

0.1 To SKO

Hello Søren

We have written some more stuff in the following:

- Equalizer from "Transformation to the digital domain" and onwards. - Also related to that is appendix C about the bilinear transform. Does it make sense and is it okay to put it in an appendix? - wifi design, the parts about 5.2.3 "Bit rate and package loss test".

Questions: - At low frequencies the output signal from the filters have a lot of noise. There is a correlation between how much the signal is downsampled and how much noise there is. Could this be a result of zero-padding or are we handling the filtering wrong? See 'Programming of filters' for code examples. - We are having trouble finding good sources on up- and downsampling as well as zero padding so if you have some that would be helpful.

Gr512.

Ps. Do you like figure 5.1?

Contents

0.1 To SKO	1
1 Introduction	1
1.1 Frequency Responses of audio systems	1
1.2 Acoustic properties of a room	1
1.3 Summation of Problem	3
2 Theory	4
2.1 Equalizer	4
2.2 Digital sound	5
2.3 Digital signal processing	7
2.4 The amazing world of sound cards	8
2.5 Platforms for digital signal processing	9
2.6 Wireless communication	10
3 Problem Statement	12
3.1 Project Scope	12
3.2 Problem statement	13
4 Requirements specification	14
4.1 Functional requirements	14
4.2 Division into subsystems	14
4.3 Requirements for audio interface	15
4.4 Requirements for equalizer	16
4.5 Requirements for controller	16
4.6 Requirements for wireless communication	16
5 Design	17
5.1 Design of audio interface	17
5.1.1 Recording of reference audio and playback of corrected audio	19
5.1.2 Recording of played signal	20
5.2 Wireless interface design	21
5.2.1 Connection structure and protocol suite	21
5.2.2 Connection using socket programming	22
5.2.3 Partialtest of WiFi connection	24
5.3 Design of Equalizer	25
5.3.1 Determining the center frequencies	25
5.3.2 Determination of filter transfer functions	26
5.3.3 Transformation to the digital domain	31
5.3.4 Down- and upsampling	36
5.4 Design of controller	45
5.5 Recap of the whole system	47
Glossary	48

Bibliography	49
A Wireless bit rate approximation	51
B Test of package loss rate	53
C The bilinear transform	55
D C-code used in the project	57
E Matlab scripts for equalizer calculations	58
F Test of equalizer	60
F.1 Test of amplitude response	60
F.2 Test of band attenuation	62
F.3 Test of phase response	62
F.4 Test of computation time	62
F.5 Conclusion	62

Introduction 1

1.1 Frequency Responses of audio systems

Hi-Fi audio equipment like amplifiers and loudspeakers, are very popular pieces of household electronics. People are seemingly willing to spend a lot of money on sound systems that is able to accurately reproduce sound. Amongst different types of sound-reproducing devices, the transducers¹ are generally the worst at accurately reproducing the actual sound, especially for loudspeakers[13]. Purely electronic devices, like amplifiers, are generally easier to control.

A key feature of high quality sound systems are their frequency response. The frequency response is a term describing the variations in amplitude and phase from the original signal as a function of frequency. It is generally desired to have a "flat" frequency response i.e. the amplitude- and phase response of the output signal behaves the same way, regardless of the frequency of the input signal.

This is hard to achieve in real loudspeakers because of resonances in the physical part of the loudspeaker, which will result in certain frequencies being played louder than others.

The Loudspeaker driver that converts the electric signal to sound waves only has a certain limited range of frequencies where it works optimally. Loudspeakers made with only one driver (Full-range drivers) usually have a hard time reproducing every frequency at a similar amplitude, especially when playing at high levels of power. A lot of loudspeaker systems are therefore made using multiple drivers, that each handles a certain band of frequencies.

There are different design and construction challenges for each type of driver depending on which part of the audio spectrum it should reproduce, but in general drivers for high frequencies are smaller than drivers for lower frequencies. Since drivers for reproducing the lowest part of the frequency spectrum (20 - 200 Hz) are usually quite big and use a lot of power. For this reason, loudspeaker systems with limited size and power usage (like laptop speakers) are usually physically incapable of reproducing these frequencies at the same amplitude as higher frequencies.

1.2 Acoustic properties of a room

Even with a perfectly flat frequency response in a sound system, the sound that is actually being perceived by the listener might still be different than the original audio because of the acoustics of the surrounding room.

Room acoustics is a whole subject onto itself, this section will not go into depth about every physical phenomenon that can influence how a room can affect the sound quality. This section will however try to briefly cover how the surrounding room can influence the perceived sound quality for a listener.

¹Devices that convert one type energy into another.

When a sound wave hits an object in its path, a combination of three things happen. Some of the wave will be reflected off of the surface of the object. The rest of the wave will travel through the object, where some of the waves energy will be absorbed by the material and transformed into heat. Whatever is left of the wave will be transmitted through the object and continue along its direction of propagation. An illustration of this can be seen on figure 1.1.

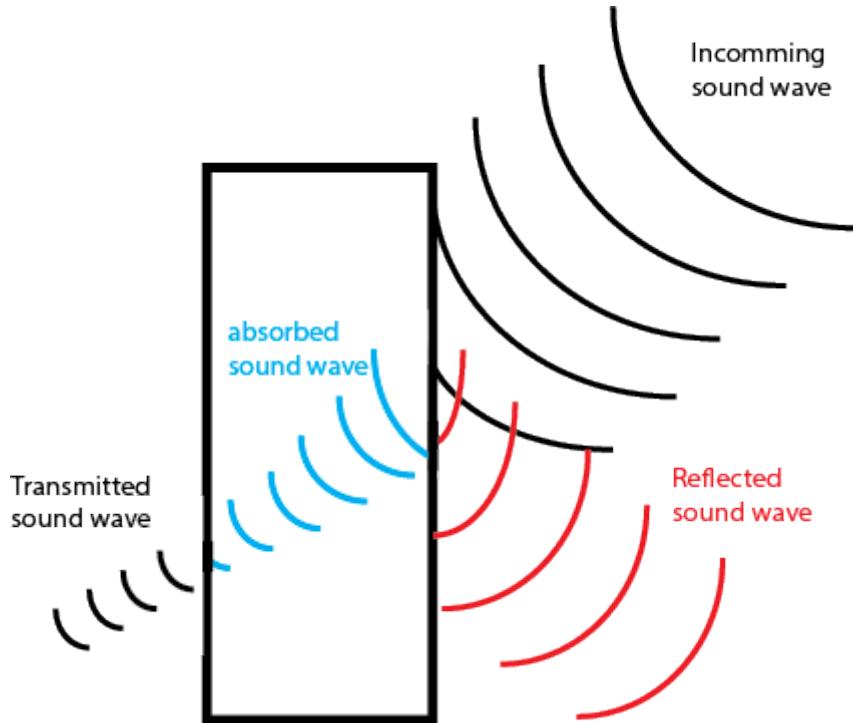


Figure 1.1: sound interacting with a large object. Illustration from [10] Wall

How much of the sound wave is reflected, absorbed or transmitted through, is dependent on both the material of the object, the angle of incidence, and the frequency of the sound wave. **rule of thumb for different building materials** If the object being struck by the sound wave is significantly larger in area than the incoming wave, then the reflected sound wave will be reflected at an angle equal to the angle of the incoming wave. For example, a room with large and bare concrete walls will reflect **specific** frequencies a lot. If the waves hits a surface that is comparable in size to its wavelength, the waves will diffuse and scatter somewhat evenly across the room.

Acoustic treatment of rooms is often done by fitting rooms with objects that will diffuse incoming sound waves, i.e. covering surfaces with sound absorbent materials. This is done to reduce reflected waves in a room.

One problem associated with reflections is the emergence of interference patterns between a sound wave in a direct path to the listener the sound wave reflected off a surface. Since the reflected wave has traveled a longer distance than the direct wave, the two waveforms will likely be out of phase with each other and interference will occur. This can cause the resulting wave to either have a larger or smaller amplitude than the original wave. An example of how this can occur can be seen in figure 1.2

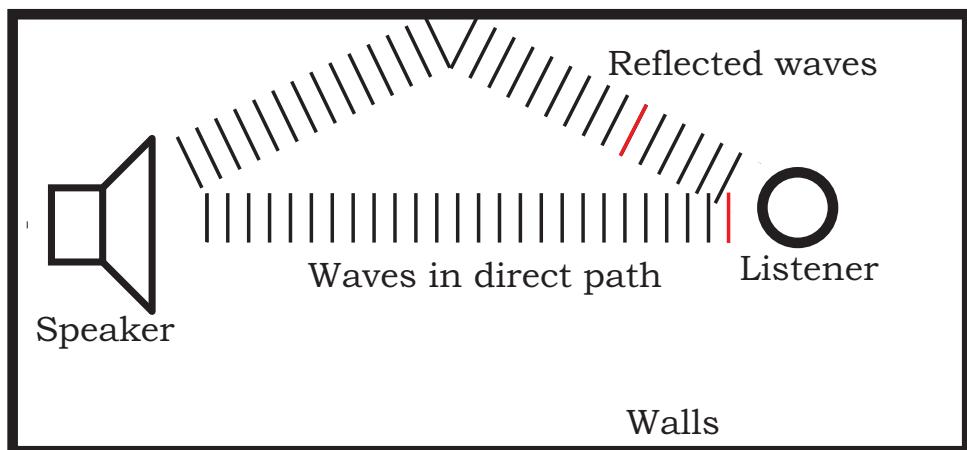


Figure 1.2: Sound traveling towards a listener along different paths²

Since a given material absorbs and reflects sound waves differently at different frequencies, and since reflected waves are attenuated at a rate given by its frequency, according to Stoke's Law of sound attenuation, sound waves will be affected by interference at the listeners position depending on the frequency. **sentence could use a touch of restructuring** These phenomena can have a quite significant effect since a wave with a frequency of 50 Hz can behave very differently from one with a frequency of 5000 Hz. Since music in general feature a pretty wide spectrum of frequencies³, the frequency response of a given piece of music can change a lot by traveling through a room. Furthermore this behavior can vary significantly based on: the position of the speaker, the size of the room, the building materials, and the interior decoration of the room. These changes of the frequency response might not be obvious to the average listener, who just wants to sit at his/her favorite listening spot.

I dont know about this reverberation section There is an important phenomenon that provides a general idea about the behavior of a room and it is independent from the listening point, the reverberation. The reverberation is the greater or the less persistence of the sound when a sound source stops emitting, and it depends on the intensity of the sound source and the threshold of hearing.

1.3 Summation of Problem

In the previous sections it has been shown that a piece of audio can often be altered differently during playback depending on the frequency. Furthermore, the processes which alters the characteristics of the sound might not be immediately obvious or even fixable for the user of the sound system. Since this is a 5th semester project in Electronics and IT, the project is meant to revolve around an electronic system that can interact with its surroundings by measuring some form of signal, processing this data, and generating a new signal based on this processing. For this reason It has been decided to work with a system that can measure the frequency response of a sound system, and correct the frequency response based on these measurements. The following chapter will describe different theories and technologies that relate to designing such a system.

²The lines representing the sound waves can be seen as tangents to the crests of the wave

³20 Hz to 20 kHz

Theory 2

This chapter will contain the analysis of the theories that make the project possible.

2.1 Equalizer

In this section the focus will be on analog equalizers, to give a basic understanding of how an equalizer works.

In the field of audio, an equalizer is a device that modifies the frequency response of the signal, and adapts it to the user, solving all kinds of problems. These devices are able to either amplify or attenuate one or more frequency bands.

There are different types of equalizers: High-pass and low-pass filters, shelving filters, graphic equalizers and parametric equalizers.

A high-pass filter is an electronic circuit that passes high frequencies and attenuates low ones. The same for the low-pass filters, that allows low frequencies to pass but attenuate the high ones [16].

On the other hand, shelving filters (figure 2.1) are used as tone controls, as they can attenuate or amplify a signal above or below a certain frequency [16].

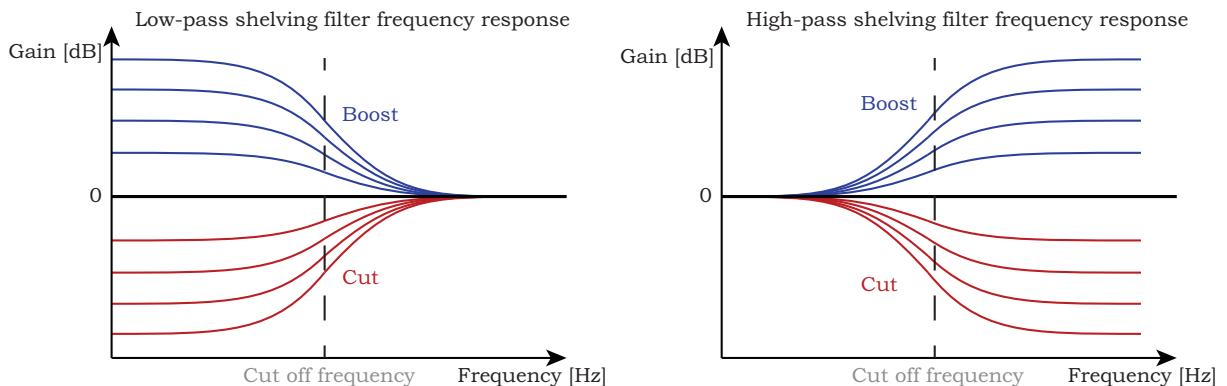


Figure 2.1: Shelving filter.

