4}$$

3.2.2 Demodulation

In order to recreate audible sound from the modulated signal it has to be demodulated. In the case of a parametric array inhomogeneous behaviour of a soundwave leads to the demodulation of AM as well as FM signals.

In linear acoustics it is presumed, that sound waves don't interact but just superimpose on each other. However in reality, when two sound waves collide, local changes in temperature occur resulting in an inhomogene behaviour of the transmission medium. Therefore, if two waves spread into the same direction their propagation is affected by each other. The resulting wave is described as an inhomogeneous wave equation by Westervelt (equation 3.2.5).[14][15]

$$\nabla^2 p_s - \frac{1}{c_0^2} \cdot \frac{\partial^2 p_s}{\partial t^2} = -p_0 \frac{\partial q}{\partial t} \tag{3.2.5}$$

$$q = \frac{\beta}{\rho_0^2 c_0^4} \cdot \frac{\partial}{\partial t} p_1^2 \tag{3.2.6}$$

p_1 and p_s are the sound pressure of the the primary and secondary wave, β is the nonlinear fluid parameter and c_0 is the sound velocity.

AM

Considering equation 3.2.2 the sound pressure of an AM signal can be described as shown in equation 3.2.7 and 3.2.8.

$$p_1(t) = p_0 \cdot (1 + m \cdot s(t)) \cdot \cos(2\pi \cdot f_c \cdot t) \tag{3.2.7}$$

$$p_1(t, r) = p_0 e^{-\alpha r} \cdot \left[1 + m \cdot s\left(t - \frac{r}{c_0}\right)\right] \cdot \cos\left(2\pi \cdot f_c \cdot \left[t - \frac{r}{c_0}\right]\right) \tag{3.2.8}$$

p_0 is the initial pressure, r is the distance from the speaker and $e^{-\alpha r}$ describes the damping of the wave along this distance.

p_1 is now inserted into equation 3.2.6.

$$q = \frac{\beta p_0^2}{\rho_0^2 c_0^4} e^{-2\alpha r} \cdot \frac{\partial}{\partial t} \left[1 + m \cdot s\left(t - \frac{r}{c_0}\right)\right]^2 \cdot \cos^2\left(2\pi \cdot f_c \cdot \left[t - \frac{r}{c_0}\right]\right) \tag{3.2.9}$$

$$\text{with } \cos^2(\varphi) = \frac{1}{2}[1 + \cos(2\varphi)] \tag{3.2.10}$$

$$q = \frac{\beta p_0^2}{\rho_0^2 c_0^4} e^{-2\alpha r} \cdot \frac{\partial}{\partial t} \left[1 + m \cdot s\left(t - \frac{r}{c_0}\right)\right]^2 \cdot \left[1 + \cos\left(4\pi \cdot f_c \cdot \left[t - \frac{r}{c_0}\right]\right)\right] \tag{3.2.11}$$

As the focus of the demodulation is audible sound all signals in the ultrasonic range are negligible.

$$q = \frac{\beta p_0^2}{\rho_0^2 c_0^4} e^{-2\alpha r} \cdot \frac{\partial}{\partial t} \left[1 + m \cdot s\left(t - \frac{r}{c_0}\right)\right]^2 \tag{3.2.12}$$

If this solution is now inserted into equation 3.2.5 it can be solved for p_s. The result is known under the Berktay far-field solution[16].

$$p_s(t) = \frac{\beta p_0^2 S}{16\pi \rho_0^2 c_0^4 \alpha_0 r} \frac{\partial^2}{\partial t^2} E^2(t - \frac{r}{c_0}) \tag{3.2.13}$$

Where S is the area of the parametric array and $E(t - \frac{r}{c_0})$ is the function envelope of $p_1(t, r)$. Equation 3.2.14 shows the demodulated AM signal using the Berktay far-field solution. The demodulation results not only in the original signal but distortion from the interaction between lower and upper sideband. However with a modulation depth $m < 1$ the distortion will always be smaller than the signal.

$$p_s(t) = \frac{\beta p_0^2 S}{16\pi \rho_0^2 c_0^4 \alpha_0 r} \frac{\partial^2}{\partial t^2} \left[2m \cdot s\left(t - \frac{r}{c_0}\right) + m^2 \cdot s^2\left(t - \frac{r}{c_0}\right)\right] \tag{3.2.14}$$

FM

Using the same approach as with the AM signal, the demodulation of the FM signal can be calculated as well. Having said this the process is mostly ignored when talking about parametric speaker arrays and therefore information about it is quite rare. Usually the effect is only verified empirically. The paper of Masato Nakayama and Takanobu Nishiura gives an idea on how the mathematical solution might look.[17]

3.2.3 Bandwith

AM

The bandwith calculation of an amplitude modulated signal is straightforward as the modulation just creates an upper and lower sideband at $f_c \pm f_s$.[13] Where f_c is the carrier frequency and f_s is the signal frequency.

As the highest tone noticeable by a human ear is at about $20kHz$, this would result in a maximum bandwith of $f_c \pm 20kHz$. With a carrier frequency of $40kHz$ the highest frequency of the modulated signal would lie at $60kHz$. Reducing the highest tone to $8kHz$, which is sufficient for most types of audio would reduce this value to $48kHz$.

FM

In theory the frequency spectrum of a frequency modulated signal is infinite with decreasing pulses at $f_c \pm n \cdot f_s$. However the bandwith can be limited without causing significant damage to the original signal.[13]

$$B = 2(\triangle f + f_s) \qquad \text{with } \triangle f = m \cdot f_c \tag{3.2.15}$$

With a modulation depth of 0.05, a carrier frequency of $40kHz$ and a maximum signal frequency of $20kHz$ this would result in a bandwith of $44kHz$. With a maximum signal frequency of $8kHz$ this would be reduced to $20kHz$. The highest frequency of the modulated signal would therefore be $50kHz$.

4 Circuit and PCB Design

Using the results from the last section the circuit for the individual modules has been designed.

4.1 DAC

The DAC in the Circuit is used to generate an analog voltaged signal from the signal recorded and modulated by a Raspberry Pi or microcontroller.

As described in section 3.2 the frequency of the modulated signal goes up to $50Hz$. In order to comply with the sampling theorem, the sampling frequency of the DAC should be at least $100kHz$. Therefore the digital interface has to bee fast enough, even if up to 7 modules are connected at the same time. Additionally, the DAC should work with $5V$ supply voltage and an internal reference voltage.

I2C only allows a maximum clockspeed of $5MHz$ (Ultra Fast-mode).[18] The length of a $10bit$ or $12bit$ DAC sample is usually about $32bit$ ($8bit$ Address + RW, $8bit$ config, $10-12bit$ Data, $4-6bit$ Don't care). As shown in equation 4.1.1 this only allows a sampling rate up to $156.25kHz$ even if only one module is connected. Because every module needs to be addressed individually, this sampling rate would drop to $22.321kHz$ when using all seven modules. For this reason SPI was selected as serial interface. The clock frequency of the SPI module is only limited by the system clock and the receiver is selected by a separate chip select pin which reduces the packet size of one sample to $24bit$. Because every module gets the same signal one CS pin for all DACs can be used.

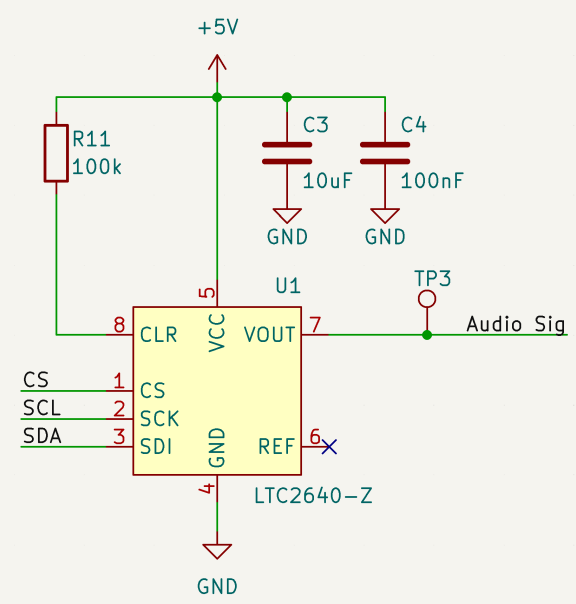


Figure 4.1.1: DAC Circuit design

$$f_{s,1} = \frac{5MHz}{32bit} = \frac{5\frac{Mbit}{s}}{32bit} = 156.25kHz \tag{4.1.1}$$

$$f_{s,7} = \frac{fs_1}{7} = \frac{156.25kHz}{7} = 22.321kHz \tag{4.1.2}$$

The DAC, selected for this project, is the LTC2640 from Linear Technology. It comes with an SPI interface and a sampling rate up to $125kHz$. Depending on the model it supports a resolution of 8, 10 or $12bit$. The DAC has an internal reference voltage of $2.5V$ or $4.096V$. $12bit$ resolution with $4.096V$ reference voltage would be the ideal configuration of this DAC. Because of component shortage only the DAC with $2.5V$ reference voltage was available. This model produces a smaller output voltage which can be compensated by increasing the gain of the amplifier circuit described in section 4.3.[2]

4.2 Ultrasonic Transducers

The ultrasonic transducer chosen for this project is the CUSA-T80-15-2400-TH. It has an operating frequency of $40kHz$ and tolerates a supply voltage up to $80V$.[19]

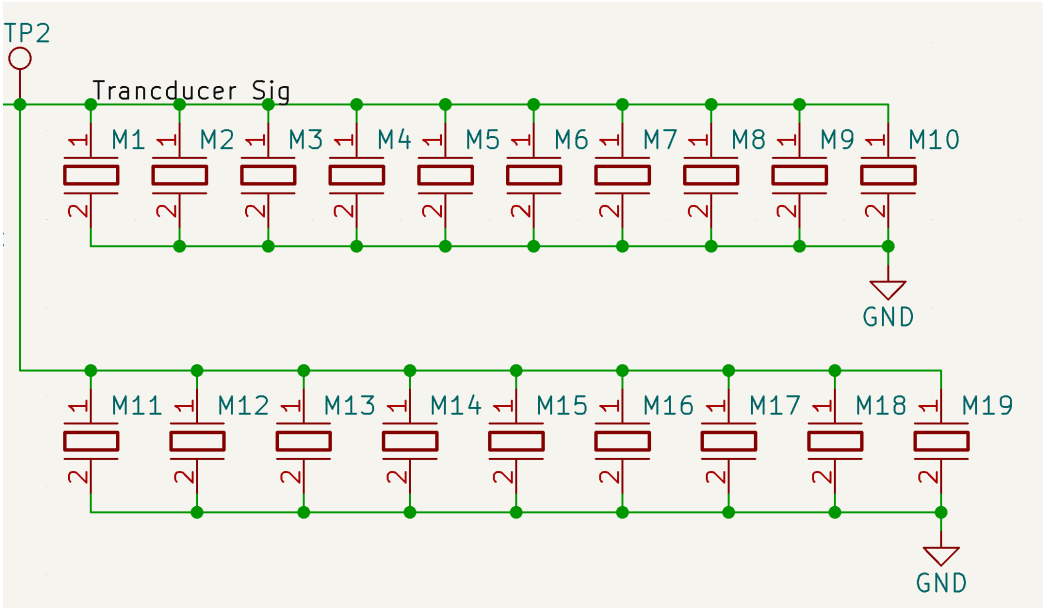


Figure 4.2.1: Transducer circuit design

One module uses 19 transducers which are connected in parallel as shown in figure 4.2.1. In order to create the aspired beamforming effect the transducers are evenly distributed on the pcb (figure 4.4.2b).

4.3 Amplifier

Ultrasonic transducers have to be driven with a much higher voltage and current than a DAC can provide. Therefore an amplifier circuit was designed which employs the DAC voltage as input and generates a suitable output for the array of ultrasonic transducers.

4.3.1 Circuit

The amplifier (figure 4.3.1) consists of two sections. First a non-inverting amplifier circuit to amplify the input voltage. Further, a Class B power amplifier is used so the circuit is able to drive the high capacitive load of the transducers. The whole circuit is designed for single supply because the power supply only provides a positive voltage.