A graphic equalizer seen on figure 2.2, is composed by band-pass filters. It takes the name from the physical position that the faders are placed, making it possible to see in an easy way the changes made [16].

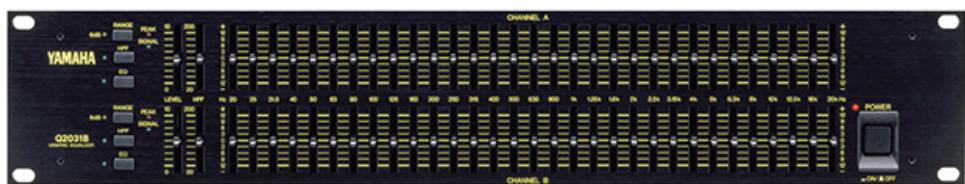


Figure 2.2: Graphic equalizer. Image from [14]

Finally, the parametric equalizer seen on figure 2.3, allows the individual control of three main parameters of the filters that compose the equalizer: amplitude, center frequency and bandwidth [16].

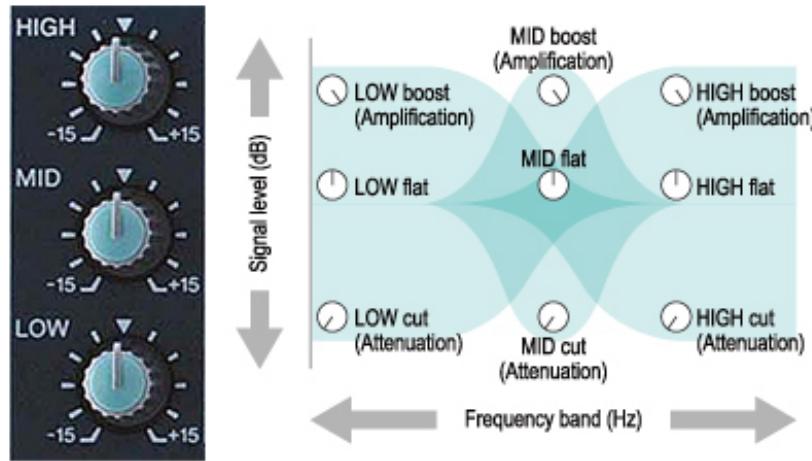


Figure 2.3: Parametric equalizer. Image from [14]

The analog filter design is mainly limited by the cost of the different elements, so a selection of better components will lead to a better equalizer.

It is well known that most of the signals contain noise, but analog filters have the disadvantage that they add noise to the signal through for example heat and electrostatic noise [7].

2.2 Digital sound

When sound has to be transported over longer distances, or stored for later use, it might be necessary to convert it into a digital signal. This can be done by converting the sound to an analog signal through a microphone and then converting it to a digital signal through an ADC (analog to digital converter). The digital signal can then be stored as a string of bits, which can be read by computers and for example be filtered or shared via the internet. Furthermore it is possible to make calculations based on the digital signal such as equalization or addition of multiple signals.

Sampling frequency

To ensure that the audio signal can be recreated as an analog signal, the sampling frequency has to be taken into account. The sampling frequency determines the number of samples, there has to be played per second. According to the Nyquist sampling theorem[19] the sampling frequency should be above two times the highest frequency of the signal. If this is not the case, some frequencies could be represented as a different frequency, as seen in figure 2.4.

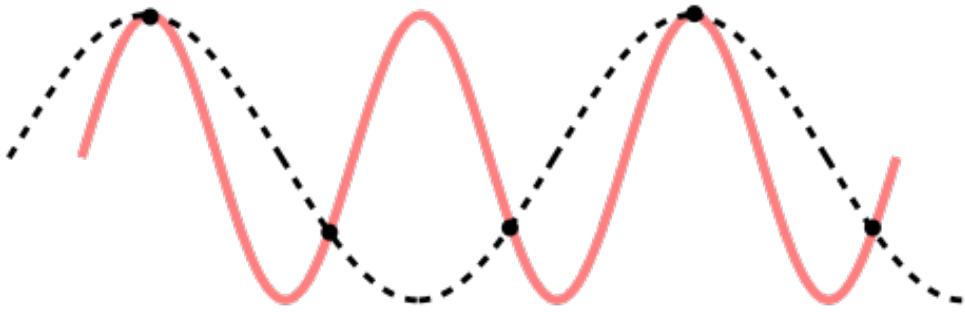


Figure 2.4: Falsely represented signal, due to the sampling frequency being to low. Figure from source [18]

Since the Nyquist sampling theorem states that the sampling frequency should be at least twice the highest frequency of the signal, the sampling frequency should be at least 40 kHz, when sampling audio signals. According to the standard IEC-60908[5] the sampling frequency of sound should be 44,1 kHz. This way audio signals can be recreated up to 22,05 kHz.

Quantization

When storing audio in a file it is defined by a string of bits that matches the amplitude of a specific sample. Here the amount of bits used to determine the sample, also known as bit depth, is a big factor in the quality of the sound. This means that the amount of bits used per sample determines how well the sound will be when converted to an analog signal. For a 4 bit signal, there would therefore be 16 different values for a sample, and for a 16 bit signal there would be 65536. An illustration of the correlation between an analog signal and its representation in bits, can be seen in figure 2.5.

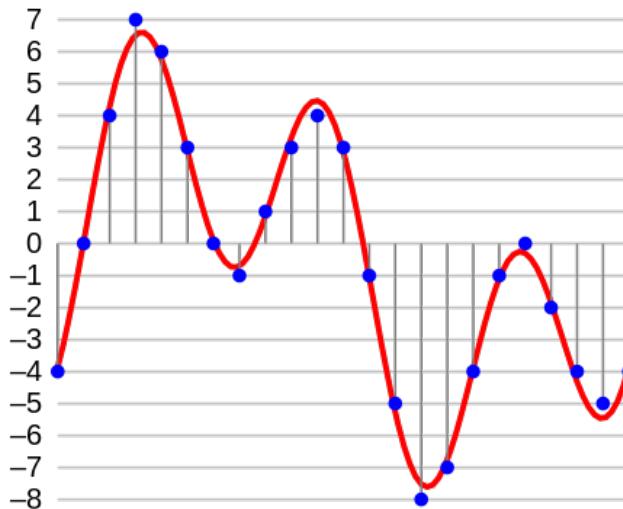


Figure 2.5: Audio signal and its representation in bits. Figure from [17]

CD quality sound is usually stored in a 16 bit resolution. In sound studios they use a bit depth of 24 or above to ensure a proper D/A conversion. If there is a need for adding two 16 bit signals together, 17 bits are needed to avoid clipping.

When audio is converted from analog to digital, there can occur an error in the sampling process. This happens because the digital value might not be exactly the same as the analog value. This

error is referred to as quantization. The quantization error can be $\pm 0,5$ times the value of the least significant bit(LSB). The signal-to-quantization-noise ratio (SQNR) can be calculated as seen in equation 2.2.1.

$$\text{SQNR} = 20 \cdot \log_{10}(2^Q) \quad [\text{dB}] \quad (2.2.1)$$

where:

$$Q = \text{Signal bit-dept} \quad [b]$$

Using the equation, a 1 bit signal would have an SQNR of 6,02 dB. For every bit added to the bit dept the SQNR goes up another 6,02 dB.

2.3 Digital signal processing

Digital signal processing and analog signal processing both is a description of a process to modulate an input. Digital signal processing utilizes digital filters to achieve this modulation. This makes digital signal processing much more versatile than analog signal processing because digital filters can both emulate analog filters, and because it is not limited to passive and active components, but to basic arithmetic, can therefore do a lot more[15]. Additionally there are no outside noise effecting a digital signal, which therefore does not have to be taken into account.

Digital filters

Finish this when lectures about digital filters are done.

Filters are used to selectively choose some wanted frequencies to be allowed through, and cut out other frequencies. Active filters can also be used to amplify or attenuate the specified frequencies.

A digital filter takes the input sequence of numbers from sampling the signal and computes a new sequence of numbers based on the filters transfer function. However digital filters cannot amplify a signal, only scale it up to the maximum bit value determined by the bit debt. Digital filters can only do basic operations such as: addition, subtraction, multiplication and division, more complex functions must be a combination of these [15] [12].

With digital filters the accuracy of the calculations of the functions can be controlled much more precisely than analog filters. The more complex a function and the closer the more accurate the function needs to be to the ideal, the longer it takes to compute. High computation times limits digital filters usability for real time applications. For real time applications either the filters complexity and accuracy needs to be lowered to decrease the computation time, or a fast digital signal processor is needed [12].

Due to the fact digital filters are calculated in a processor, they can be changed easily compared to analog filters, where the passive or active components would have to be changed.

new stuff, might be changed

There is generally two ways of implementing a digital filter: FIR filters (Finite impulse response filter) and IIR filters (Infinite impulse response).

IIR filters have an impulse response that continues forever, just like analog filters made from resistors, capacitors and inductors.

This means that an IIR filter can be designed by taking the transfer function of a known analog filter and transforming it to the Z-domain using Z-transformation and later to the digital domain using reverse Z-transform.

In this way, a digital filter can be given be a difference equation, which is a relatively easy type of equation to solve for a computer to solve given that the computer is able to store a certain number of previous inputs and outputs. In that sense a IIR filter has to have some kind of "feedback".

The other type of filters, FIR filters, on the other hand, have impulse responses that becomes zero after a certain point.

For FIR filter with an order n , the impulse response will consist of $n + 1$ samples. the output will be computed as a convolution sum of the impulse response and the last $n + 1$ input samples.

FIR filters has several advantages over IIR filters. The impulse response of a FIR filter can be based on an ideal impulse response. This means that high order FIR filter can closely resemble ideal filters. FIR filters cannot be unstable since the impulse response will always be zero after a certain point. It is also very easy to design a FIR filter with a linear phase response by making the impulse response symmetric.

However, these advantages comes at a cost. For a FIR filter to have as sharp a cutoff as a IIR filter, the FIR filter needs to be several orders bigger. Because of this it is more computationally expensive to implement a FIR filter than a IIR filter.

2.4 The amazing world of sound cards

In order for a computer to play and record audio, it needs a device that can convert analog sound to digital, and vise versa. A sound card also allows the user to perform some degree of digital signal manipulation, where the most common would be to manipulate the audio with an equalizer.

A sound card has a lot of interfaces, depending on the type of sound card. One of these could be a digital input or output from a CD-ROM drive or a MIDI(Musical Instrument Digital Interface). Depending on the type of sound card and the price range, the interfaces of the sound card differs, but the most common is analog input to a microphone or analog outputs to loudspeakers.

Since a sound card performs its operations digitally, all analog signal needs to be converted to digital in order to make recordings, and digital signal must be converted to analog for audio playback. This is done with an ADC(Analog to Digital Converter) and a DAC(Digital to Analog Converter). The resolution of these components depends on the price range, but a resolution of 16 bits is needed for audio in CD quality [11][20].

For the digital audio processing there are mainly two options. Either to use a computers CPU or to use a Digital Signal Processor(DSP).

In order to control and issue commands to the sound card, some form of device driver is needed. The driver handles the data connection between the sound card and an operating system. The driver needed depends mostly on the users OS. For Windows users the drivers will generally be written by the manufactures and will therefore be licensed. Linux users on the other hand,

have a bit more freedom. The most widely used driver is called ALSA (Advanced Linux Sound Architecture). One of the bigger advantages with the ALSA-driver compared to most Windows drivers, is that it allows the user to directly interact with the kernel and therefore the hardware [2].

2.5 Platforms for digital signal processing

In order to correct the soundlevels of the loudspeaker compared to the measured sound, a platform that can process and compare the audio is needed.

For real time digital signal processing the highly specialized digital signal processors (DSP) was created, these had a lot less functionality than normal CPU's but were faster for processing digital signals. Today however CPU's have gotten so fast that they can be used for most real time digital signal processing.

Microcontroller

A microcontroller is a circuit created for and used to control embedded systems. A microcontroller usually contains a CPU and some random access memory (RAM), besides that it has input output ports, so that it can be used in a embedded system[9].

A microcontroller will typically be set to run a single program indefinitely. This makes it possible to use the CPU¹ quite effectively. It also removes the overhead of trying to effectively schedule the CPU usage between different processes.