Voltage input

C8 is used to filter the DC part of the input signal. **R8** and **R9** then generate a virtual ground (operating point) at $12V$. For AC signals a DC power supply can be seen as short circuit. This

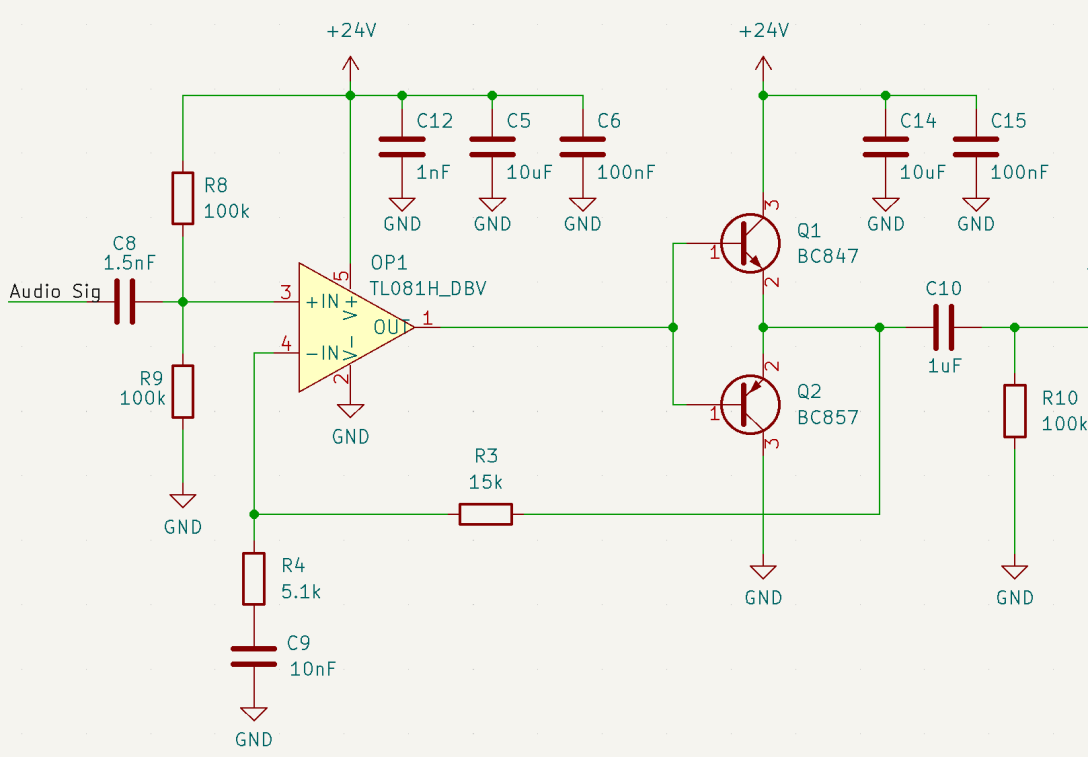


Figure 4.3.1: Amplifier circuit design

will result in **R8** and **R9** being parallel (figure 4.3.2). Combined with **C8** they build a simple high-pass which can be calculated for a specific cut-off frequency.

$$R_8||R_9 = \frac{1}{2}R_{8/9} \tag{4.3.1}$$

$$\frac{U_a}{U_{in}} = \frac{sC_8 \cdot \frac{1}{2}R_{R8/9}}{1 + sC_8 \cdot \frac{1}{2}R_{8/9}} \tag{4.3.2}$$

$$f_g = \frac{1}{2\pi C_8 \cdot \frac{1}{2}R_{R8/9}} \tag{4.3.3}$$

$$C_8 = \frac{1}{2\pi f_g \cdot \frac{1}{2}R_{R8/9}} \tag{4.3.4}$$

As cut-off frequency $4kHz$ was chosen and for **R8** and **R9** a value of $100k\Omega$ was selected. This results in a capacity of around $1.5nF$ for **C8**.

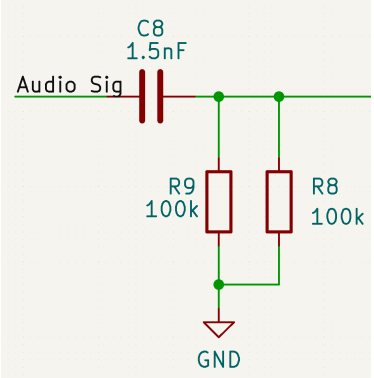


Figure 4.3.2: Input hightpass circuit

Feedback

Due to the new operating point of the input only AC signals should be amplified by the OpAmp. This is achieved with the capacity **C9** in the feedback. It blocks signals below the cut-off frequency, creating a voltage follower. For signals above the cut-off frequency the capacity **C9** acts as a short circuit. The transfer function of the feedback can be used to calculate **C9**. A capacitance of $10nF$ was selected. This results in a cut-off frequency of $3.2kHz$.

$$A = \frac{1 + sC_9(R_3 + R_4)}{1 + sC_9R_4} \tag{4.3.5}$$

$$f_g = \frac{1}{2\pi C_9 R_4} \tag{4.3.6}$$

The amplification can be calculated like a non-inverting amplifier. In order to take advantage of the $24V$ voltage range with the $4V$ DAC output[1] the amplification should be 4. With $R_3 = 15k\Omega$ the result for **R4** is $5.1k\Omega$.

$$V = 1 + \frac{R_3}{R_4} \tag{4.3.7}$$

As described above a power amplifier is needed to drive the high current of the ultrasonic transducers. A class B amplifier was chosen because of its simple but powerefficient design.[20] Usually this design causes crossover distortion but this is solved by putting the feedback behind the power amplifier. Note, this circuit design is only possible if the OpAmp is powerfull enough to drive the base current of the transistors. A better solution for higher currents would be to use the more complex class AB power amplifier.

[1] At this point it was not clear, that this version of the DAC would not be available

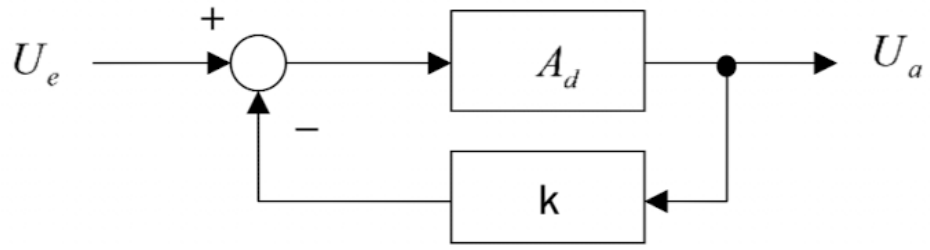


Figure 4.3.3: Block circuit of an amplification circuit[1]

Voltage output

At the output of the circuit a filter is used to filter the $12V$ DC voltage caused by the voltage divider at the input.

$$A = \frac{sC_{10} \cdot Z_{10}}{1 + sC_{10} \cdot Z_{10}} \tag{4.3.8}$$

$$f_g = \frac{1}{2\pi C_{10} \cdot Z_{10}} Z_{10} = R_{10} || Z_M \tag{4.3.9}$$

Because the behaviour of this high-pass depends on the ESR and capacity of all transducers, the size of **C10** was determined empirically. The best results where achieved with $10uF$.

4.3.2 Stability

As mentioned before the feedback of the amplification circuit includes a capacitance. To improve stability OpAmps are usually phase compensated to represent a first order low-pass. Introducing a capacitance into the feedback could disturb the stability of the circuit.[1]

A general amplification circuit can be represented as shown in figure 4.3.3. The output signal can therefore be represented as shown in equation 4.3.11.

$$U_a = \underline{\mathrm{A}}_d (U_e - \underline{\mathrm{k}} U_a) \tag{4.3.10}$$

$$U_a = U_e \frac{\underline{\mathrm{A}}_d}{1 + \underline{\mathrm{k}}\underline{\mathrm{A}}_d} \tag{4.3.11}$$

This results in an unstable circuit if $kA_d = -1$.

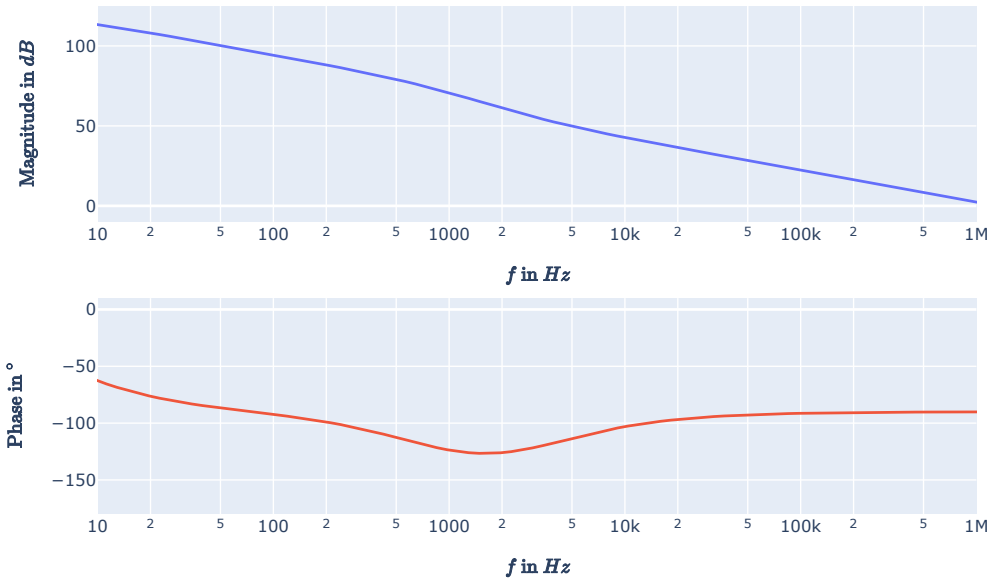


Figure 4.3.4: Loop gain

$$|\underline{\mathrm{kA}}_d| = 1 \tag{4.3.12}$$

$$\arg(\underline{\mathrm{kA}}_d) = -180° \tag{4.3.13}$$

In order to verify the stability, $\underline{\mathrm{A}}_d$ and $\underline{\mathrm{k}}$ of the amplifier circuit where calculated.

$$\underline{\mathrm{A}}_d = \frac{A_0}{1 + \frac{s}{w_g}} \tag{4.3.14}$$

$$w_g = 2\pi \cdot \frac{GBW}{A_0} \tag{4.3.15}$$

The open loop gain A_0 as well as the GBW were taken from the data sheet.[21]

$$\underline{\mathrm{k}} = \frac{1 + sC_9R_4}{1 + sC_9(R_4 + R_3)} \tag{4.3.16}$$

Figure 4.3.4 shows the bode plot of $\underline{\mathrm{kA}}_d$. It can be observed that the amplifier still has a phase reserve of 55°, which is enough to keep it stable.

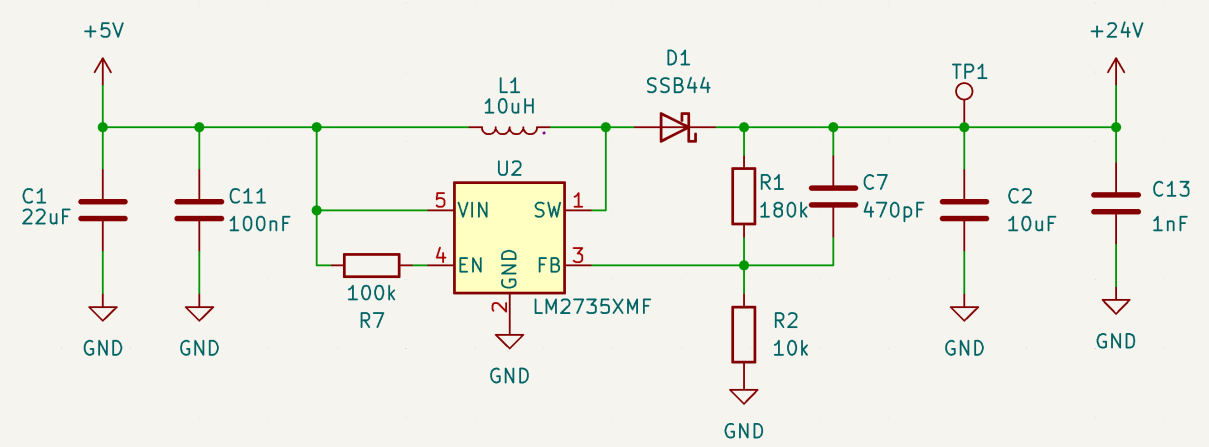


Figure 4.4.1: Powersupply circuit design

4.4 Power Supply

Because the speaker will be used with a Raspberry Pi or microcontroller, it should be compatible with an input voltage of $5V$. In order to provide the $24V$ needed by the amplifier circuit a boost converter is used.

In order to power the OpAmp and additional elements, the power supply should at least provide $200mA$. Apart from that, a higher switching frequency helps to reduce interferences at the speaker output.