A very widely used series of microcontrollers are the Arduinos. The most common Arduinos are equipped with an ARM CPU with a clock speed in the range of about 8-80 MHz.

Computer

Another option for a platform for digital signal processing, could be a fully fledged computer. Besides a CPU and RAM, a complete computer will often have some form of permanent memory, network capabilities, a dedicated graphical processing unit (GPU), and multiple ports for communication with peripherals. A complete computer is also easily implemented with a operating system, which makes it easier for the CPU to handle multiple simultaneous processes with the help of a kernel.

While you normally think of a desktop PC or a laptop, when you hear the term "computer", There has recently been an emergence of small single board computers, where you have a complete computer implemented on a single PCB. The most well known example of a single board computer is the Raspberry Pi.

¹which is often somewhat slow compared to CPU's used in PC's

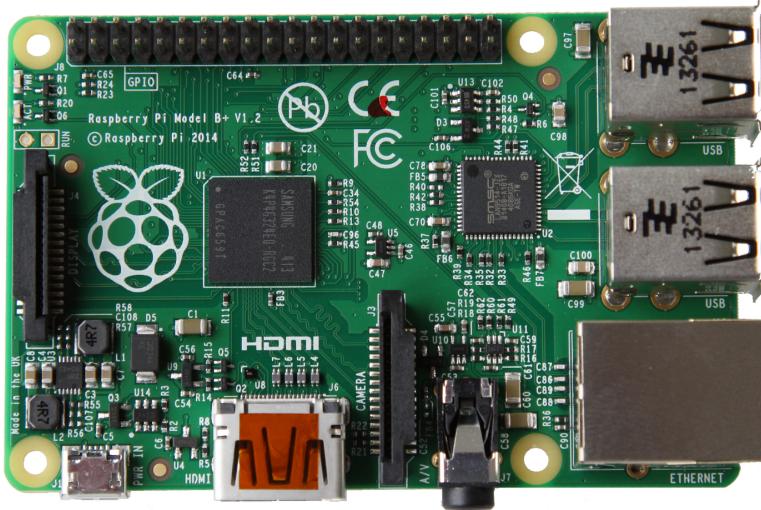


Figure 2.6: Rasberry Pi model B+. Image from [1]

There are multiple versions of the Raspberry Pi, each version with slightly different hardware. Each type has an ARM processor with a clock speed of at least 700 MHz which is several times faster than even the fastest Arduino. This fact might help with negating the extra overhead associated with running a program in an operating system instead of an embedded system. Another benefit of using a computer, is that it might be possible to use already existing drivers to communicate with a sound card.

2.6 Wireless communication

While the system may not necessarily need a wireless connection. Yet do to the curriculum and because it is expected to be of convenience, some useful forms of wireless communication will be studied to a basic level, so that it is possible to make a suitable decision for the design phase of the Self adjusting loudspeaker system.

Commonly used wireless systems

Because there is lot of different types of communication systems, some which is not relevant for the scope of this project, we will try to list some of the more commonly used systems, which is expected to work within the scope of the project.

Wireless Local Area Network

In general most of people with wireless access to the internet, are using either cellular or Wireless Local Area Network (WLAN for short). Cellular communication will be discussed later. The most common implementation of WLAN is through WIFI, which is based around the IEEE 802.11 standard. There are many varriations in 802.11 such as

do we want elaboration on frames and CSMA/CA here? or later if we "choose" to use wifi later on in the project

Wireless Personal Area Netowrk

PersonalArea Netowrk or PAN for short, and the wireless version is call WPAN (wireless PAN). The idea behind WPAN is like that of the WLAN, except the range is typical limited to *personal* space. The range is a lot shorter compared to WLAN, often around 10 meters (but can be wider for industrial purposes), where WLAN often have a range that of a house. WPAN is based on IEEE 802.15 standard. Probably the widest implementation of WPAN is the bluetooth standard (the IEEE 802.15.1 standard).

Bluetooth

Bluetooth was originally meant as a wireless alternative of the RS-232 standard. The RS-232 is no longer in use by the average person, due to the slow transmission speed, short maximal cable length and large physical connector size. It is still in use in commercial usage. So bluetooth is designed to be a short range wireless communication standard, working in areas with interference. This is done using something frequency hopping spread spectrum (FHSS) in the 2.4 to 2.5 GHz ISM band, sending small packages at up to 79 changing frequency channels. It works in a master slave relation with up to seven slaves. Later on power consumption and bit rate have been improved significantly. It is possible to achieve bit rates up to 2.1 Mb/s [4] and with the use of 802.11 AMP (Alternate MAC/PHY) bit rates of 54 Mb/s is possible. Adaptive frequency hopping spread spectrum (AFH) was also introduced, mainly to avoid interference with WIFI.

Zigbee

is this worth mentioning??

Another usage of the IEEE 802.15 standard, 802.15.4 to be exact, is the zigbee protocol. This protocol is meant for begin low on price, power consumption, bit rate and low range (though a mesh grid can be a solution to this). So this technology is not useful for this project since the bit rate is about 250 kbit/s using the 2.4 Ghz ISM band. While with the right compression of the sound data, it its possible to achieve a real time system. It would create serious limitations of the system, eg. sampling and transmitting raw sound data at the 44.1 kHz sampling rate with a bit depth of 16 bits is not possible, since this would require 705,6 kb/s. **Do we want to make this delimitation here, or should we wait to the delimitation chapter?**

Should we mention technologies such as 4G (and cellular in general) and radio broadcasting such as FM-radio?

Problem Statement 3

3.1 Project Scope

The preceding analysis, in chapter 2 of potential technologies, it is now possible to outline the project.

As mentioned earlier, the goal of the project is to make a system that automatically can correct the frequency response of an audio system. It has been decided that the specific solution that will be described in this project, is a system that can measure audio in a room. The measured audio will then be compared to the original audio signal, and an equalizer will then be used to adjust for an uneven frequency response, and then correct imminent audio signal.

It would be preferable to be able to configure the equalizer for many different kinds of frequency responses. Therefore it has been decided to implement an equalizer using a series of digital band pass filters, that can be automatically adjusted independently. This is analogous to the graphic equalizers described in section 2.1.

Delimitations

A big advantage for the imagined system would be if the microphone could be freely moved around the room. Therefore, it has been decided to include some form of wireless communication in the implementation.

It has been decided to use WIFI (one of the IEEE 802.11 standards). The decision is based on the two most reasonable communication standards WIFI and bluetooth, of these WIFI has been chosen. WIFI is chosen because it has the highest bandwidth making the initial design easier, because bandwidth is expected not be critical and optimization can be made late on if need be.

For this reason it has been chosen to implement the system on two separate devices that can communicate wirelessly: one with a microphone to measure sound system, and another device to do the signal processing.

To save time and money, and to make it easier to implement a prototype it has been decided not to make everything in the project from scratch. It would be more relevant to use already existing hardware.

It has been decided to implement both devices on Raspberry Pi single board computers (specifically a Raspberry Pi 3 for the stationary unit, and a Raspberry Pi Zero W for the microphone unit). The conversion from analog signals to digital will also be done using already existing sound cards and drivers. The same goes for microphones and wireless transmitters/receivers.

In general the hardware will be chosen mostly with availability, simplicity and ease of use in mind.

Since it might not be possible to correct the whole audio spectrum, it has been decided to put a lower limit on the frequencies that will be corrected. [where and why there](#).

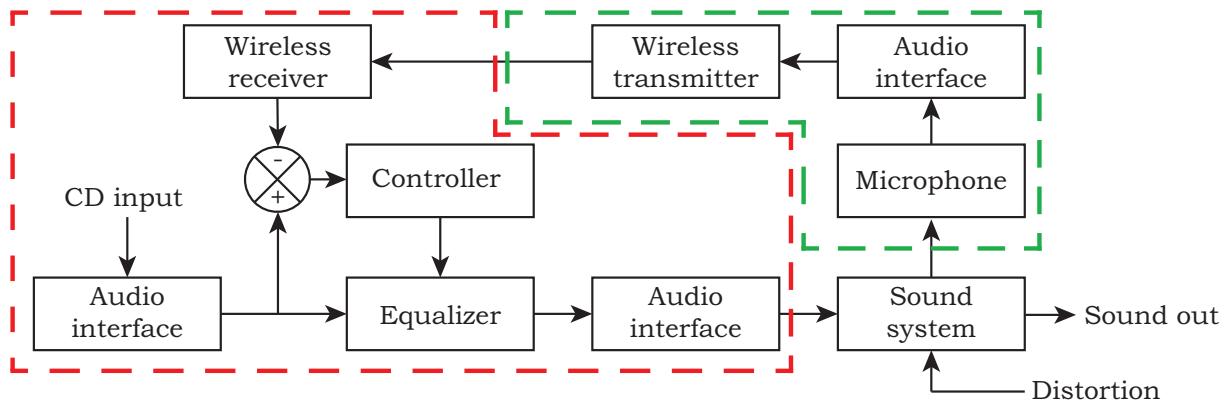


Figure 3.1: Diagram of project

[table over chosen hardware](#)

[more if necessary](#)

3.2 Problem statement

How does?

Requirements specification 4

In this chapter the specific requirements for the project will be established. This will include wanted functionality and how this will be divided into subsystems, with their own requirements.

4.1 Functional requirements

The system should be able to:

- 1.1 Receive an analog audio input signal.
- 1.2 Output a corrected analog audio signal in CD standard quality.
- 1.3 Correct the frequency response of the signal compared to the input within **TBD** dB.
- 1.4 Change the frequency response by **TBD** dB for each alteration **guess: 1 dB**
- 1.5 Modulate frequencies from **TBD** Hz to **TBD** Hz **guess: 30 Hz - 20 kHz**
- 1.6 Correct the signal within **TBD** ms.
- 1.7 Run on two separate devices that uses wireless communication.

4.2 Division into subsystems

asdf

Based around the functional requirements outlined above, the functionalities of the system have been divided into four different subcategories. These can be seen in figure 4.1.

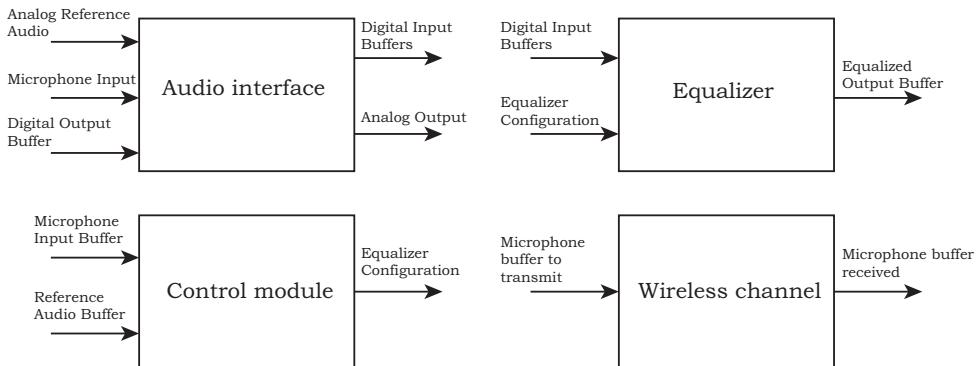


Figure 4.1: some description

Audio interface(s)

This module is tasked with converting the analog reference signal and the signal captured by the microphone to digital signals. Furthermore it has to convert the corrected digital signal to an analog output signal. It will also be tasked with storing the sampled input in correctly sized buffers for further use by the system.

Equalizer

This module is tasked with dividing the input signal into **a number of** frequency bands, using parallel band pass filters. Each band pass filter will need to have an adjustable gain.

Controller

The controller module is supposed to perform frequency analysis on a buffer of both the measured audio signal and the input signal. The results of these analyses are compared, and the equalizer is configured to try and minimize the differences of the frequency responses of the two signals.

Wireless channel

The last of the four modules has the job of ensuring that the two raspberry pi's can communicate over WiFi at all times. More specifically it needs to make sure that the microphones device is always transmitting its measured audio signal to the stationary device. For this to be possible the wireless channel needs a bit rate of at least:

$$\text{sample rate} \cdot \text{bit depth} \cdot \text{mono input} = \text{bit rate} \quad (4.2.1)$$

the sample rate and bit rate has been decided to be 44.1kHz and 16 bit respectively **ref to something**. Since only one microphone is in use, and more is not functionally necessary for the system, only mono is recorded. which means the necessary bit rate for continuous streaming of the sound audio signal is:

$$44.1\text{kHz} \cdot 16\text{bits} = 705.6\text{kbpss} \quad (4.2.2)$$

REWRITE THIS, needs to send faster than the bit rate of 705.6 kbps, to make time for the calculations and space for headers. For this the bit rate is set to 10 times higher ie. 7.056 Mbps

2.1 Audio interface

2.2 Equalizer

2.3 Controller

2.4 Wireless channel

The requirements for the modules is an early draft

4.3 Requirements for audio interface

The audio interface module should be able to:

- 3.1 Convert an analog audio signal to a digital signal.
- 3.2 Convert an digital audio signal to a analog signal.
- 3.3 Sample audio signals with a sampling frequency of 44,1 kHz and a bit debt of **TBD** bits.
- 3.4 Store input and measured data in two buffers of **TBD size**.
- 3.5 Convert the signal within **TBD** ms. **Less then the total delay**

4.4 Requirements for equalizer

The equalizer module should be able to:

- 4.1 Capable of dividing a digital signal into 10 frequency bands that covers the frequency range from 30 Hz to 20 kHz.
- 4.2 Adjust the amplitude of the individual frequency bands with - **TBD** dB.
- 4.3 Have an amplitude response of ± 1 dB when all bands are at maximum gain.
- 4.4 Make the frequency adjustments based on input from the controller module.
- 4.5 ****Adjust the signal within **TBD** ms. **Less then the total delay******
- 4.6 Have a maximum group delay lower than **TBD** ms.

4.5 Requirements for controller

The control module should be able to:

- 5.1 Analyze frequencies from **TBD** Hz to **TBD** Hz **guess: 50 Hz - 20 kHz**
- 5.2 Change the frequency response by **TBD** dB for each alteration **guess: 1 dB**
- 5.3 Correct the frequency response of the signal compared to the input within **TBD** dB.
- 5.4 Correct the signal within **TBD** ms. **Less then the total delay**

4.6 Requirements for wireless communication

The wireless channel module should be able to:

- 6.1 transmitting and receiving data between the two raspberry pi's at a constant rate of 7Mb/s.
- 6.2 Communicate using the IEEE 802.11 standard.
- 6.3 Send data with **TBD%** package loss or below.

Design 5

This chapter will go through the design of each subsystem for the self adjusting loudspeaker. For each subsystem the methods and reasoning behind the choices made will be made clear.

5.1 Design of audio interface

In this section, the design and implementation of the audio interface will be covered.

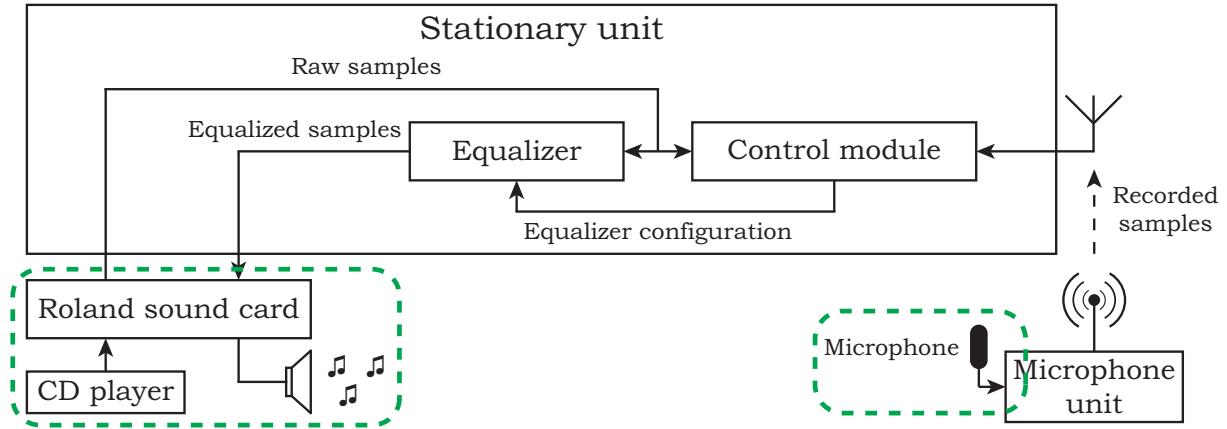


Figure 5.1: The green dotted line indicates the current place in the system

The purpose of the audio interface is to convert a reference analog signal, and the signal captured by the microphone, to digital signals. These digital signals are to be saved, so the controller can analyze and compare them. Finally it must be able to convert a digital signal to an analog signal, after the equalizer has made its potential adjustments.

The module can be split into two separate parts. A part that records the reference signal and plays the corrected signal, and the part that records the signal being played, and makes it ready to be transmitted by the wireless channel.

The recording of the reference signal and playback of the corrected signal is done by the UA-25EX sound card by Roland[8], which contains a 24 bit DSP with a sampling rate up to 96 kHz and two channel input and output. The sound card can be seen in figure 5.2.



Figure 5.2: Roland UA-25EX

The recording of the signal being played is done by a AK5371 USB microphone [3], which has 2 channels, a 16 bit A/D converter and a sampling rate of 44,1 kHz.

In order to communicate between either the Roland sound card, or the USB-microphone and their respective Raspberry PI, the ALSA-API, as described in section 2.4, is used. To ease the understanding of the designing of the audio interface, some basic ALSA terminology will be explained. Instead of working in samples, a term called frames is used. A frame consists of a sample from each channel, and since two channels is being used with two bytes in each sample, the size of one frame is in this case 4 bytes.

When streaming audio, or any kind of data for that matter, a constant transmission of frames is needed. This is not feasible since the CPU most likely has other jobs as well, and if the sample is not received at the exact right moment, the audio stream will have gaps between the samples. This can be resolved by using a buffer, so data can be transferred to the sound card in batches, which will result in latency. The buffer is used as a ring buffer so new frames just needs to be put in after the previous frames. The size of this ring buffer will determine the maximum amount of latency.

Another problem occurs when applying the ring buffer. Assuming the ring buffer is completely filled with frames when it is completely empty, it is not guaranteed that the new data will arrive in time for the new first frame to be played. Therefore ALSA uses a term named periods. A period is either a specific amount of frames or time between status updates. If the period size for example is 512 frames, an interrupt will be made every time 512 frames has been played, meaning it is time to fill the buffer with new data. This also means that the ring buffer should at least have the double size of the period size, since it should be possible to fill the ring buffer with new data right after the interrupt has been made.

This leads to the design the audio interface, where the amount of latency in the system has to be considered. For the recording using the USB-microphone, the latency does not matter, since the data is being used, to adjust the equalizer. On the other hand, the Roland sound card must be able to both record and play in "real time". In this case, real time is when the recorded data is being played fast enough for the user not to notice a time difference between pressing play and samples being played by the sound card. A maximum latency of **100 ms** is considered to be enough. **I know it's a guess but can we find any source on how quickly a person would notice, or would it even matter if everything is delayed this amount of time?**

The latency is the time difference between a frame is put into the buffer and the same frame is played. Since the amount of frames being played every second is determined by the sampling rate, the maximum latency can be found as:

$$\text{latency} = \frac{\text{ringBuffer}_\text{size}}{2 \cdot \text{samplingRate}} \quad [\text{s}] \quad (5.1.1)$$

In order to play in stereo the sampling rate must be doubled, which is the case with the ring buffer where the samples are interleaved. It has been decided to use a ring buffer with a size of 2048 frames, which is 4096 samples, and since the sampling rate is 44,1 kHz, the maximum latency is:

$$\text{latency} = \frac{4096 \text{ samples}}{2 \cdot 44100 \frac{\text{samples}}{\text{s}}} = 46,4 \text{ ms} \quad (5.1.2)$$

As seen, the latency is well below the requirement, and with the chosen ring buffer size of 2048 frames, and period size of 512 frames, it is now possible to design the actual audio interface.

5.1.1 Recording of reference audio and playback of corrected audio

As mentioned, the recording of the reference audio and playback of the corrected audio is done by the UA-25EX sound card, which is controlled by the stationary unit. First the sound card must be initialized for both recording and playback. Starting with initialization of the recording, hardware parameters of the sound card must be set before it is usable. This is done by first loading the default parameters and afterwards edit the parameters of interest. These parameters are in this case the streaming mode, the sampling rate, the format, the number of channels, and how to read the sampled data. Since the reference audio needs to be recorded, the streaming mode is set to "capture". Since it has been chosen to use cd quality, the sampling rate is set to 44100 Hz, and the bit-depth is set to 16 bits per sample. In this case the format is little endian since the stationary unit has an ARM-processor, which uses little endian. It has been decided to record in stereo because most music is in stereo. It would be a shame to simply discard one of the channels, and therefore the number of channels is set to two.

The initialization of the sound card in playback mode is done in almost the same way as in recording mode. The only difference that the streaming mode is set to "playback" instead of "capture". When the sound card has been initialized to both record and playback, the actual reading and writing of data can take place. In figure 5.3, the general flow of the record/playback can be seen. Here the buffer is accessible by other parts of the system, so adjustments of the frames are possible.

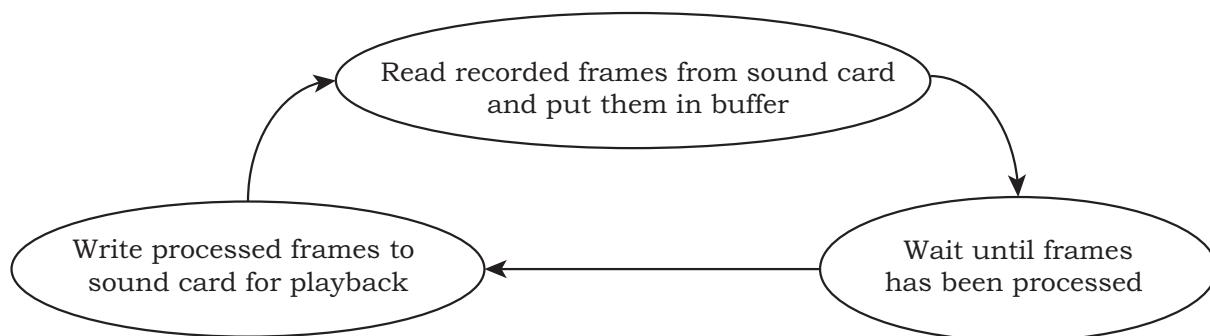


Figure 5.3: Flow of recording reference audio and playback of corrected audio.

Before filling the buffer with recorded frames, it is necessary to check how many frames are available for capturing, so the number of recorded frames won't exceed the size of the buffer. Afterwards the number of available frames are read and stored. This can be seen in code example 5.1.

Code example 5.1: Reading of available frames

```

1 int avail = snd_pcm_avail_update(recordHandle); //Updates the amount of available frames.
2
3 if (avail > 0) {
4     if (avail > frames){ //If there are too many frames, only read the maximum amount possible.
5         avail = frames;
6     }
7     snd_pcm_readi(recordHandle,buffer,avail); //Reads interleaved frames and stores them in the buffer.
8 }
```

When the frames have been processed by the control unit, they are written to the sound card for playback. This is done almost the same way as when reading frames. The amount of available space in the sound card buffer is checked before writing any frames. If the free space exceeds the amount of frames, all the frames are written, else an overrun will occur. With a ring buffer size four times as large as the period size, this should not be possible. Writing frames to the sound card can be seen in code example 5.2. **space check can maybe omitted, but need test to be sure.**

Code example 5.2: Writing available frames

```

1 int avail = snd_pcm_avail_update(playbackHandle); //Updates the amount of available frames.
2
3 if (avail > 0){
4     if (avail > frames){
5         avail = frames;
6     }
7     snd_pcm_writei(playbackHandle,buffer,avail); //Writes interleaved frames to the sound card.
8 }
```

With the setup and functionality of the stationary unit done, the microphone unit can now be designed.

5.1.2 Recording of played signal

The recording of the played signal is done by the USB-microphone which is controlled by the microphone unit. The task is to record a number of frames for transmitting by the wireless channel.

On figure 5.4 can the overall functionality be seen.

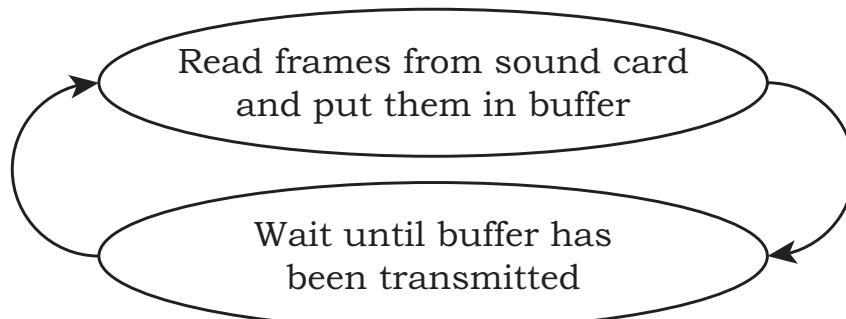


Figure 5.4: Flow of recording the played signal.

Since ALSA is being used for the USB-microphone as well, both the initialization and use is almost the same as for the stationary unit. The only difference is that the USB-microphone is initialized to use one channel, and the amount of samples is therefore halved.

Now that the functionality for the sound module of both the stationary and microphone unit have been designed, the communication between the units will be explained.