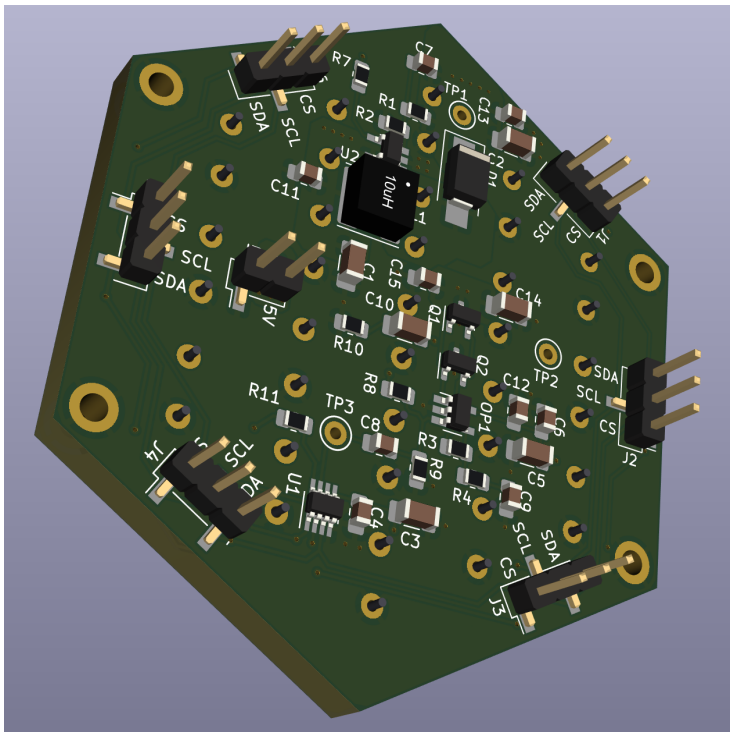
Considering those requirements the LM2735 boost converter was chosen. Because the power usage of the ultrasonic transducers was not clear in the beginning of this project the power supply provides up to $3A$ output current. The output voltage can be adjusted between $3V$ and $24V$.[22]

4.4.1 Circuit

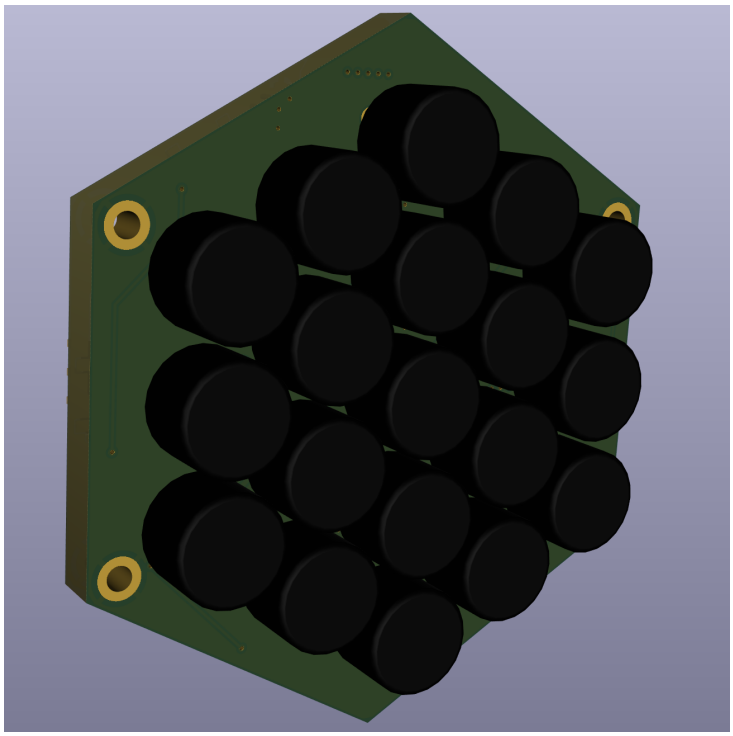
Figure 4.4.1 shows the circuit of the power supply. The feedback was designed for using the equation given in the datasheet. Note that **R1** and **R2** are switched compared to the circuit in the datasheet.

$$R_1 = \left(\frac{V_{out}}{V_{ref}} - 1\right) \cdot R_2 \qquad \text{with } V_{ref} = 1.255V,\ V_{out} = 24V \tag{4.4.1}$$

The dimensions of **L1** as well as the capacity **C7** and the diode **D1** were taken from the examples in the datasheet.



(a) Back



(b) Front

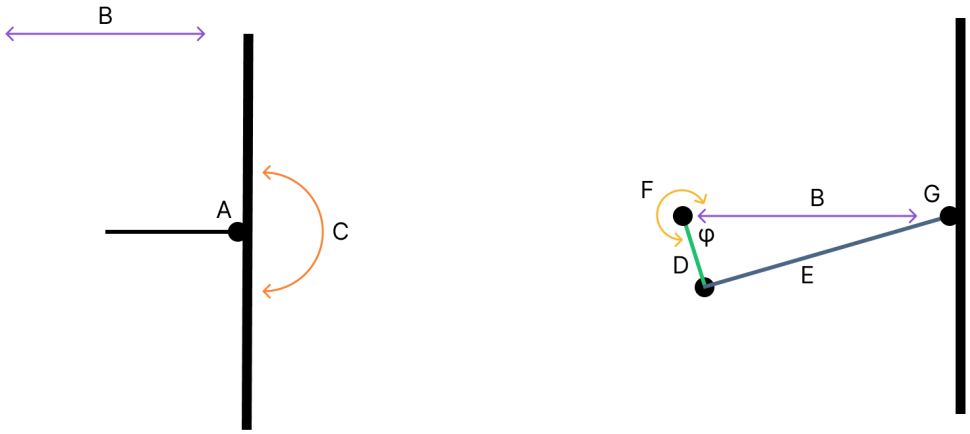
Figure 4.4.2: 3D model of the PCB layout

5 Speaker Construction

As described in section 2 the direction of the speaker is altered using a mechanical construction, which is explained in the following section.

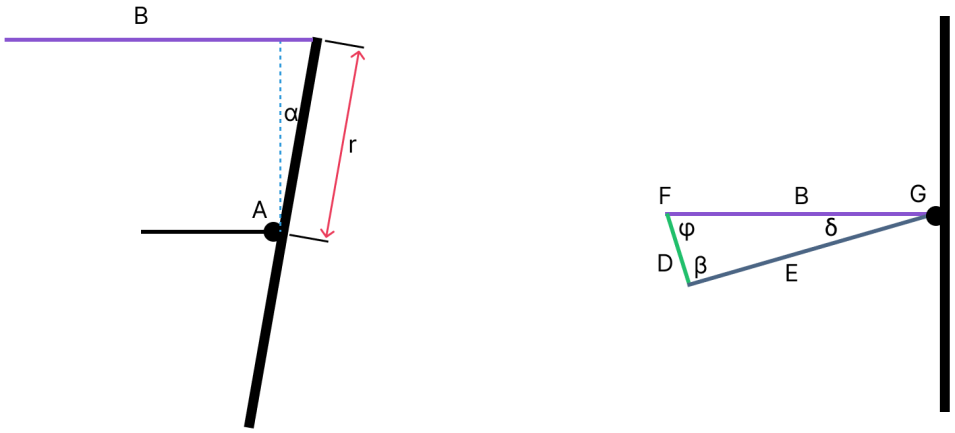
5.1 Tilting Mechanism

The tilting mechanism allows the PCBs to tilt around two axis. For developing the concept the PCBs are displayed as a plane. Figure 5.1.1 and 5.1.2 show the front and top view of the tilting mechanism for one axis.



(a) Front view **(b)** Top view

Figure 5.1.1: Mechanical sketch of the tilting mechanism



(a) Front view **(b)** Top view

Figure 5.1.2: Mathematical sketch of the tilting mechanism

The PCBs are held in place by a joint in the center of the plane (**A**). When the top of the PCBs is moved forwards or backwards (**B**), the plane rotates. The forward and backward movement can be created by connecting to shafts (**D** and **E**) with a joint. The base of **D** (**F**) is fixed while the end of **E** is connected to the top of the plane. This creates the triangle **D** - **E** - **B**. When the base of **D** is now rotated the shape of the triangle is changed, increasing or decreasing the length of **B**. The rotation around two axis can be achieved by adding a second **D** - **E** - **B** triangle to the left or right side of the plane.

Using this setup the tilting angle α is controlled using the angle φ. This can be described mathematically as follows. First φ is defined with the length of **B** using the law of cosines.

$$E^2 = D^2 + B^2 - 2DE \cdot \cos\varphi \tag{5.1.1}$$

$$\varphi = \arccos\left(\frac{D^2 + B^2 - E^2}{2DE}\right) \tag{5.1.2}$$

Next α is used to derive **B**. The result is inserted into equation 5.1.2.

$$\sin\alpha = \frac{B - B_0}{r} \tag{5.1.3}$$

$$B = \sin(\alpha) \cdot r + B_0 \tag{5.1.4}$$

$$\text{with } B_0 = B(\alpha = 0°, \varphi = 90°) = \sqrt{D^2 + E^2} \tag{5.1.5}$$

$$B = \sin(\alpha) \cdot r + \sqrt{D^2 + E^2} \tag{5.1.6}$$

$\mathbf{B_0}$ is the length of **B** when $\alpha = 0°$. To achieve symmetric movement α is defined to be $0°$ when $\varphi = 90°$. This results in the following equation for φ:

$$\varphi = \arccos\left(\frac{D^2 + (\sin(\alpha) \cdot r + \sqrt{D^2 + E^2})^2 - E^2}{2DE}\right) \tag{5.1.7}$$

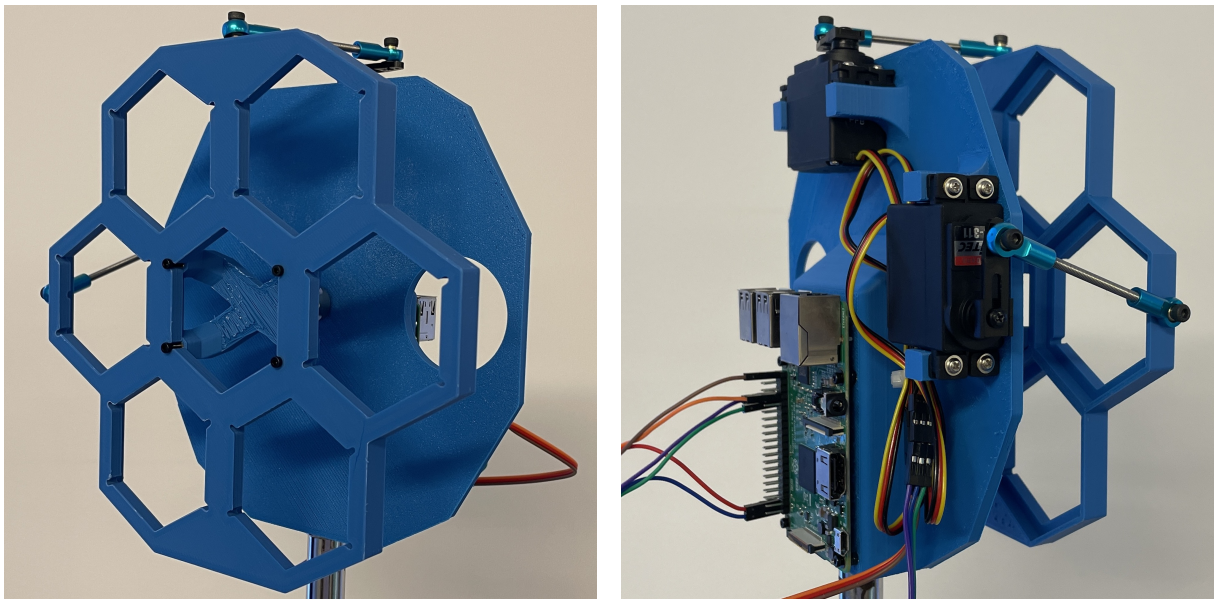
It should be noted that this solution is only an approximation. As shown in figure 5.1.2 even though the plane rotates, **B** remains in a straight line. In reality the plane would rotate on a circular path which would change the angle of **B** (in the front view) and with it the actual length of **B** depending on the angle α. However during testing on the real speaker the error of the calculation was to small to be noticeable.

5.2 Integration

As described in section 2 the speaker consist of seven modules. Therefore a frame is needed to mount them all into place. In addition a tripod mount should be added. In order to integrate the tilting mechanism the construction is divided into two parts: The frame, which mounts the PCB and the base, holding the servos and the control board (raspberry pi, microcontroller or similar). All parts are printed with the 3D printer. The finished speaker is shown in figure 5.2.1.

Tilting Mechanism

The tilting mechanism is realized using a cardan joint (figure 5.2.2a) as joint (**A**). The rotation of **F** is created with servos. The shafts **D** and **E** are built with a servo arm and threaded rod connected with ball joints (figure 5.2.2b). The whole construction is displayed in figure 5.2.1



(a) Front view **(b)** Back view

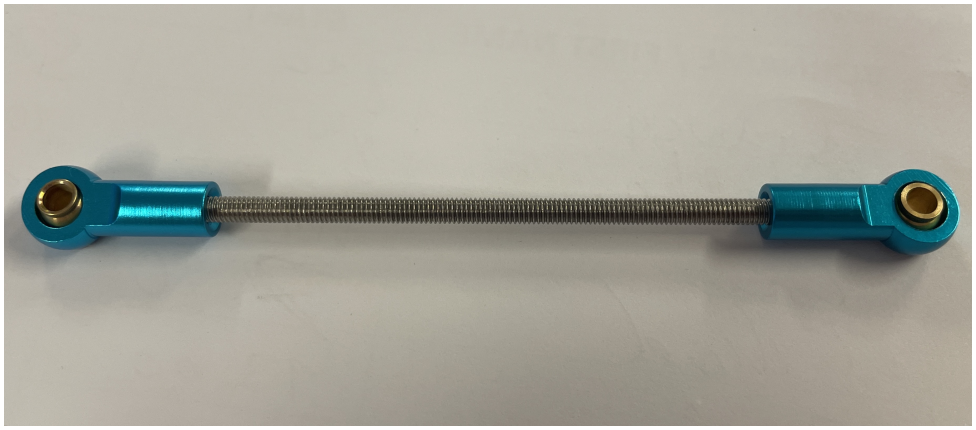
Figure 5.2.1: Complete speaker

PCB Mount

One speaker module is mounted using a hexagon frame (figure 5.2.3). The PCB can be bolted to the frame. To mount all modules, seven of those frames are combined to one model (figure 5.2.1). Threaded holes on the top and the left side provide an option to attach the ball joints for the tilting mechanism. The cardan joint is connected using the model shown in figure 5.2.3b. It is screwed to the back of the center PCB.

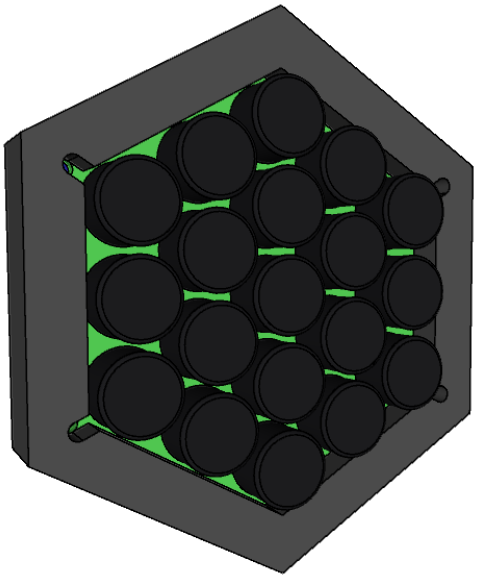
Base

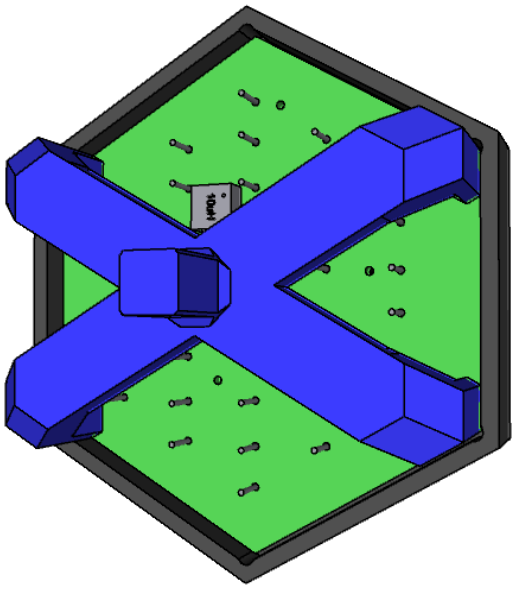
The base mimics the form of the PCB mount. A squared hole in the center allows the connection of the cardan joint. Servos are mounted to the back of the base at similar positions as the ball joints on the PCB mount. A thread on the bottom of the model will allow the speaker to be mounted on a tripod. The design of the base plate is shown in figure 5.2.4.



(a) Cardan Joint **(b)** Shaft

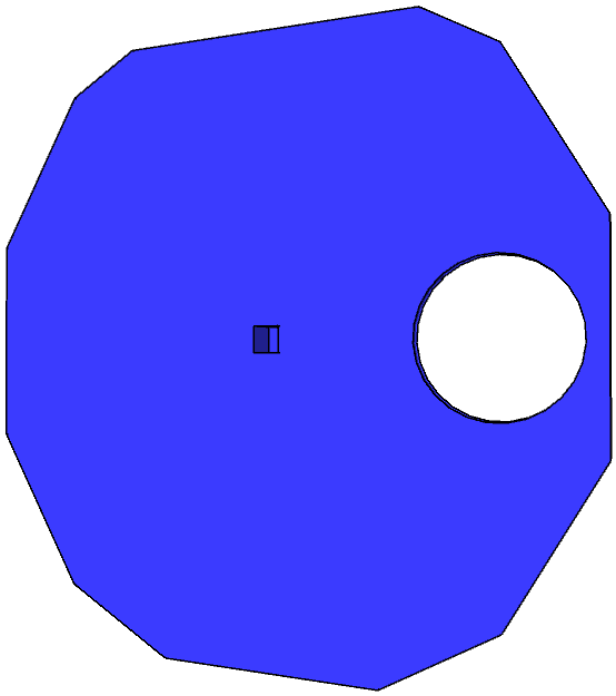
Figure 5.2.2: Components for the tilting mechanism



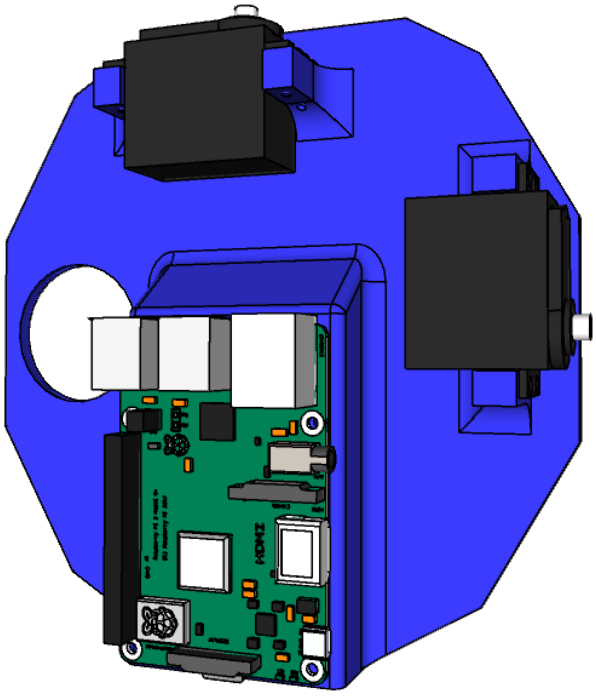


(a) Front view **(b)** Back view

Figure 5.2.3: PCB mount model with cardan joint connector



(a) Front view



(b) Back view

Figure 5.2.4: Base plate model with servo motors and Respberry Pi

6 Software

The software of the speaker has two main goals: Controlling the movement and playing an audio signal from an external source over SPI. Because both parts of the software don't need to interact with each other it was decided to split them up into two separate programs. The Speaker control is written in python but the Sound playback is written in C++ to increase performance.

A Raspberry Pi 3 Model B+ was chosen to run the software. With its Linux operating system it provides an easy way to parallize tasks. The Raspberry Pi also has a SPI interface and enough PWM pins to control the Servo Motors.[23]

6.1 Sound Playback

The sound playback program starts by reading samples from a connected audio device. Those samples are then modulated and sent over SPI with the right sampling rate. In order to generate a fluent audio playback the tasks where splitted into three threads. The whole sequence of the program is shown in figure 6.1.1.

Reading audio samples and modulating them is done in the class AudioProcessor. The SPI transmission is handled in the class Speaker. Both classes are built with the singleton pattern.

The communication between threads is handlend with a queue. The implementation shown in listing 6.1.1 makes the std :: queue threadsafe by using a lock to access the underlying queue and also blocks the thread if no data is available. This is usefull, among other things, because there is no need to send data over SPI when no new sample is generated.

A hardware abstraction layer is used to separate logic and hardware interfaces.

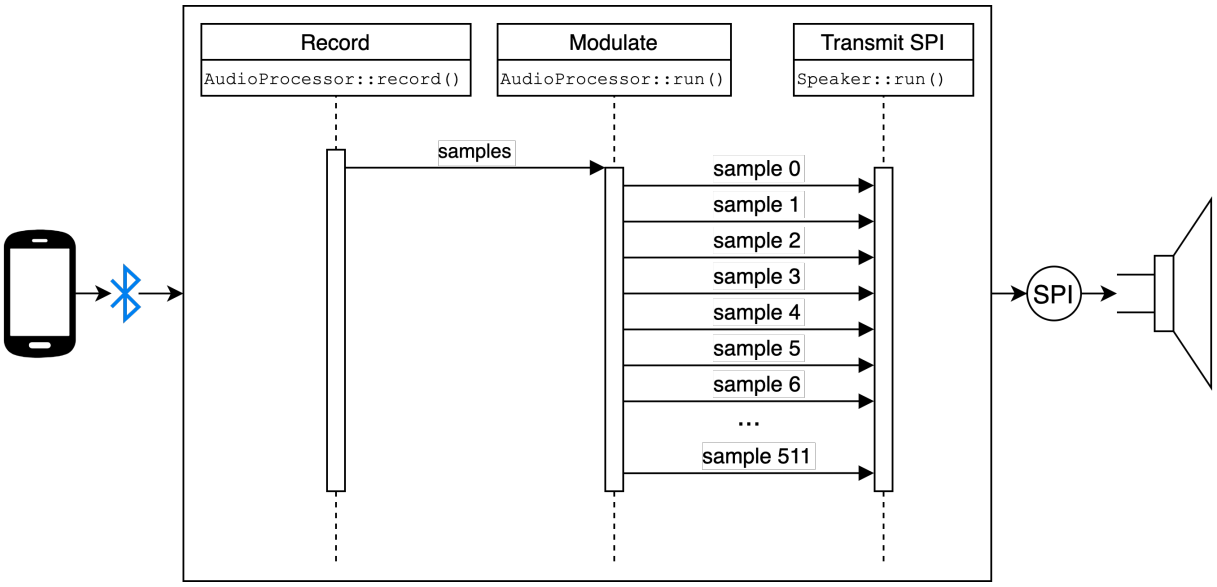


Figure 6.1.1: Sound playback sequence diagram

Listing 6.1.1: Threadsafe and blocking queue

```
template <class T> class BlockingQueue : public queue<T> {
  public:
  void push(T item) {
    {
      unique_lock<std::mutex> lck(lock);
      queue<T>::push(item);
    }
    not_empty.notify_one();
  }

  T pop() {
    unique_lock<std::mutex> lck(lock);
    not_empty.wait(lck, [this]() { return queue<T>::size() > 0; });

    T value = queue<T>::front();
    queue<T>::pop();
    return value;
  }

  bool notEmpty() { return !queue<T>::empty(); }

  private:
  std::mutex lock;
  condition_variable not_empty;
};
```

6.1.1 Audio Input

There are different methods to connect an external audio device to the Raspberry Pi. The Raspberry provides an AUX port where the microphone line could be used as input. An alternative would be to use an external USB audio card or an audio card HAT[2] with a dedicated AUX IN port. The third variant would be to connect the device over bluetooth.

Because of the current shortage off Raspberry Pi's and Raspberry Pi accessories a HAT is not an option. In order to use the Raspberry as a bluetooth speaker some complicated configuration and programming of the bluetooth interface has to be done, which would take a lot of time. The integrated audio interface of the Raspberry Pi could be used but it is known to have a mediocre sound quality. Therefore the best option would be an external USB audio card. However during research the open source tool BlueZ-Alsa[24] was found. This program makes it possible to configure the Raspberry Pi as bluetooth audio speaker with one simple shell-command. The connected device (Audio in/out depending on the configuration) can then be accessed using the Advanced Linux Sound Architecture (ALSA).

ALSA Interface

The interface to ALSA is the HAL class PCM. On construction it takes a device name, channel count and sampling rate. With those parameters the ALSA PCM device is configured using the **ALSA** C library (Listing 6.1.2).[25] The configuration is based on an example of the ALSA Audio API tutorial from Paul Davis.[26]

Listing 6.1.2: ALSA interface configuration

```
1  PCM::PCM(std::string device_name, char channel_cnt, uint32_t fs) {
2    int i;
3    int err;
4    snd_pcm_format_t format = SND_PCM_FORMAT_S16_LE;
5
6    if ((err = snd_pcm_open(&capture_handle, device_name.c_str(),
         SND_PCM_STREAM_CAPTURE, 0)) < 0) {
7      std::cerr << "cannot open audio device (" << snd_strerror(err) << ")\n";
8      exit(1);
9    }
10
11   if ((err = snd_pcm_hw_params_malloc(&hw_params)) < 0) {
```

[2] Hardware attached on top

```
12      std::cerr << "cannot allocate hardware parameter structure (" <<
            snd_strerror(err) << ")\n";
13      exit(1);
14    }
15
16    if ((err = snd_pcm_hw_params_any(capture_handle, hw_params)) < 0) {
17      std::cerr << "cannot initialize hardware parameter structure (" <<
            snd_strerror(err) << ")\n";
18      exit(1);
19    }
20
21    if ((err = snd_pcm_hw_params_set_access(capture_handle, hw_params,
          SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
22      std::cerr << "cannot set access type (" << snd_strerror(err) << ")\n";
23      exit(1);
24    }
25
26    if ((err = snd_pcm_hw_params_set_format(capture_handle, hw_params, format)) <
          0) {
27      std::cerr << "cannot set sample format (" << snd_strerror(err) << ")\n";
28      exit(1);
29    }
30
31    if ((err = snd_pcm_hw_params_set_rate(capture_handle, hw_params, (unsigned int
          )fs, 0)) < 0) {
32      std::cerr << "cannot set sample rate (" << snd_strerror(err) << ")\n";
33      exit(1);
34    }
35
36    if ((err = snd_pcm_hw_params_set_channels(capture_handle, hw_params,
          channel_cnt)) < 0) {
37      std::cerr << "cannot set channel count (" << snd_strerror(err) << ")\n";
38      exit(1);
39    }
40
41    if ((err = snd_pcm_hw_params(capture_handle, hw_params)) < 0) {
42      std::cerr << "cannot set parameters (" << snd_strerror(err) << ")\n";
43      exit(1);
44    }
45
46    snd_pcm_hw_params_free(hw_params);
47
48    if ((err = snd_pcm_prepare(capture_handle)) < 0) {
49      std::cerr << "cannot prepare audio interface for use (" << snd_strerror(err)
            << ")\n";
50      exit(1);
51    }
52
53    format_width = snd_pcm_format_width(format) / 8;
54    format_zero = 0;
55    format_max = pow(2, snd_pcm_format_width(format)) / 2;
56  }
```

After the configuration the method readFrames() can be used to read a given number of samples

(Listing 6.1.3). The method will block the code execution until all frames are received or an error occurs.