5.2 Wireless interface design

full code can be found somewhere important In this section the design of the wireless connection between the control unit and the microphone unit will be described. The wireless connection must be designed to live up to the specifications from section 4.6. From the specifications the following requirements needs to be fulfilled: The wireless connection needs a data rate of at least **2 Mbps**, it should be using a WiFi protocol and should have a maximum package loss of **TBD** package loss.

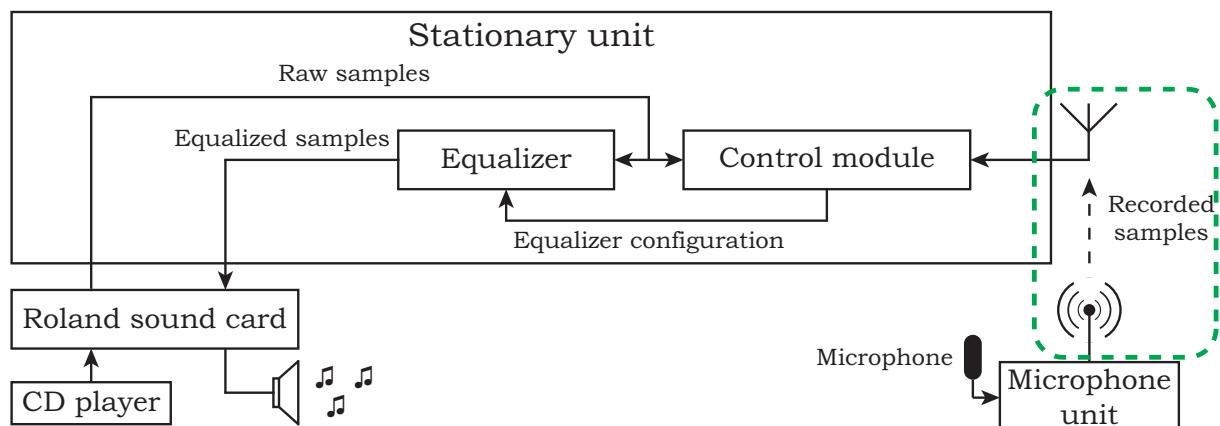


Figure 5.5: You are here

5.2.1 Connection structure and protocol suite

There are two obvious possibilities for setting up a connection between the two Raspberry Pi's. Either a direct connection, where one of them acts as a router, or a indirect connection through an already existing router. Since the two Raspberry Pi's doesn't need to be connected to anything but each other, and since a preexisting router would not be dedicated to this connection, which means some of the routers bandwidth capacity would be used for other purposes. A direct connection is more reliable and therefore preferable.

To setup a direct connection between the two Raspberry Pi's, the stationary unit is set to Ad hoc mode. Ad hoc is latin for: "For this specific purpose" and is used in wireless networks to describe a connection that does not use any preexisting infrastructure. The stationary unit is used as host and therefore an SSID (service set identifier) is set. To make it possible for the microphone unit to connect, there is a need for an administrator of IP addresses. For this to be done automatically an DHCP (Dynamic Host Configuration Protocol) is installed. The final step for establishing a connection, is that the microphone unit also needs to be set to Ad hoc mode and connect to the stationary unit using its SSID.

With a connection established between the two Raspberry Pi's data can be send, for this purpose

a transport protocol suite is needed. For the suite many options exists. The two most prominent are UDP (User Datagram Protocol) and TCP (Transmission Control Protocol).

The protocol suite TCP is the most widely used protocol suite on the Internet, because compared to UDP it offers a lot of options for the users of the connection that UDP do not. These options include congestion control, connection control, retransmission capabilities and transmission windows.

When a connection is established a "three-way-handshake" is used as seen on figure 5.6. This is done to insure the connection is possible, and to determine parameters for the data transmission. These parameters determines things such as the size of the segments which is transferred and the window scale factor, used for determining the amount of bytes transferred before an acknowledgement is needed for the receiver. These options and parameters makes the TCP a connection-oriented protocol. The TCP header is at least 20 Bytes in size, but can be bigger if some additional options are in.

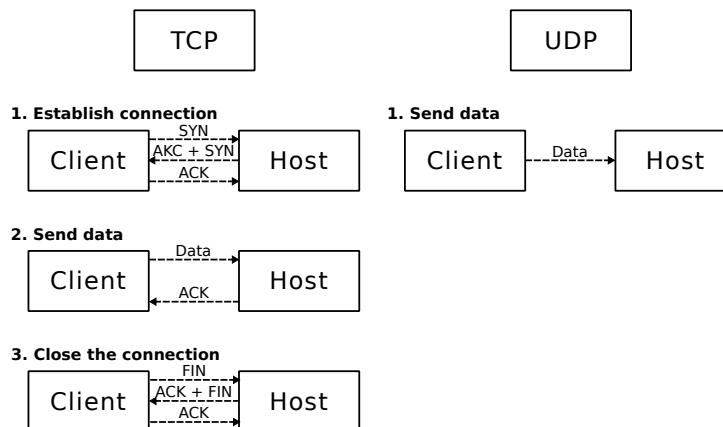


Figure 5.6: TCP hand need cite aau Jens Slides

The protocol suite UDP is connectionless which compared to TCP means that the connection is not set up prior to the data transfer that also means the every time a transmission is done a new connection is established. The UDP header only contains a port description, a check sum and a length. This means that the UDP header is very small compared to TCP header, the UDP header is a minimum of 8 Bytes.

The complete system is a real time system, and is established over an ad hoc connection. This means that there are no use of the TCP protocols features such as congestion control and retransmission, other than this the TCP-header is much larger than that of UDP-header. Making the UDP protocol the choice for the complete system. This reduces overhead and complexity of the system.

5.2.2 Connection using socket programming

To establish a UDP connection between a host and a client, the host must establish a socket as seen in Code example 5.3.

Code example 5.3: UDP socket creation

```

1 #include <netinet/in.h>
2 // Creates socket determines: Family(ipv4), type(connectionless), protocol(UDP):

```

```

3 | if((servSock = socket(PF_INET,SOCK_DGRAM, IPPROTO_UDP)) < 0)
4 |     error("socket creation\n");

```

The socket is created, using the socket function, this function takes three inputs: family, type and protocol. The input parameter `PF_INET` specifies the socket to use the IPv4 protocol, `SOCK_DGRAM` and `IPPROTO_UDP` specifies the socket to be UDP protocol suite. **is `sock_dgram` and `ipproto_udp` two sides of the same coin??**

The host now has to bind the socket to the address family, an IP address and a port, seen in 5.4.

Code example 5.4: UDP binding socket

```

1 | struct sockaddr_in server; // Struct to store data about server
2 | server.sin_family      = AF_INET;    // Determines family of socket
3 | server.sin_addr.s_addr = htonl(INADDR_ANY); // Determines client address (any)
4 | server.sin_port = htons(atoi(argv[1])); // Determines port to open socket on
5 |
6 | // Binds socket to specifications above:
7 | if(bind(servSock, (struct sockaddr *) &server, sizeof(server)) < 0)
8 |     error("Port is in use\n");

```

Before the socket can be bound the input parameters must be specified, this is done in a struct of the type `sockaddr_in` which is specified in the `netinet/in.h` library. **As can be seen in the code example 5.4 the family is and so is the port, the port is determinant as input to the c program -wat??.** The input address is set to `INADDR_ANY` which means it is specified to all available address on the network.

The bind function takes three inputs: the socket id of the socket specified previously, a pointer to the struct `sockaddr_in` `server`, and the size of this struct.

The host now waits to receive data from the client, in this case the microphone unit.

Code example 5.5: UDP Receive data from client

```

1 | #define x 1024           // Size of the sound buffer in shorts
2 | #define y x * 2         // Bytes in buffer
3 | int16_t Data[x];       // Array to store data
4 |
5 | struct sockaddr_in client; // Struct to store data about client
6 | clientSize = sizeof(struct sockaddr_in);
7 |
8 | // Receiving data:
9 | if((recvMsgSize = recvfrom(servSock,Data,y,0,(struct sockaddr *) &client, &clientSize)) < 0)
10 |     error("Error in receiving data\n");

```

The size of the array where the data is stored after being received is determined by the size of the audio values send from the microphone unit - see section 5.1.

The `recvfrom` function takes six inputs as can be seen in code example 5.5: First input specifies the socket id. The second specify what kind of data and where it should be saved. **The third determines how many bytes should be received, since it is `int16_t`, every value is 2 bytes in size thus y is (2048) twice the size of x, the reason for the y value is that each frame consists of two channels each defined to be 512 values, making the total 2048 bytes to be send. Making y the about of bytes to be transmitted and x the number of values. The fourth determines flags, but is not used in the UDP protocol?.** The fifth determines the source address and saves it into the

`client struct` similar to the server struct, see code example 5.4. The sixth inputs the size of the client struct.

Then For the client to connect to the host socket, it first has to create a client socket and bind to the right port. This is done in the same way as for the server see code example 5.3, and 5.4, the port has to be the same as that for the server.

For the client to be able to send data it has to setup a server struct with the IP and port of the host socket.

Code example 5.6: UDP Send data to server

```

1 #define x 1024      // Size of the sound buffer in shorts
2 #define y = x * 2  // Bytes in buffer
3 int16_t Data[x]; // Array to store data
4
5 struct sockaddr_in server;
6
7 server.sin_family = AF_INET;
8 // Determines family of socket
9 server.sin_addr.s_addr = inet_addr(argv[2]);
10 // Sets the host IP address
11 server.sin_port = htons(atoi(argv[1]));
12 // Determines port for connecting to host with
13
14 // Transmit data
15 if(recvMsgSize = sendto(clientSock,Data,y,0, (struct sock_addr*) &server,sizeof(server)) < 0)
16     error("Error in sending data");

```

The address and the port, is inputs for the microphone unit's program, this could however be fixed to 192.168.1.1 and a predetermined port. This is possible since the microphone unit always has to connect to the stationary unit, which is hosting the ad hoc server.

As can be seen in the code example 5.6 the `sendto` function takes similar inputs as the `recvfrom` function, and the data transmit and received must both be the same size and type to ensure no mistakes happen. the difference is that the `sendto` function transmit the data to the already known IP and port of the host.

To make sure that the host computer does not read the data from the client incorrectly, due to different usage of endian. All data is converted from the host's byte order to network's byte order, and back again when received by the host.

To be certain that the data arrives at the stationary unit correctly and not corrupted, a checksum is implemented. The checksum is computed on the microphone unit, and transmitted with the rest of the data. On the stationary unit the data is again computed and compared to the checksum received by the microphone.

5.2.3 Partialtest of WiFi connection

With the WiFi connection established, the transmission speed between the stationary unit and microphone unit needs to be tested to see if it reaches the requirement of at least 2 Mb/s. The reliability of the connection also needs to be tested to see if package loss between the two units is an issue.

In appendix A a test is made to give an approximation of the bit rate between the two units. This

approximation is made by sending a large file consisting of 400 MB of data from the microphone unit to the stationary unit. Time is measured of how long the transmission takes to complete. An approximation of the bit rate is then calculated to be 4 - 5.5 MB/s. A much higher bit rate than the required 2Mbps (0.25MB/s).

In appendix B a test was conducted to find the amount of package loss between the two units. Of all the packages received by the stationary unit no package loss were recorded. Therefore it can be assumed that there is no need for any extra overhead to cope with this.

5.3 Design of Equalizer

In this section the design of the equalizer module will be described. The equalizer must be designed to live up to the specifications from section 4.4.

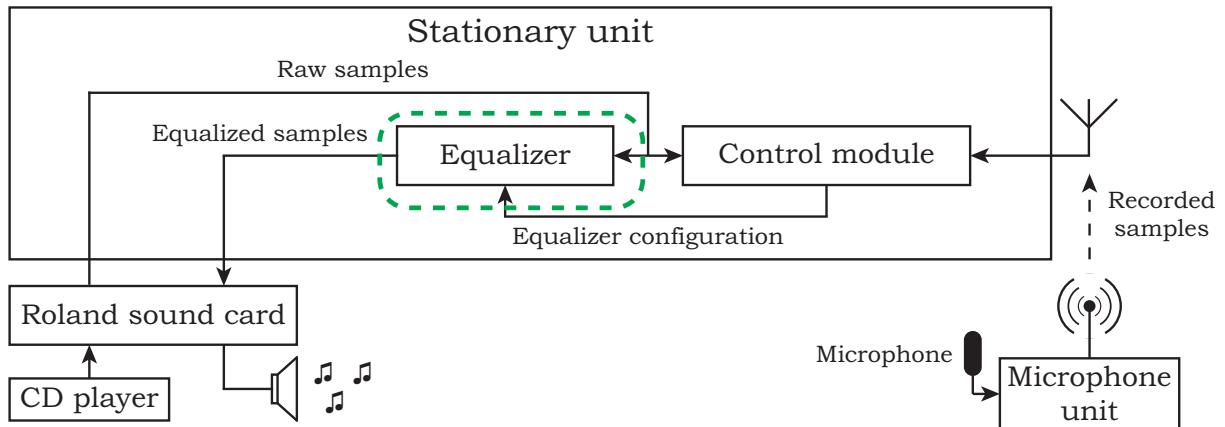


Figure 5.7: The green dotted line indicates the current place in the system

From the requirements specification, it is given that the equalizer should be divided into 10 frequency bands. These 10 frequency bands can be implemented using 10 bandpass filters.