Listing 6.1.3: Method for reading frames from an ALSA device

```
1 void PCM::readFrames(sample_t* buffer, size_t frames) {
2   int err;
3
4   if ((err = snd_pcm_readi(capture_handle, buffer, frames)) != frames) {
5       std::cerr << "read from audio interface failed (" << snd_strerror(err) <<
            ")\n";
6       exit(1);
7     }
8 }
```

Record Audio

The PCM class is initialized in AudioProcessor::configure(). This instance is then used in the method record() shown in listing 6.1.4. The method creates a loop in which a set of samples is read into a buffer and then pushed into the AudioProcessor::samples queue. record() can be executed directly or by calling record_thread() which creates a new thread for the loop.

Listing 6.1.4: Record loop

```
1 void AudioProcessor::record() {
2   std::cout << "Run Record Loop\n";
3   size_t len = frame_size;
4
5   std::vector<sample_t> buffer(len);
6
7   while (true) {
8     pcm_dev->readFrames(buffer.data(), frame_size);
9     samples.push(buffer);
10  }
11 }
```

6.1.2 Modulation

The modulation process is realised in AudioProcessor::run() (Listing 6.1.5). To make switching between modulation techniques easier the methods for calculation the modulation itself are provided by the Modulator class.

Listing 6.1.5: Audio processing loop

```
1  void AudioProcessor::run() {
2    std::cout << "Run Audio Processor\n";
3
4    const auto& speaker = Speaker::instance();
5    Modulator mod(sampling_rate, frame_size, speaker->sampling_rate, 40000);
6
7    while (true) {
8      const auto& buffer = samples.pop();
9
10     for (const auto t : mod.time) {
11       const auto sig = mapValue(buffer.at(floor(t * sampling_rate)));
12       const auto sample = mod.AM(sig, t, 0.5f, 0.8f); // AM
13       // const auto sample = mod.FM(sig, t, 1.0f, 0.15f);    // FM
14       speaker->samples.push(speaker->mapSample(sample));
15     }
16   }
17 }
```

First the samples captured from the audio input are fetched from the AudioProcessor::samples queue. As the signal recorded with ALSA is a regular audio signal the sampling rate goes up to $48kHz$ depending on the configuration. The resulting signal however should have a sampling rate above $100kHz$. Therefore the signal needs to be resampled. This can be realised with different forms of interpolation. In this example the zero order hold variant is used.

As the Modulator class is instantiated a time vector with the target sampling rate and a size corresponding to the number of recorded samples is created. The modulation is calculated for every sample by iterating over the time vector.

Listing 6.1.6 and 6.1.7 show the C++ implementations of the AM and FM explained in section 3.2.

Listing 6.1.6: Amplitude modulation

```
1 float Modulator::AM(float signal, float t, float U0, float m) {
2   float c = sin(carrier * t);
3   return U0 * (1 + m * signal) * c;
4 }
```

Listing 6.1.7: Frequency modulation

```
float Modulator::FM(float signal, float t, float U0, float m) {
  float w0 = carrier;
  return U0 * sin(w0 * (1 + m * signal) * t);
}
```

6.1.3 SPI (Audio Output)

SPI Interface

As described before the sampling rate of the SPI output should be at least $100kHz$. This means every $10us$ one packet has to be submitted. The Linux OS for the Raspberry Pi provides a SPI implementation which can be accessed via C++ or Python. After putting the SPI write function into a loop without any pause or other calculations in between it became noticeable that this implementation is only able to send one package every $50\mu s$ to $80\mu s$. This problem can be solved with the **bcm2835** C library.[27] This library bypasses the linux drivers and directly accesses the bcm2835 SOC from the Raspberry Pi. It provides interfaces for GPIO pins, PWM and SPI as well as bcm2835_delayMicroseconds() function which is much more accurate than the builtin C++ functions.

The SPI configuration is put into the HAL class SPI (See listing 6.1.8)

Listing 6.1.8: SPI configuration

```
SPI::SPI() {
  if (!bcm2835_spi_begin()) {
    throw std::runtime_error("bcm2835_spi_begin failed. Are you running as root
        ??");
  }

  bcm2835_spi_setBitOrder(BCM2835_SPI_BIT_ORDER_MSBFIRST);
  bcm2835_spi_setDataMode(BCM2835_SPI_MODE0);
  bcm2835_spi_setClockDivider(BCM2835_SPI_CLOCK_DIVIDER_64); //
      BCM2835_SPI_CLOCK_DIVIDER_16);
  bcm2835_spi_chipSelect(BCM2835_SPI_CS0);
  bcm2835_spi_setChipSelectPolarity(BCM2835_SPI_CS0, LOW);
}
```

The SPI class provides the method write() which takes a byte array and writes it on the SPI bus (Listing 6.1.9). write() uses the method transfer() which passes through the function

bcm2835_spi_transfernb() from the **bcm2835** library. This function needs two buffers. One contains the data to be transmitted. The second one is used to save the received data.

Listing 6.1.9: Method for writing data onto the SPI bus

```
1 void SPI::write(char* data, uint32_t size) {
2   this->transfer(data, this->rx_buffer, size);
3 }
```

rx_buffer is a 4096 Byte array where the unused, received data from the write() method is dumped.

Transmit Audio

The audio transmission over SPI is performed in the class Speaker. The method Speaker::run() creates a loop in which the next audio sample is fetched from the Speaker::samples queue and transmitted over SPI. After the transmission the loop is paused to reach the right sampling rate. The sleep time is calculated dynamically depending on the time the sending of the data took. The coresponding code is shown in listing 6.1.10.

Listing 6.1.10: Audio transmission loop

```
1 void Speaker::run() {
2   std::cout << "Run Speaker Loop with a max delay of " << delay << "\n";
3
4   long long diff;
5   uint64_t delay = (uint64_t)(1000000000 / sampling_rate);
6   char data[WORD_SIZE] = { conf, 0x00, 0x00 };
7   auto start = std::chrono::high_resolution_clock::now();
8
9   while (true) {
10    auto sample = samples.pop();
11    data[1] = (char)(sample >> 4);
12    data[2] = (char)((sample & 0x000F) << 4);
13
14    spi.write(data, WORD_SIZE);
15
16    auto diff = std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono
          ::high_resolution_clock::now() - start)
17                    .count();
18    auto diff_abs = (uint64_t)diff <= delay ? diff : delay;
19    auto sleep = (delay - diff_abs) / 1000;
20
21    bcm2835_delayMicroseconds(sleep);
22    start = std::chrono::high_resolution_clock::now();
23  }
24 }
```

Linux Realtime

Because the signal is split into a fixed sampling rate, timing is really important for this task. As described before this is improved by using bcm2835_delayMicroseconds() over builtin C++ sleep functions. Nevertheless, because of the nature of Linux scheduling, it can not be quaranteed that the thread will pause exactly the given delay every time. Linux scheduling is fair. This means every process gets its time on the CPU. If a process was paused and needs to wake up again, it has to wait for the current process on the CPU to finish its time slice. For time relevant processes Linux provides a feature to bypass this behaviour. A process can be given a scheduling priority. When this process wants to wake up and a process with a lower or no priority uses the CPU it is paused immediately and the new proccess does not need to wait. This still does not quarantee a fixed sleep time because there could always be a task with higher priority but it reduces the risk of waiting.[28]

The scheduling priority can be changed using pthread_setschedparam() from the **PThread** library (Listing 6.1.11). It takes a priority between 1 and 99, where 99 is the highest priority and 1 is the lowest. A priority of 1 or 2 is usually sufficient because most processes do not have any scheduling priority.[29]

Listing 6.1.11: Example for creating a thread with realtime priority

```
1 std::thread* Speaker::run_thread() {
2   loop = new std::thread(&Speaker::run, this);
3   pthread_t id = (pthread_t)(loop->native_handle());
4
5   sched_param sched_params = { 2 };
6   pthread_setschedparam(id, SCHED_FIFO, &sched_params);
7
8   return loop;
9 }
```

All threads used in the program are started with a priority of 1 or 2 as shown in listing 6.1.11.

6.2 Speaker Movement

6.2.1 Speaker Control

The GPIO Pins of the Raspberry Pi can be controlled in Python using the **GPIOZero** library. It provides a class servo which takes the pin to which the servo is connected. The servo can then

be positoned by setting servo.value to a value between 0 and 1. Note that servo works with every pin but just pin **12** and **13** support hardware pins. For the other pins the PWM will be created by software. This variant usually creates some jitter which makes the servo tremble. To make sure the hardware PWM is used for pin **12** and **13** the pin factory should be changed to PiGPIOFactory (Listing 6.2.1).[30]

Listing 6.2.1: Changing the pin factory

```
1 from gpiozero.pins.pigpio import PiGPIOFactory
2 from gpiozero import Device
3 Device.pin_factory = PiGPIOFactory()
```

The configuration and control of the servo is handled in the HAL class ServoHAL (Listing 6.2.2). The class provides options to invert the servo or set an offset if the servo arm is not centered properly. set_position() takes an angle between $-90°$ and $90°$ and sets the servo position accordingly.

Listing 6.2.2: Servo configuration and control

```
1  class ServoHAL(HAL):
2
3  def __init__(self, pin: int, inverted: bool, offset: float):
4      self.pin = pin
5      self.offset = offset
6      self.inverted = -1 if inverted else 1
7
8      self.pwm = Servo(pin, initial_value=self._map_position(0), min_pulse_width
           =0.615 /
9                       1000, max_pulse_width=2.495 / 1000)
10
11 def _map_position(self, angle: float):
12     pos = self.inverted * (angle + self.offset) / 90
13
14     if pos > 1: pos = 1
15     if pos < -1: pos = -1
16
17     return pos
18
19 def set_position(self, pos: float):
20     self.pwm.value = self._map_position(pos)
21
22 def close(self):
23     self.pwm.value = None
```

In the class SpeakerControl the ServoHAL class is used to actually tilt the speaker. The class defines two servos for rotation around the x- and y-axis. Tilting can be initiated using the tilt_x() or tilt_y() method one of which is shown in listing 6.2.3. Both methods take the tilting angle in degree an map it to the corresponding servo postion using _map_position() which implements the calculations from section 5.1. The result is used to change the servo position.

Listing 6.2.3: Method for titling the speaker around the x-axis

```
def tilt_x(self, angle: float):
    servo_pos = self._map_position(angle)

    print(f"Tilt X - {servo_pos}")

    self._x_servo.set_position(servo_pos)
    self.x_angle = angle
```

6.2.2 Control Interface

The control interface consists of a website where a user can control the tilting of the speaker and an API over which the speaker is actually controlled. Both are created in Python with the **aiohttp** library.[31]

API

The API is generated in the class API. It registers the routes *tilt/x*, *tilt/y* and *tilt* as shown in listing 6.2.4. *tilt/x* and *tilt/y* are used to post a new titling angle to the server. *tilt* returns the current tilting position around the x- and y-axis in JSON format.

Listing 6.2.4: Control interface API

```
class API:
    def __init__(self, server: web.Application) -> None:
        self.server = server
        self._create_api()

    def _create_api(self):
        self.speaker_control = SpeakerControl(12, 13)
        self.server.router.add_post("/tilt/x", self._tilt_x_handler)
        self.server.router.add_post("/tilt/y", self._tilt_y_handler)
        self.server.router.add_get("/tilt", self._get_tilt_handler)
```

```
11
12  async def _tilt_x_handler(self, req: web.Request):
13      try:
14          content = json.loads(await req.text())
15          angle = content["value"]
16          print(angle)
17          self.speaker_control.tilt_x(angle)
18
19          res = dict(success=True, message="")
20      except Exception as err:
21          res = dict(success=True, message=getattr(
22              err, "message", repr(err)))
23
24      return web.Response(text=json.dumps(res), content_type='application/json')
25
26  async def _tilt_y_handler(self, req: web.Request):
27      try:
28          content = json.loads(await req.text())
29          angle = content["value"]
30          print(angle)
31          self.speaker_control.tilt_y(angle)
32
33          res = dict(success=True, message="")
34      except Exception as err:
35          res = dict(success=True, message=getattr(
36              err, "message", repr(err)))
37
38      return web.Response(text=json.dumps(res), content_type='application/json')
39
40  async def _get_tilt_handler(self, req: web.Request):
41      return web.Response(
42          text=json.dumps(dict(
43              x=self.speaker_control.x_angle,
44              y=self.speaker_control.y_angle
45          )), content_type='application/json')
```

Website

Even though the API can be used to control the speaker programmatically a website provides an easy interface for manual inputs.

In order to publish a website, a webbrowser needs to be configured. This is done in the class Webserver. As shown in listing 6.2.5 the constructor takes a reference to the **aiohttp** Application and a path to the directory containing the *.html*, *.js* and *.css* files. The webserver checks all files contained in this directory and registers new routes for each file so they can be accessed from a webbrowser.

Listing 6.2.5: Minimal python webserver

```python
class Webserver:
    def __init__(self, server: web.Application, web_directory: str) -> None:
        self.server = server
        self.web_path = web_directory
        self._create_webserver()

    def _create_webserver(self):
        files = os.listdir(self.web_path)
        routes = self._create_routes(files)

        for route in routes:
            self.server.router.add_get(f"/{route}", self._webadress_handler)

        self.web_routes = routes

    async def _webadress_handler(self, req: web.Request):
        path = req.path.lstrip("/")
        route_config = self.web_routes[path]
        with open(os.path.join(self.web_path, route_config[0])) as f:
            return web.Response(text=f.read(), content_type=route_config[1])

    def _create_routes(self, files: List[str]) -> Dict[str, Tuple[str, str]]:
        routes = {}
        for file in files:
            route = file

            split_file = os.path.splitext(file)
            content_type = f"text/{split_file[1].lstrip('.')}"

            if split_file[1] == ".html":
                if split_file[0] == "index":
                    route = ""
                else:
                    route = f"{split_file[0]}"
            if split_file[1] == ".js":
                content_type == f"text/javascript"

            routes[route] = (file, content_type)

        print(routes)

        return routes
```

The design of the website is shown in figure 6.2.1. When the site is loaded it uses the *tilt* route to get the current position of the speaker. Now the two text fields can be used to enter a new angle for rotation around the x- or y-axis. Clicking on the **move** button will post this input to *tilt/x* and *tilt/y* respectively.

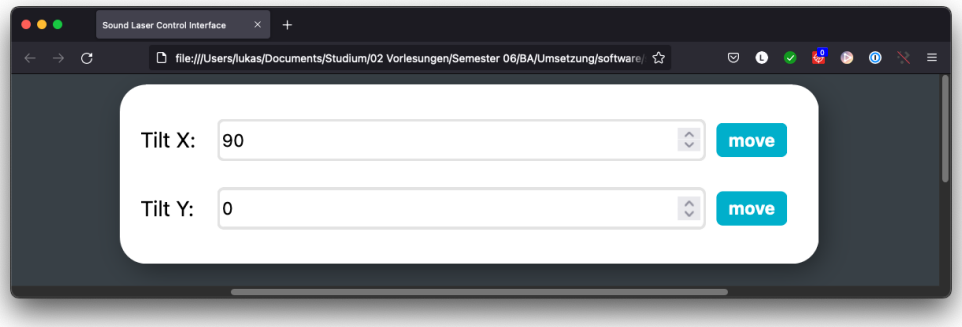


Figure 6.2.1: Graphical control interface of the speaker

7 Realworld Performance

The upcoming section will discuss the performance of the circuit as well as the software of the speaker. A special focus is placed onto the beamforming characteristics.

7.1 Circuit

7.1.1 Power Supply

Figure 7.1.1 shows the output of the boost converter. The measurements show that the output voltage reaches up to $23.7V$. Besides that voltage spikes of $239mV$ where measured.

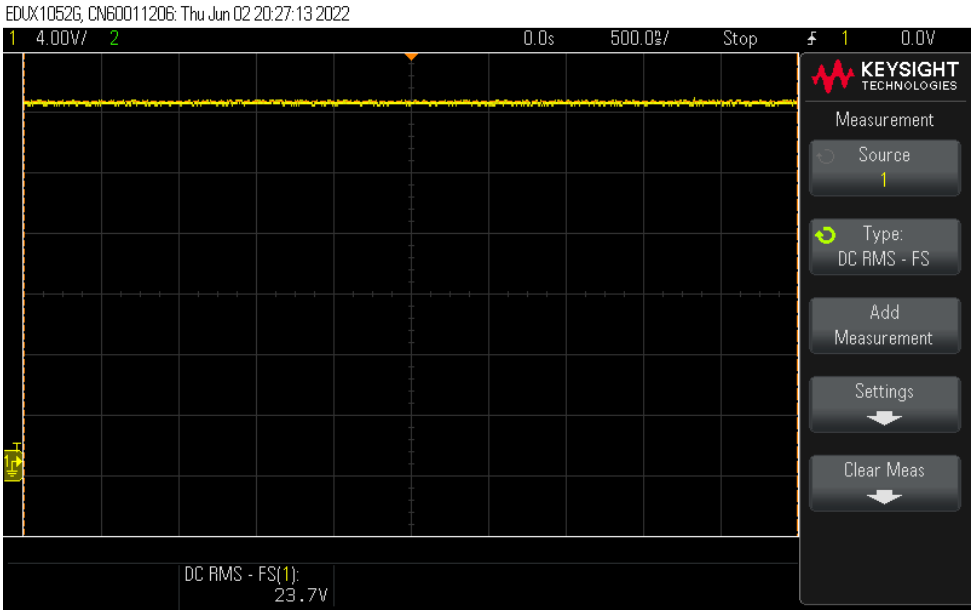
7.1.2 Amplifier Circuit

The frequency response of the amplifier circuit is shown in figure 7.1.2. It shows the amplification of $V = 4$ for frequencies above $5kHz$ as intended. Additionally, the trend shows an amplification of $0dB$ for low frequencies which is necessary because of the single supply design.

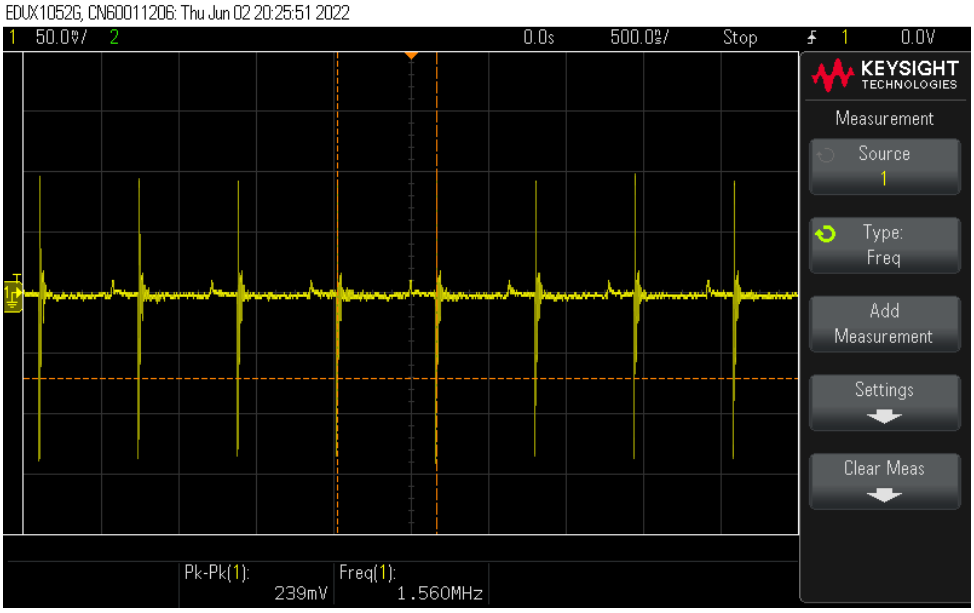
Figures 7.1.3 shows a $40kHz$ sine wave amplified by the circuit with and without a load. Both measurements display a clean sine wave.

7.1.3 DAC

During the writing of this thesis this component turned out to not work properly. As mentioned in the Datasheet the DAC takes the signal structure shown in Figure 7.1.4. When the 4 config bits are set to 0011 the output of the DAC is updated with the new value. The DAC also supports a power down mode by sending 0100. The power down can be recognized by observing the output of the reference voltage. In power down mode the pin is set to high impedance and slowly falls to $0V$.[2]



(a) Output voltage



(b) Ripple

Figure 7.1.1: Power supply output

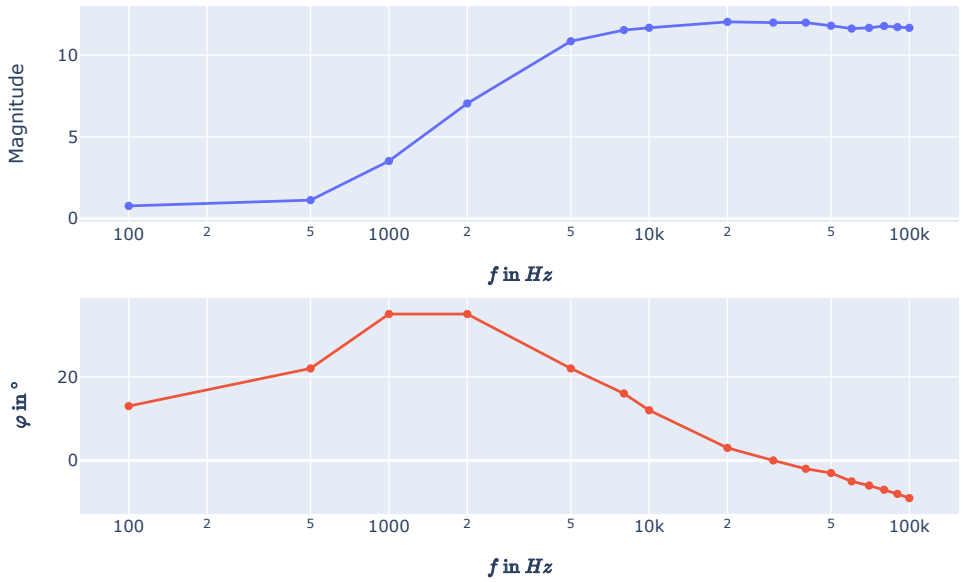
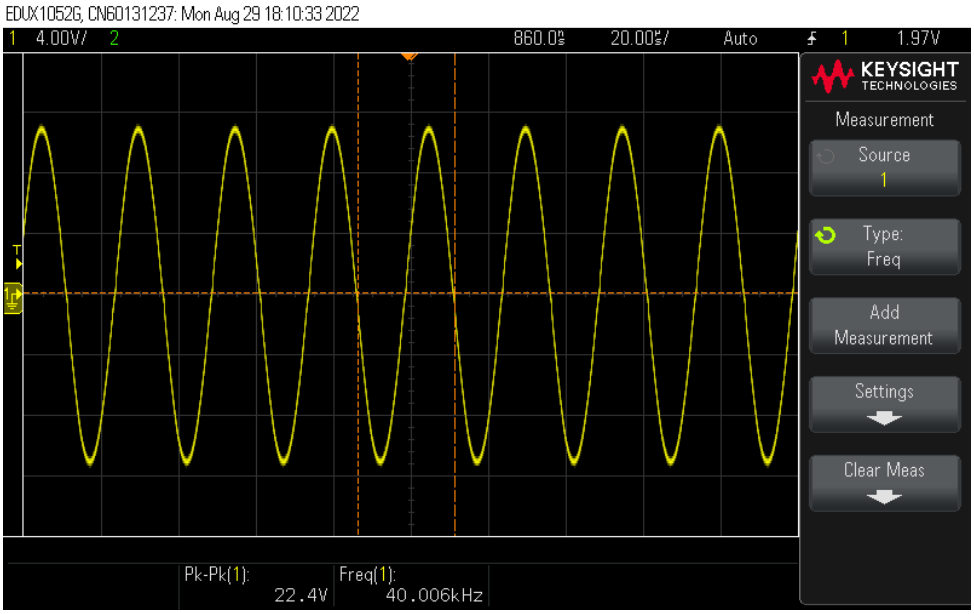


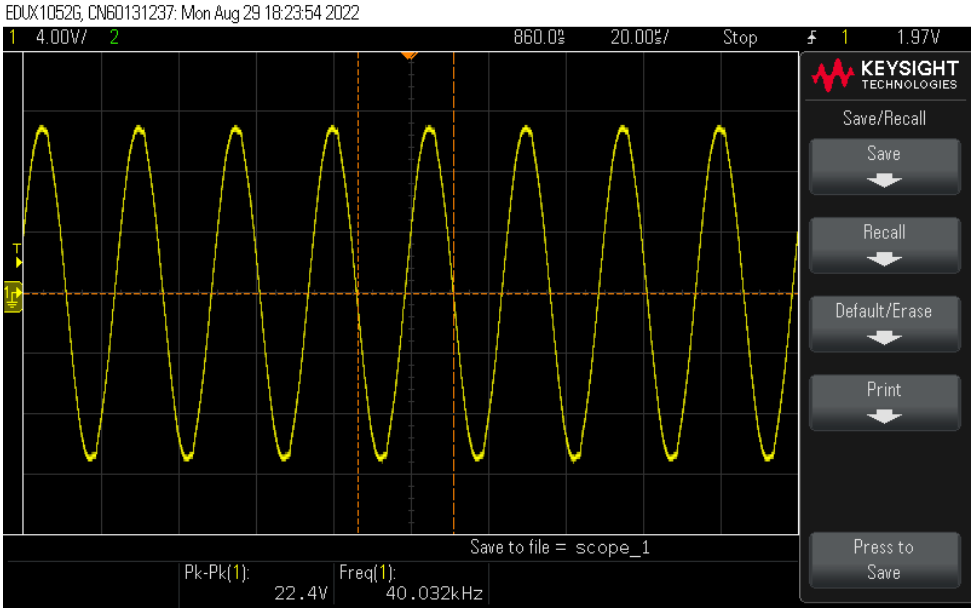
Figure 7.1.2: Frequency response of the amplifier circuit

While trying different values for the DAC it was observed, that the output voltage as well as the reference voltage became zero for some values. When sending the next value the DAC usually turned back on again. An analysis of the input and output signals during this process can be observed in Figure 7.1.5.