5.3.1 Determining the center frequencies

For all the bands to be equally distinct from each other, the center frequencies of the bandpass filters must be distanced logarithmically from each other. The reason for this is that human hearing is logarithmic in its sensitivity to frequencies.

The center frequencies are calculated by starting from 1 kHz and then multiplying or dividing by two, so that each center frequency is placed an octave apart. This nicely covers the audible frequency range by having the lowest frequency band around $\omega_{c1} = \frac{1000\text{Hz}}{2^5} = \frac{1000\text{Hz}}{32} = 31,25\text{Hz}$ and the highest frequency range around $\omega_{c10} = 2^4 \cdot 1000\text{Hz} = 16 \cdot 1000\text{Hz} = 16\text{kHz}$

All the center frequencies can thus be written as $2^n \cdot 1000\text{Hz}$ where $n = -5, -4, -3, \dots, 3, 4$.

ω_{C1}	ω_{C2}	ω_{C3}	ω_{C4}	ω_{C5}
$2\pi \cdot 31,25 \text{ Hz}$	$2\pi \cdot 62,5 \text{ Hz}$	$2\pi \cdot 125 \text{ Hz}$	$2\pi \cdot 250 \text{ Hz}$	$2\pi \cdot 500 \text{ Hz}$
ω_{C6}	ω_{C7}	ω_{C8}	ω_{C9}	ω_{C10}
$2\pi \cdot 1 \text{ kHz}$	$2\pi \cdot 2 \text{ kHz}$	$2\pi \cdot 4 \text{ kHz}$	$2\pi \cdot 8 \text{ kHz}$	$2\pi \cdot 16 \text{ kHz}$

Table 5.1: Center frequencies spaced one octave apart

These ten center frequencies are similar to the center frequencies specified in the ISO 266 standard for octave band equalizers [6].

5.3.2 Determination of filter transfer functions

The passband of the bandpass filters should cover the whole frequency range from **TBD - TBD** specified in section 4.4, so that the equalizer will have a flat passband in the neutral configuration¹.

For each bandpass filter to handle an equally significant amount of frequencies, the cutoff frequencies of two filters next to each other should meet at the logarithmic mid point of their center. This is because human frequency sensitivity is logarithmic. Thus the edge frequencies of one of the bandpass filters can be formulated as:

$$\omega_{Hn} = \sqrt{\omega_{Cn} \cdot \omega_{Cn+1}} \quad [\text{rad/s}] \quad (5.3.1)$$

and

$$\omega_{Ln} = \sqrt{\omega_{Cn} \cdot \omega_{Cn-1}} \quad [\text{rad/s}] \quad (5.3.2)$$

Where ω_{Hn} is the higher edge frequency and ω_{Ln} is the lower edge frequency of bandpass filter number n .

Since the center frequencies are spaced exactly one octave apart, the ratio between the edge frequencies can be calculated as:

$$\omega_{Hn} = \sqrt{2 \cdot \omega_{Cn}^2} \iff \frac{\omega_{Hn}}{\omega_{Cn}} = \sqrt{2} \quad (5.3.3)$$

and

$$\omega_{Ln} = \sqrt{\frac{1}{2} \cdot \omega_{Cn}^2} \iff \frac{\omega_{Ln}}{\omega_{Cn}} = \frac{1}{\sqrt{2}} \quad (5.3.4)$$

Considering this it can also be shown that the passband of any single filter covers exactly an octave.

$$\frac{\frac{\omega_{Hn}}{\omega_{Cn}}}{\frac{\omega_{Ln}}{\omega_{Cn}}} = \frac{\omega_{Hn}}{\omega_{Ln}} = \frac{\sqrt{2}}{\frac{1}{\sqrt{2}}} = 2 \iff \omega_{Hn} = 2 \cdot \omega_{Ln} \quad (5.3.5)$$

In an ideal world the frequency response of the equalizer should then look like figure 5.8

¹Meaning that all the bandpass filters have the same gain.

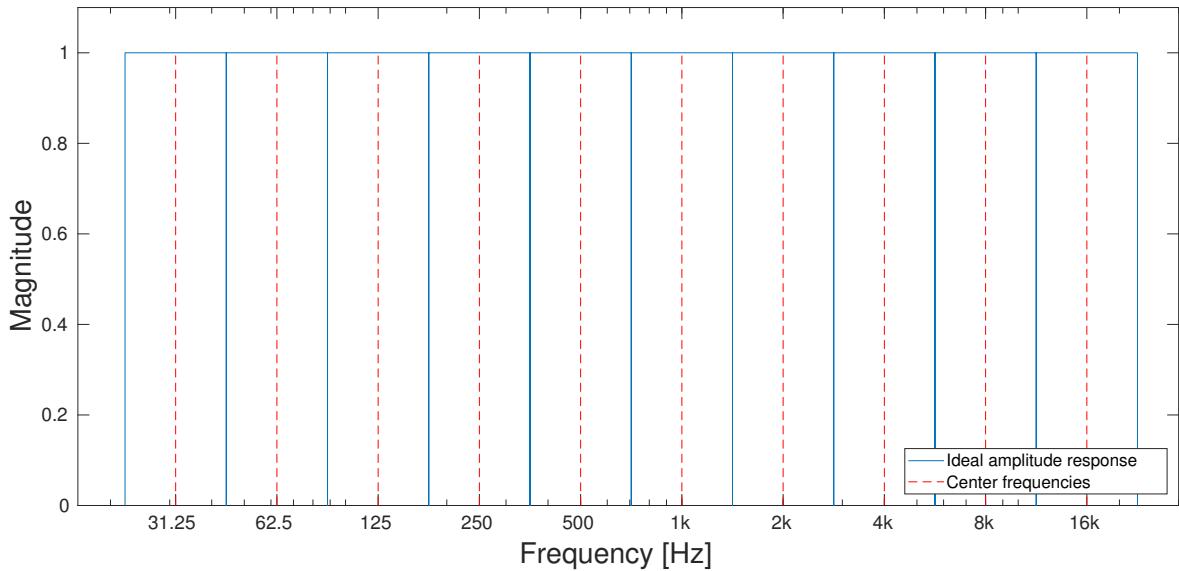


Figure 5.8: Ideal amplitude response of the equalizer.

Since the equalizer is supposed to be implemented on a Raspberry Pi, the bandpass filter will have to be digital filters. There are generally two ways of designing a digital filter as can be seen in section 2.3: FIR and IIR. The signal processing of the equalizer will have to be made in approximately real time on the processor of the stationary raspberry pi. Since IIR filters needs fewer computations, it makes more sense to implement the filters as IIR filters. Another reason to use IIR filters is that the filters should be able to handle very low frequencies compared to the sampling frequency of 44.1 kHz.

To get the desired frequency response given by the requirements in section 4.4, continuous time filters, that meets these requirements, will be designed initially. They will then afterwards be converted to the digital domain.

Since it is desired to have as flat a pass band as possible, it is chosen that the filters will be implemented as Butterworth filters. The order of the filters can be calculated using the following equation:

$$n \geq \frac{1}{2 \cdot \log\left(\frac{f_s}{f_p}\right)} \cdot \log\left(\frac{10^{\alpha_s/10} - 1}{10^{\alpha_p/10} - 1}\right) \quad (5.3.6)$$

Where:

n	= filter order	[·]
f_p	= pass frequency	[Hz]
f_s	= stop frequency	[Hz]
α_p	= pass attenuation	[dB]
α_s	= stop attenuation	[dB]

To calculate this the bandpass filter around 1 kHz will be used as a reference point. Using equation 5.3.3 the pass frequency can be calculated as $\sqrt{2}$ times the center frequency. In this case the pass frequency will be 1414 Hz. Because this is the edge of the filter the attenuation at the cutoff frequency would be 3 dB. Since a Butterworth filter is being used, it will require a large filter order to have no influence on the surrounding filters. Therefore is has been decided

that the attenuation at the next filters cutoff frequency should be at least 20 dB. As before the cutoff for the next filter will be $\sqrt{2}$ times the center frequency. This would be at 2828 Hz. With this the necessary order of the filter can be calculated.

$$n \geq \frac{1}{2 \cdot \log\left(\frac{2828\text{Hz}}{1414\text{Hz}}\right)} \cdot \log\left(\frac{10^{20\text{dB}/10} - 1}{10^{3\text{dB}/10} - 1}\right) = 3.32 \quad (5.3.7)$$

Since it is not possible to have a decimal number as the filter order, the value for n will be rounded up and $n = 4$. Given that a bandpass filter is needed the order would be twice as high, as this would be a combination of a 4th order high and lowpass filter. As a starting point, bandpass filters with the same center frequencies and bandwidth as earlier, will be implemented as 8th order bandpass filters.

To see if these filters can fulfill the requirements, all ten bandpass filters along with the sum of all of them are plotted in a bode plot in MATLAB.

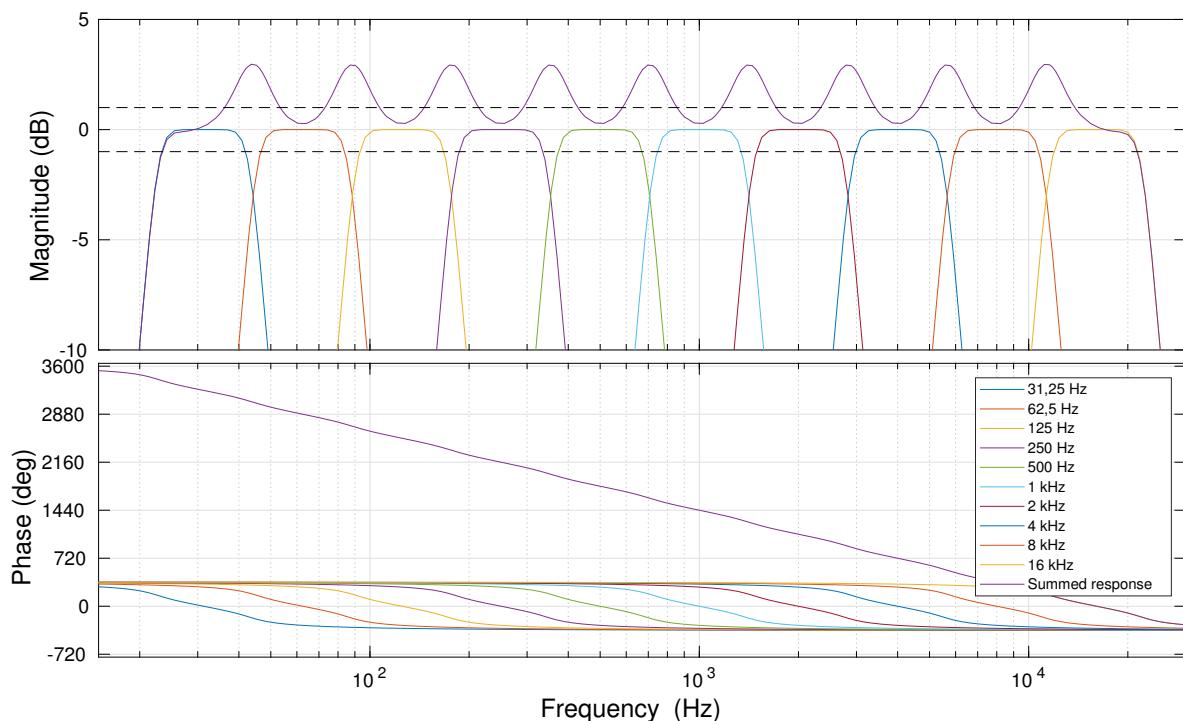


Figure 5.9: Bode plot of bandpass filters as 8th order Butterworth

As can be seen in figure 5.9, that these bandpass filters cannot fulfill the requirement of a bandpass ripple of ± 1 dB. It can be noticed that the gain where the bands meet ($\omega_{Hn} = \omega_{Ln+1}$), are significantly higher than 1 dB. This is expected since two gains of -3 dB are added together at the point there the bands meet. This should give a gain at these points of about $\frac{2}{\sqrt{2}} = \sqrt{2} = 2$ which is $20 \cdot \log 2 = 3\text{dB}^2$. Since the filters will be Butterworth filters, the edge frequencies will always have a gain of -3 dB. Therefore it is impossible to have a gain of less than 1 dB if the bands meet at the edge frequencies.