The picture shows the digital signal which has the config set to 0011 as it should be. Despite this the reference voltage (blue) becomes zero and the DAC powers down after reading the config bits. Neither the reason nor a solution for this problem could be found.



(a) Without load



(b) With load (19 ultrasonic transducers in parallel)

Figure 7.1.3: Amplification of a sine wave

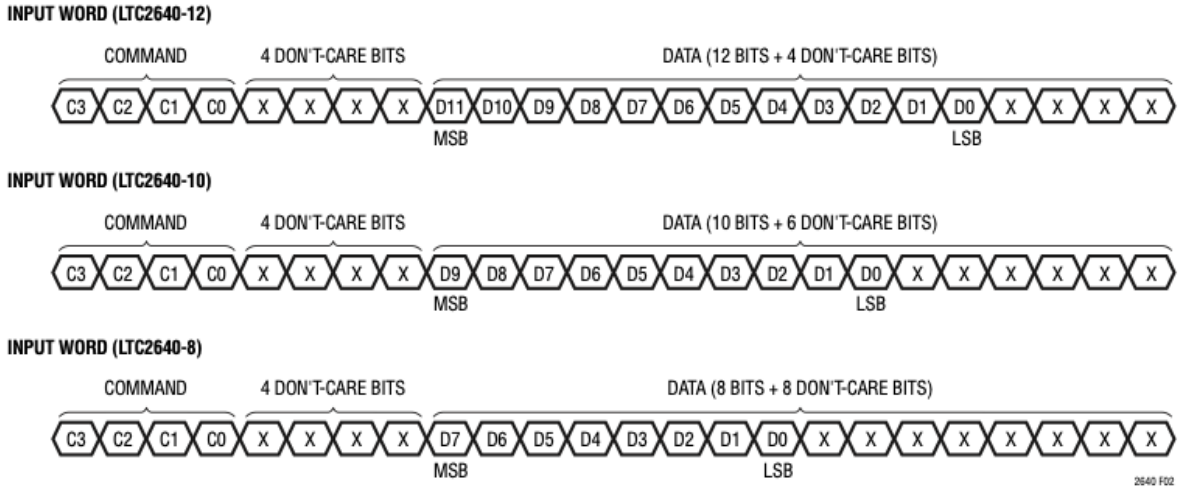


Figure 7.1.4: LTC2640 SPI signal structure[2]

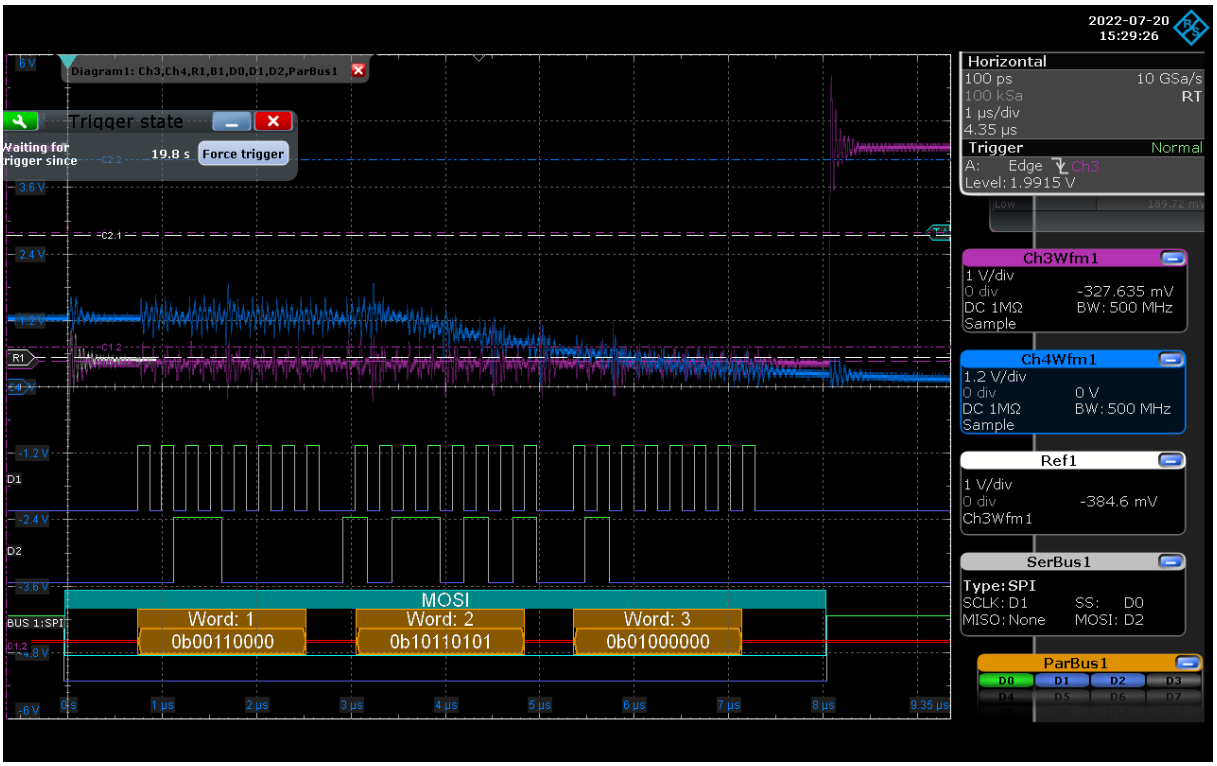


Figure 7.1.5: DAC Power down process

7.2 Software

7.2.1 Measurement Setup

The validation of the software was done using the Tabel class. This class provides multiple vectors which can be used to save samples or other data. When the program is exited this data is written to stdout using the *CSV* format. This can then be analyzed directly in the terminal or printed to a file and imported into a python notebook.

Instead of sending the resulting signal over SPI a timestamp is generated and both (timestamp and signal) are added to the table with addSignal(). Additionally, the duration of one iteration (with sleep) is saved using addDiff(). Finally the position in the table is increased to the next row with nexSignal().

7.2.2 Discussion

Figure 7.2.1 shows the duration of one iteration over the transmission of a $1s$ signal. It can be observed that the loop does not reach a stable duration. The smaller variations can be explained by the scheduling of linux and should not be a problem. However the graph also shows larger deviations from the intended $10\mu s$ loop duration.

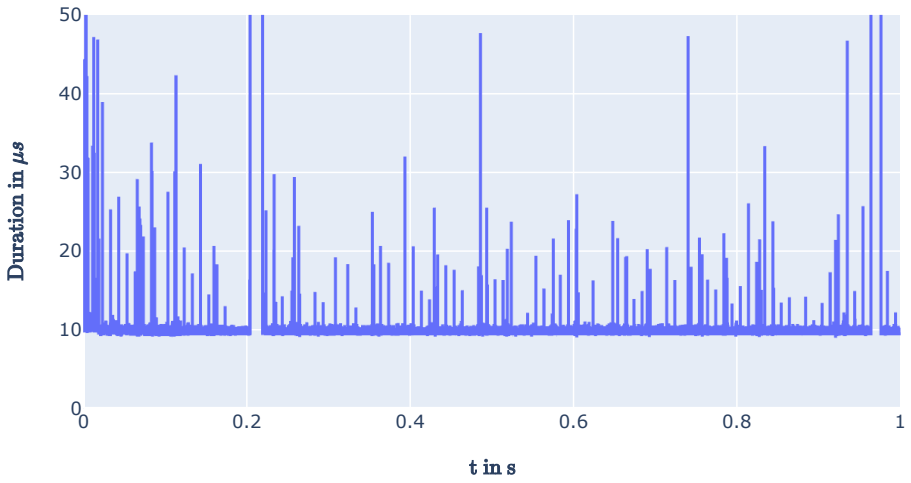
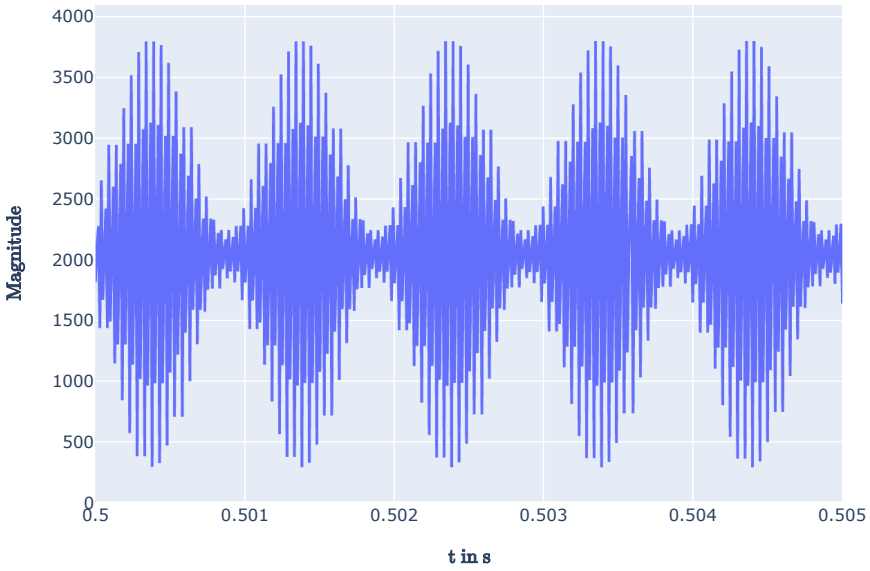


Figure 7.2.1: Duration of the loop

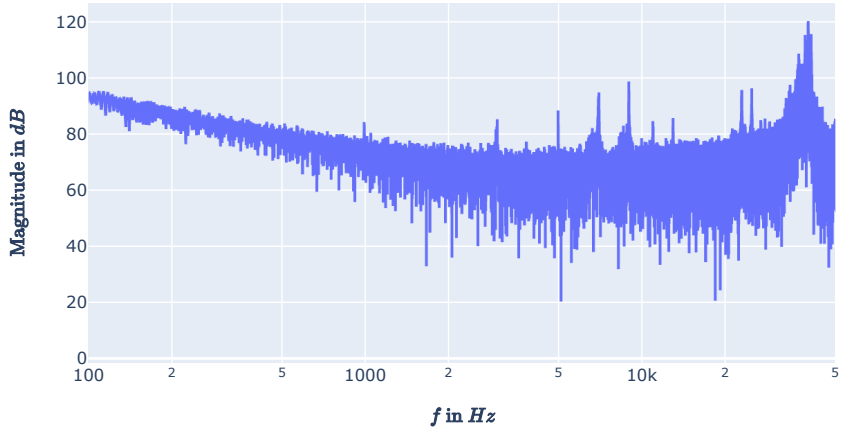
In Figure 7.2.2 the time and frequency spectrum of an amplitude modulated sine wave generated by the sound laser software is displayed. The time plot shows a clearly recognizable AM signal while the frequency spectrum is quite distorted. However the peak of the carrier wave as well as the upper and lower sideband are unmistakable.

7.2.3 Audiotest

In addition to the measurements above the software was tested using the speaker module. Because the original DAC did not work (See section 7.1.3) an 8 Bit DAC (MCP4901T) was used. Different sine waves as well as a song where played. In the resulting audio all sounds could be recognized. Unfortunately, the signal was very noisy. As analyzed before in figure 7.2.1 the signal also paused repeatedly for a short time.



(a) Time signal



(b) Frequency spectrum

Figure 7.2.2: AM modulated sine wave

7.3 Beamforming

7.3.1 Measurement Setup

The Goal of the measurements is to get an impression of the beamforming effect and compare it to the theoretical results from section 3.1. As discussed in the previous sections the DAC and the software have some problems which could interfere with the result. Therefore the testsignal is generated by a waveform generator. Figure 7.3.1 shows the measurement setup.

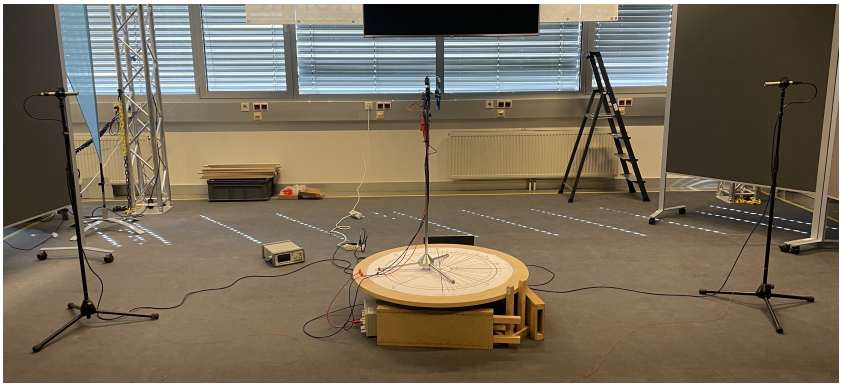


Figure 7.3.1: Measurement setup

The speaker is positioned on a rotating table to measure the audio for different angles. Two condenser microphones are positioned in front and behind the speaker. This halves the number of measurements needed for a full rotation. The audio is recorded with a **Zoom H5** audio interface.

AM and FM modulated sine signals with different frequencies between $0Hz$ and $20kHz$ are used to measure the behaviour of the speaker. The exact parameters are shown in table 7.3.1. Each frequency is recorded for $7s$. Additionally, $3s$ of silence are recorded which can be used to filter noise.

Table 7.3.1: Measurement parameters

	Modulation Depth	**Carrier frequency**	**Amplitude**
AM	0.8	$40kHz$	$2V$
FM	$5kHz$	$40kHz$	$2V$

7.3.2 Discussion

In order to analyze the measurements the signal as well as the noise are loaded and fourier transformed. In the next step the noise is substracted from the signal in order to reduce the noise in the actual measurement.

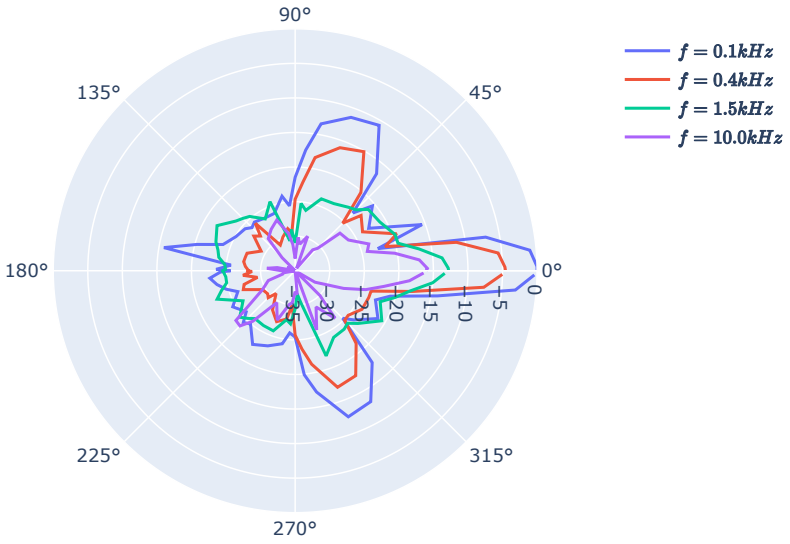


Figure 7.3.2: Measured beamforming characteristics of AM signals

To display the beamforming characteristics of different AM and FM signals the amplitude of the corresponding frequency is picked from the frequency spectrum. This is done for every angle. In the end the power of the signal in relation to the highest value is calculated in dB. Figures 7.3.2 and 7.3.2 show the resulting characteristics.

For comparison a graph using the same number of transducers (5) and the same distance between them ($10.5mm$) as the real speaker was created using the results from section 3.1. It is displayed in figure 7.3.4.

Both AM and FM signals show clear beamforming behaviour. Having said this the signal amplitude outside of the beam seems to be much higher for FM signals. Additionally, two side lobes at around $\pm 65°$ can be observed. They arise because the distance between the transducers is significantly higher than $\frac{\lambda}{2}$ and can also be seen in figure 7.3.4. While analyzing the results for all measured frequencies it was noticed that the power of the signal as well as the beamforming effect was smallest for signals between $1kHz$ and $2kHz$.

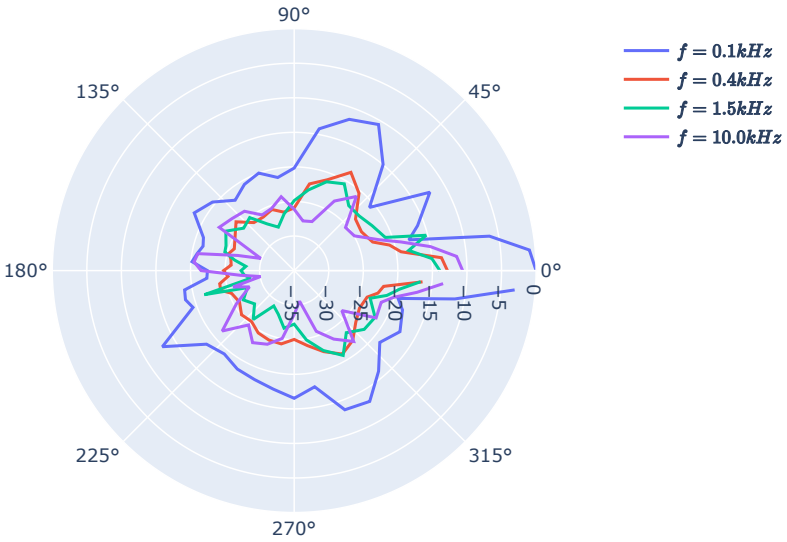


Figure 7.3.3: Measured beamforming characteristics of FM signals

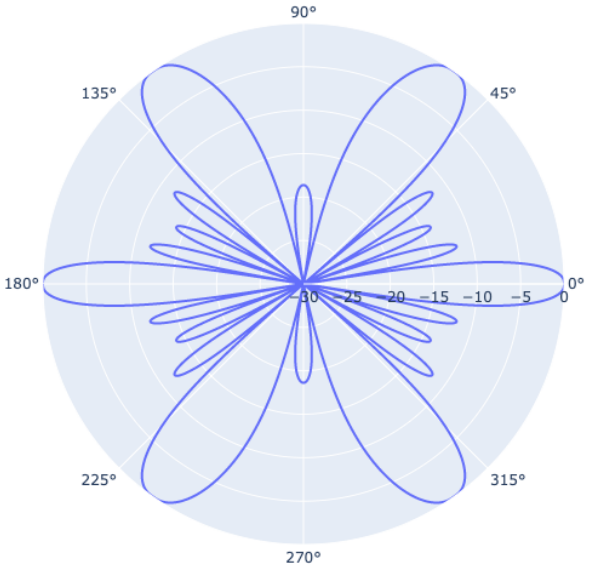


Figure 7.3.4: Calculated beamforming characteristics

8 Conclusion

Summary

In this project it was achieved to created a parametric speaker array which emits audible sound with clear beamforming behaviour. A circuit to operate this speaker was designed and a software was developed which receives audio over bluetooth modulates the signal and transmits it to the speaker. Additionally, a construction to tilt the speaker was built.

Follow Up

Even though the circuit did not work in the end, this can be easily fixed by replacing the DAC with an other model. The performance problems of the software can be fixed as well by porting the code onto a realtime capable device. A compromise would be to record and modulate the signal on the Raspberry Pi and transmit it to a microcontroller in a large buffer. Due to that, the microcontroller transmits the signal with the given sampling rate.

Following this project, an in-depth analysis of different modulation techniques, for example pulse modulation or single sideband AM, could be conducted. Furthermore, it would be interesting to examine the behaviour of multiple speaker modules as this would increase the beamforming characteristics.

Ressources

The source code of the software as well as 3D Modells for the tilting mechanism and a KiCAD project for the PCB design can be accessed on Github: https://github.com/lufixSch/sound-laser.

Bibliography

[1] R. Okorn, *Analoge Signalverarbeitung.* Graz: Vorlesungsskript FH JOANNEUM, Feb. 2021.

[2] Linear Technology, "LTC2640 - Single 12-/10-/8-Bit SPI VOUT DACs with 10ppm/°C Reference," Jun. 2017.

[3] C. Netzberger, *Kommunikationstechnologie Kaptiel 3 - Übertragungskanal.* Graz: Vorlesungsskript FH JOANNEUM, 2021.

[4] R. A. Novelline, *Squire's fundamentals of radiology.* Cambridge, Mass. : Harvard University Press, 1997. [Online]. Available: http://archive.org/details/squiresfundament0000nove_f4e9 [Accessed: 2022-08-31].

[5] E. Konofagou, J. Thierman, and K. Hynynen, "A focused ultrasound method for simultaneous diagnostic and therapeutic applications—a simulation study," *Physics in Medicine and Biology*, vol. 46, no. 11, pp. 2967–2984, Oct. 2001. [Online]. Available: https://doi.org/10.1088/0031-9155/46/11/314 [Accessed: 2022-08-31].

[6] T. G. Muir and R. J. Wyber, "High-resolution seismic profiling with a low-frequency parametric array," *The Journal of the Acoustical Society of America*, vol. 76, no. S1, pp. S78–S78, Oct. 1984. [Online]. Available: https://asa.scitation.org/doi/10.1121/1.2022023 [Accessed: 2022-08-31].

[7] B. Van Veen and K. Buckley, "Beamforming: a versatile approach to spatial filtering," *IEEE ASSP Magazine*, vol. 5, no. 2, pp. 4–24, Apr. 1988.

[8] G. Elert, "Speed and Velocity," *The Physics Hypertextbook*, 2021, publisher: hypertextbook. [Online]. Available: https://physics.info/velocity/ [Accessed: 2022-09-08].

[9] D. C. Cassidy, G. Holton, F. J. Rutherford, and H. P. Physics, *Understanding Physics.* Springer Science & Business Media, Sep. 2002.

[10] Curtis Technology (UK) Ltd, "TIME DOMAIN SONAR BEAMFORMING," p. 12, 1998.

[11] B. T. Cox and P. C. Beard, "The frequency-dependent directivity of a planar fabry-perot polymer film ultrasound sensor," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 54, no. 2, pp. 394–404, Feb. 2007.

[12] S. Hill and S. Dixon, "Frequency dependent directivity of periodic permanent magnet electromagnetic acoustic transducers," *NDT & E International*, vol. 62, pp. 137–143, Mar. 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0963869513001631 [Accessed: 2022-08-31].

[13] C. Netzberger, *Kommunikationstechnologie Kapitel 5 - Analoge Modulationsverfahren.* Graz: Vorlesungsskript FH JOANNEUM, 2021.

[14] P. J. Westervelt, "Parametric Acoustic Array," *The Journal of the Acoustical Society of America*, vol. 35, no. 4, pp. 535–537, Apr. 1963. [Online]. Available: https://asa.scitation.org/doi/abs/10.1121/1.1918525 [Accessed: 2022-09-02].

[15] M. Yoneyama, J. Fujimoto, Y. Kawamo, and S. Sasabe, "The audio spotlight: An application of nonlinear interaction of sound waves to a new type of loudspeaker design," *The Journal of the Acoustical Society of America*, vol. 73, no. 5, pp. 1532–1536, May 1983. [Online]. Available: http://asa.scitation.org/doi/10.1121/1.389414 [Accessed: 2022-08-29].

[16] L.-l. Bai, "Analysis of the performance of underwater acoustic parametric array under different input signals," *2012 Symposium on Piezoelectricity, Acoustic Waves, and Device Applications (SPAWDA)*, pp. 89–92, Nov. 2012.

[17] M. Nakayama and T. Nishiura, "Synchronized amplitude-and-frequency modulation for a parametric loudspeaker," *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pp. 130–135, Dec. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8282014/ [Accessed: 2022-09-05].

[18] NXP, "I2C-bus specification and user manual," 2021.

[19] CUI Devices, "CUSA-T80-15-2400-TH - ULTRASONIC SENSOR," Feb. 2020.

[20] R. Okorn, *Halbleiterschaltung.* Graz: Vorlesungsskript FH JOANNEUM, Aug. 2020.

[21] Texas Instruments, "TL08xx FET-Input Operational Amplifiers," Dec. 2021.

[22] Texas Instruments, “LM2735-Q1 520-kHz and 1.6-MHz Space-Efficient Boost and SEPIC DC/DC Regulator,” 2018.

[23] G. van Loo, “BCM2836 Peripharals,” Aug. 2014.

[24] A. Bokowy, “BlueZ-Alsa,” Aug. 2022. [Online]. Available: https://github.com/Arkq/bluez-alsa [Accessed: 2022-08-21].

[25] “ALSA project - the C library reference.” [Online]. Available: https://www.alsa-project.org/alsa-doc/alsa-lib/index.html [Accessed: 2022-08-21].

[26] P. Davis, “A tutorial on using the ALSA Audio API,” 2002. [Online]. Available: http://equalarea.com/paul/alsa-audio.html [Accessed: 2022-08-21].

[27] M. McCauley, “bcm2835: C library for Broadcom BCM 2835 as used in Raspberry Pi.” [Online]. Available: http://www.airspayce.com/mikem/bcm2835/index.html [Accessed: 2022-08-21].

[28] J. Faschingbauer, *Realtime*. Graz: Vorlesungsskript FH JOANNEUM. [Online]. Available: https://www.faschingbauer.me/trainings/material/soup/linux/sysprog/scheduling/realtime.html [Accessed: 2022-08-31].

[29] “pthread_setschedparam(3) - Linux man page.” [Online]. Available: https://linux.die.net/man/3/pthread_setschedparam [Accessed: 2022-08-31].

[30] B. Nuttall, “GPIO Zero 1.6.2 Documentation,” Mar. 2021. [Online]. Available: https://gpiozero.readthedocs.io/en/stable/ [Accessed: 2022-08-31].

[31] “aiohttp 3.8.1 documentation.” [Online]. Available: https://docs.aiohttp.org/en/stable/ [Accessed: 2022-08-31].