But as a consolation the phase looks similar to a straight line, which corresponds to constant group delay, which is also desired. This makes sense as the bands are an octave apart. Since all

²This is assuming that the other bands have an insignificant effect at this frequency.

the bands are the same just shifted an octave along, the phase responses will behave the same way. The results is that when the phase response of one the filters starts to flatten out, one of the other filters phase will start to decrease. This makes the sum of all the phases look similar to a straight line along the passband.

To lower the additional gain, the bandwidth of each band will be reduced to lower the gain of the filters at the point where the bandpass filters meet. It is desired to both keep the center frequencies from section 5.3.1 and have each band be the same size. The bands are shrunken by changing the ratio between the center frequencies ω_{Cn} and the - 3 dB frequencies ω_{Ln} and ω_{H-n} defined as $r = \frac{\omega_{Ln}}{\omega_{Cn}} = \frac{\omega_{Cn}}{\omega_{Hn}}$, which was initially set to $r = \frac{1}{\sqrt{2}} \approx 0.7071$. As this ratio r approaches 1, the -3 dB frequencies will get closer to the center frequencies, which will make the bands more narrow. To find a value of r that will satisfy the requirements, the magnitude response of the summed filter responses has been plotted in MATLAB. These can be seen in figure 5.10.

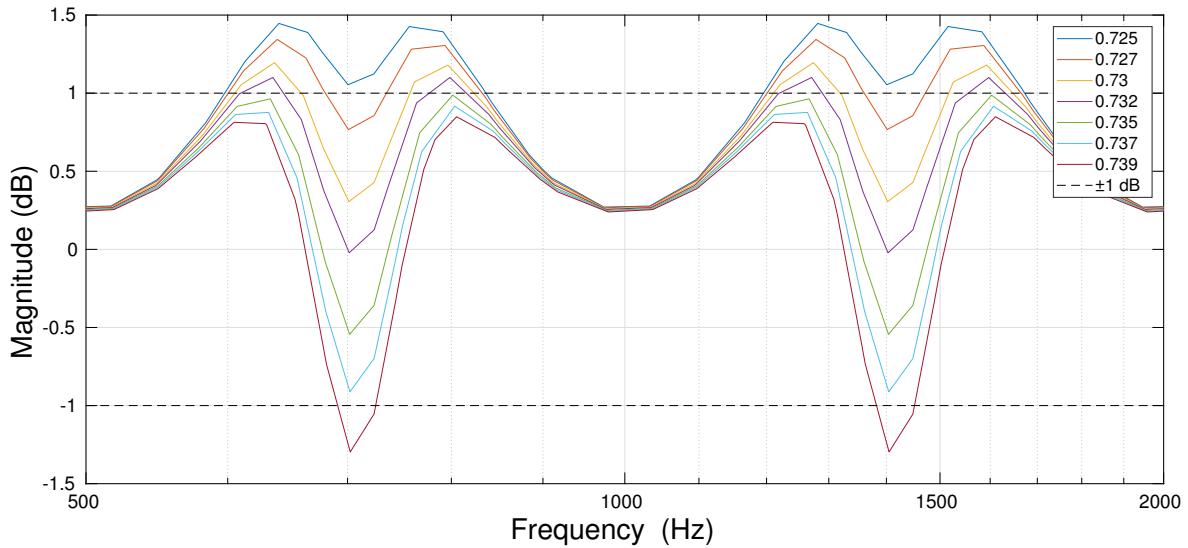


Figure 5.10: Summed filter responses with different values of r , around the 1 kHz band.

By graphical estimation it can be seen that the value of $r = 0.73$ results in the least amount of fluctuations. It can be seen that the graph with $r = 0.73$ goes from about 0.2 dB to 1.2 dB. To make the midpoint be at 0 dB the output needs to be attenuated with a factor of $\frac{(1.2-0.2)\text{dB}}{2} = -0.5\text{dB} = 0.94$. New 3 dB frequencies can now be calculated as

$$f_{Ln} = 0.73 \cdot f_{Cn} \quad (5.3.8)$$

and

$$f_{Hn} = \frac{f_{Cn}}{0.73} \quad (5.3.9)$$

These values are computed for each center frequency and are listed in table 5.2

Since the edge frequencies of adjacent bandpass filters are still pretty close to each other, this frequency shifting is assumed to not have a big impact on the phase response of the equalizer compared to the one seen in figure 5.9.

Band	f_C [Hz]	f_L [Hz]	f_H [Hz]
1	31.25	22.813	42.808
2	62.5	45.625	85.616
3	125	91.250	171.23
4	250	182.50	342.47
5	500	365.00	684.93
6	1 k	730.00	1.370 k
7	2 k	1.460 k	2.740 k
8	4 k	2.900 k	5.480 k
9	8 k	5.840 k	10.959 k
10	16 k	11.680 k	21.918 k

Table 5.2: The center and edge frequencies of the bandpass filters

It is now possible to determine the transfer functions of each of the bandpass filters.

A way to determine the transfer function of an arbitrary 8th order bandpass filter of a specific filter type, is to initially look at a normalized 4th order lowpass filter of the same type, and then perform a frequency transformation.

The coefficients of a 4th order Butterworth lowpass filter can be looked up in a table. The transfer function can then be written as:

$$H_{LPP}(s) = \frac{1}{s^4 + 2.613s^3 + 3.414s^2 + 2.613s + 1} \quad (5.3.10)$$

where $s = j\omega$ [Rad/s]

To transform this into a bandpass filter, the pass band is shifted along the frequency axis, to be centered around a given center frequency. This shifts the s in the transfer function to $S + \omega_C$. To implement a zero at zero and poles on the each edge of the passband, s is further transformed to $\frac{S^2 + \omega_C^2}{S}$. Lastly the transfer function is frequency scaled to cover a given bandwidth $B = \omega_H - \omega_L$, which gives a transformation of:

$$s = \frac{S^2 + \omega_C^2}{S \cdot B} = \frac{S^2 + \omega_C^2}{S \cdot (\omega_H - \omega_L)} \quad (5.3.11)$$

The transfer functions for the 8th order bandpass filter can thus be found as:

$$\begin{aligned} H_{BP}(s) &= H_{LPP} \left(\frac{S^2 + \omega_C^2}{S \cdot B} \right) \\ &= \frac{1}{\left(\frac{S^2 + \omega_C^2}{S \cdot B} \right)^4 + 2.613 \left(\frac{S^2 + \omega_C^2}{S \cdot B} \right)^3 + 3.414 \left(\frac{S^2 + \omega_C^2}{S \cdot B} \right)^2 + 2.613 \left(\frac{S^2 + \omega_C^2}{S \cdot B} \right) + 1} \\ &= \frac{(S \cdot B)^4}{(S^2 + \omega_C^2)^4 + 2.613(S^2 + \omega_C^2)^3 \cdot SB + 3.414(S^2 + \omega_C^2)^2 \cdot (SB)^2 + 2.613(S^2 + \omega_C^2) \cdot (SB)^3 + (SB)^4} \end{aligned} \quad (5.3.12)$$

It can be seen that the system has turned into an 8th order filter with 4 zeros at zero and 8 poles i.e. an 8th order bandpass filter. Thus the 10 bandpass filter transfer function can be found by inserting the values found in table 5.2 into equation 5.3.12.

As an example the transfer function on the filter centered around 4 kHz³ will be calculated.

For this filter it is known that:

$$B_8 = 2\pi \cdot (f_{H8} - f_{L8}) = 2\pi \cdot (5.48\text{kHz} - 2.90\text{kHz}) = 16.21 \cdot 10^3 \text{rad/s}$$

$$\omega_{C8} = 2\pi \cdot f_{C8} = 2\pi \cdot 4\text{kHz} = 25.13 \cdot 10^3 \text{rad/s}$$

By inserting these values into equation 5.3.12 we get the following transfer function:

$$H_8(s) = \frac{6.69E16s^4}{s^8 + 4.20E4s^7 + 3.41E9s^6 + 9.05E13s^5 + 3.58E18s^4 + 5.72E22s^3 + 1.36E27s^2 + 1.06E31s + 1.59E35} \quad (5.3.13)$$

Solving the roots of the polynomial in the denominator gives you that poles of the filter.

$$p_8 = \{-3.954 \pm 33.465i, -8.375 \pm 27.313i, -6.482 \pm 21.139i, -2.199 \pm 18.615i\} \cdot 10^3 \text{rad/s}$$

5.3.3 Transformation to the digital domain

Since the equalizer is going to be implemented on a raspberry pi, it is necessary that the filters handles time discreet signals. Accordingly the transfer functions will be transformed to the z-domain. There are a couple of different ways to obtain a z-domain transfer function from s-domain. For this specific task the bilinear transformation is chosen since this method largely preserves the shape of the original Butterworth filters. The bilinear transform is derived in appendix C, but in short the bilinear transform simply maps the s-plane to the z-plane. This is done by replacing s with:

$$s = \frac{2}{T_d} \frac{z - 1}{z + 1} \quad (5.3.14)$$

As it can be seen from equation C.0.11 it is necessary to pre-warp the s-domain transfer function to get the desired z-domain transfer function using the formula:

$$\omega_a = \frac{2}{T_d} \tan\left(\frac{\omega_d T_d}{2}\right) \quad [\text{rad/s}] \quad (5.3.15)$$

As the audio interface of the system is sampling and playing back at a sampling rate of 44.1 kHz, the discretization time T_d is set to $(44.1\text{kHz})^{-1} = 22.676\mu\text{s}$. To preserve the shape of the bandpass filters it is important that both the higher and lower cutoff frequencies are preserved. This is accomplished by pre-warping these frequencies, and redoing the lowpass to bandpass transform form equation 5.3.12 with these new values. Once again the filter centered around 4 kHz is used as an example.

$$\begin{aligned} \omega_{L,a} &= \frac{2}{T_d} \tan\left(\frac{\omega_{L,d} T_d}{2}\right) = \frac{2}{22.676\mu\text{s}} \tan\left(\frac{2\pi \cdot 2.92\text{kHz} \cdot 22.676\mu\text{s}}{2}\right) \\ &= 18.616 \cdot 10^3 \text{rad/s} \end{aligned} \quad (5.3.16)$$

$$\begin{aligned} \omega_{H,a} &= \frac{2}{T_d} \tan\left(\frac{\omega_{H,d} T_d}{2}\right) = \frac{2}{22.676\mu\text{s}} \tan\left(\frac{2\pi \cdot 5.480\text{kHz} \cdot 22.676\mu\text{s}}{2}\right) \\ &= 36.291 \cdot 10^3 \text{rad/s} \end{aligned} \quad (5.3.17)$$

³which is the 8th filter counting from lowest to highest frequency

These values can be used to compute a new center frequency and bandwidth.

$$B_a = \omega_{H,a} - \omega_{L,a} = 36.291 \cdot 10^3 \text{ rad/s} - 18.616 \cdot 10^3 \text{ rad/s} = 17.674 \cdot 10^3 \text{ rad/s} \quad (5.3.18)$$

$$\omega_{C,a} = \sqrt{\omega_{H,a} \cdot \omega_{L,a}} = \sqrt{36.291 \cdot 10^3 \text{ rad/s} \cdot 18.616 \cdot 10^3 \text{ rad/s}} = 25.992 \cdot 10^3 \text{ rad/s}$$

By once again inserting these values into equation 5.3.12, a pre-warped analog transfer function has been found in equation 5.3.19.

$$H_{a,8}(s) = \frac{9.76E16 s^4}{s^8 + 4.62E4 s^7 + 3.76E9 s^6 + 1.08E14 s^5 + 4.28E18 s^4 + 9.30E22 s^3 + 1.72E27 s^2 + 1.42E31 s + 2.08E35} \quad (5.3.19)$$

This transfer function is then converted to the z-domain using the bilinear transform.

$$H_8(z) = H_{a,8} \left(\frac{2}{T_d} \frac{z-1}{z+1} \right) = H_{a,8} \left(\frac{2}{22.676\mu\text{s}} \frac{z-1}{z+1} \right) \\ = \frac{0.0007154 z^8 - 0.002862 z^6 + 0.004293 z^4 - 0.002862 z^2 + 0.0007154}{z^8 - 5.923 z^7 + 16.27 z^6 - 26.86 z^5 + 29.08 z^4 - 21.12 z^3 + 10.06 z^2 - 2.882 z + 0.3832} \quad (5.3.20)$$

It can be seen from figure 5.11 that this new filter transfer function $H_8(z)$ largely retains the frequency response of the original analog filter $H_8(s)$ from equation 5.3.13.

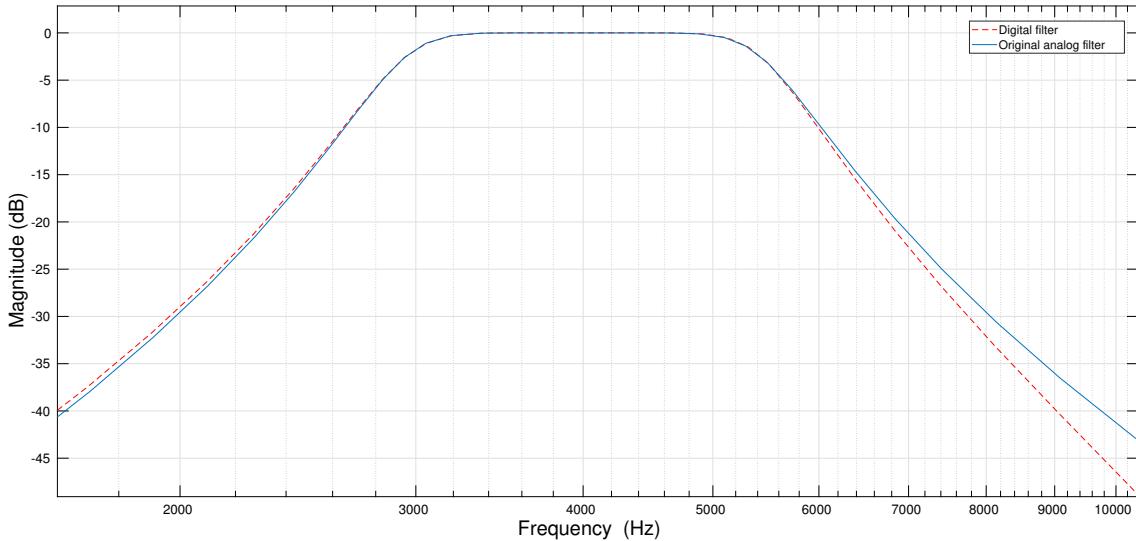


Figure 5.11: The 4 kHz band transfer function in s- and z-domain.

Though it can be seen that the digital filter attenuates lower frequencies less than the original analog filter, this is assumed to not have a big effect since deviation only looks significant at frequencies where the magnitude is already under -20 dB. At these frequencies there will be other bands with a gain of approximately 0 dB, so the deviations are assumed to have a negligible effect on the overall response of the equalizer.

It is possible to directly convert the transfer function from equation 5.3.20 but since the transfer function is quite unwieldy it has been decided to implement it as four 2nd order filters in cascade. Another reason for this is to ensure that poles of a filter are not close enough together to give error in the coefficients due to quantization [cite signal procesisng with ove](#). This can be ensured by only letting a single filter have a single complex conjugated pair of poles.

These four filters are each constructed from two poles and two zeros from complete transfer function $H_8(z)$. The poles and zeros are found by solving the roots of the denominator and numerator in $H_8(z)$ respectively. The poles and zeros can be seen in figure 5.12

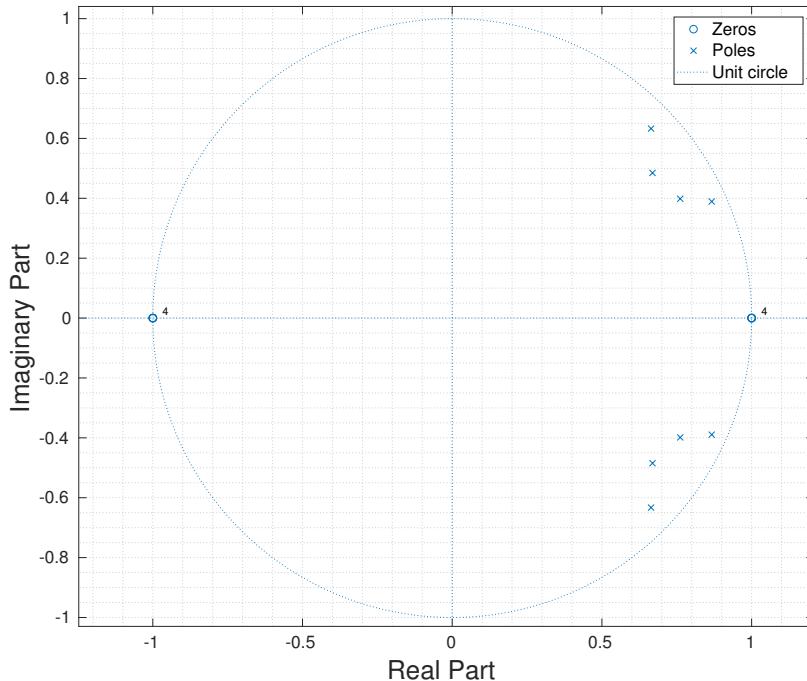


Figure 5.12: The poles and zeroes of the digital filter $H_8(z)$.

There are four zeros in 1 (0 Hz) and four zeros in -1 (half the sampling frequency $f_s/2 = 22.05$ kHz). Once again there are four pairs of complex conjugated pole pairs can be seen in the following equation.

$$\text{Poles} = \{0.6642 \pm 0.6329j, 0.8667 \pm 0.3893j, 0.6689 \pm 0.4848j, 0.7617 \pm 0.3985j\} \quad (5.3.21)$$

All the 2nd order filters will each have a complex conjugated set of poles and two zeros. By constructing a 2nd order filter from the zeros at -1, the filter will be a lowpass filter, and by using the zeros at 1 the filter will be a highpass filter.

A common way to choose which poles should go with which zeros is to start with the pole closest to the unit circle and pair it with its closest zero. This ensures that the coefficients of the transfer functions are around the same size, and therefore that the filters won't have extreme gain or attenuation. In this case the poles closest to the unit circle are the poles at $0.6642 \pm 0.6329j$.⁴ These two poles will then be paired with two zeros in 1.

⁴This is determined by looking at the absolute value of the poles.

As the 8th order filter is timed by a constant of $7.154 \cdot 10^{-4}$ every 2nd order filter will be made with constant factor:

$$c = (7.154 \cdot 10^{-4})^{1/4} = 0.1635 \quad (5.3.22)$$

So the first filter of the cascade structure can now be determined as:

$$H_{8,1}(z) = c \cdot \frac{(z-1)(z-1)}{(z-p_1)(z-p_1^*)} = \frac{c - 2cz^{-1} + cz^{-2}}{1 - 1.328z^{-1} + 0.8418z^{-2}} \quad (5.3.23)$$

The same is repeated for the next closest pole in $, 0.8667 \pm 0.3893j$

$$H_{8,2}(z) = c \cdot \frac{(z-1)(z-1)}{(z-p_2)(z-p_2^*)} = \frac{c - 2cz^{-1} + cz^{-2}}{1 - 1.733z^{-1} + 0.9027z^{-2}} \quad (5.3.24)$$

The next two filters will then be constructed with the remaining poles and the four zeros in -1 . These filters will then be lowpass filters.

$$H_{8,3}(z) = c \cdot \frac{(z+1)(z+1)}{(z-p_3)(z-p_3^*)} = \frac{c + 2cz^{-1} + cz^{-2}}{1 - 1.338z^{-1} + 0.6825z^{-2}}$$

$$H_{8,4}(z) = c \cdot \frac{(z+1)(z+1)}{(z-p_4)(z-p_4^*)} = \frac{c + 2cz^{-1} + cz^{-2}}{1 - 1.523z^{-1} + 0.739z^{-2}}$$

The amplitude responses of these four filter can b seen in figure 5.13.

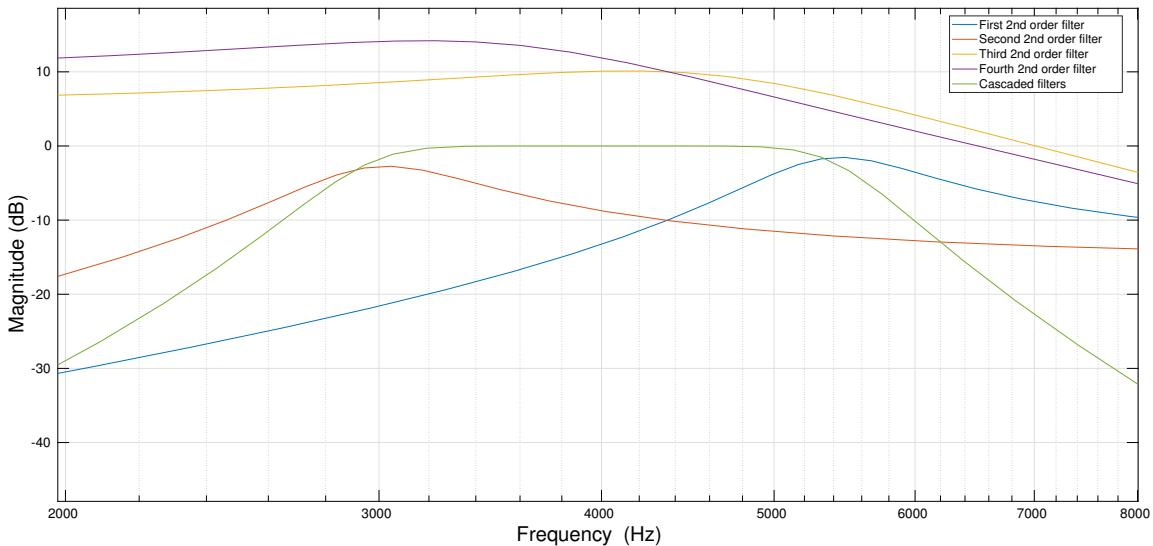


Figure 5.13: cascaded filters

Despite the fact that neither of the filter looks like a Butterworth filter on their own, the result of the cascaded filters looks like the wanted filter. These transfer function can then be transformed into a difference equation by doing a inverse z-transform, like it is shown in equation 5.3.25.

$$H_{8,1}(z) = \frac{Y_{8,1}(z)}{X_{8,1}(z)} = \frac{0.1635z^2 - 0.3271z + 0.1635}{1 - 1.328z^{-1} + 0.8418z^{-2}} \quad (5.3.25)$$

$$\Leftrightarrow Y_{8,1}(z) \cdot (1 - 1.328z^{-1} + 0.8418z^{-2}) = X_{8,1}(z) \cdot (0.2989 + 0.5977z^{-1} + 0.2989z^{-2})$$

$$\Leftrightarrow y_{8,1}[n] - 1.328y_{8,1}[n-1] + 0.8418y_{8,1}[n-2] = 0.2989x_{8,1}[n] - 0.5977x_{8,1}[n-1] + 0.2989x_{8,1}[n-2]$$

The same can be done for all the other 2nd order filters.

Not every bandpass filter will behave like they should when implemented in this way. This is illustrated in figure 5.14, where it can be seen that the 16 kHz band has a weird shape compared to what the Butterworth bandpass filter is supposed to look like, and every band below the 250 Hz band has neither the correct central frequency, bandwidth or gain.

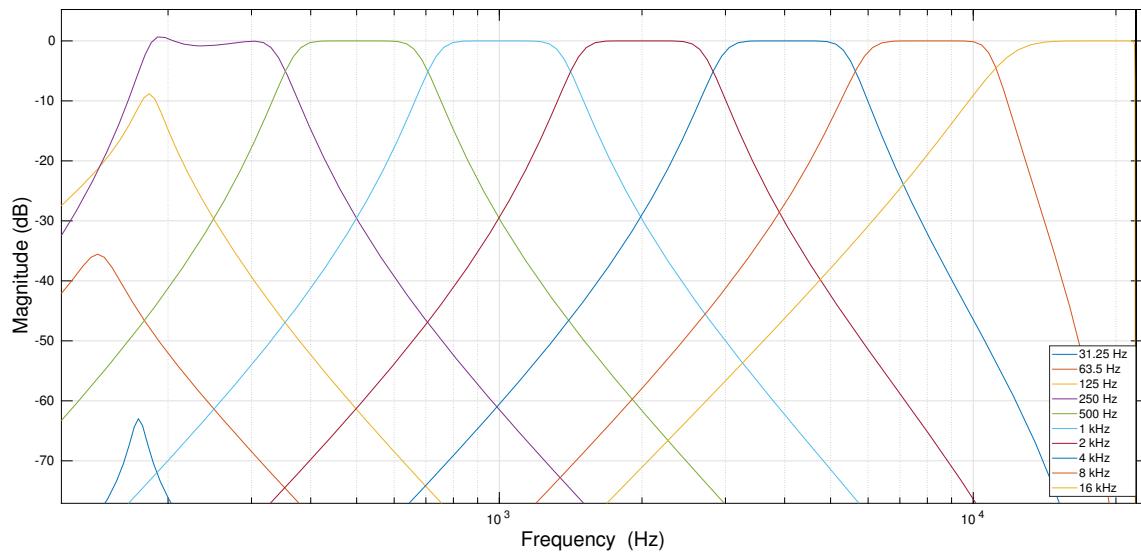


Figure 5.14: Amplitude responses of bandpass filters obtained with the bilinear transform.

The reason for this distortion of the amplitude response can be seen by looking at the poles of the last four bands which can be seen in figure 5.15

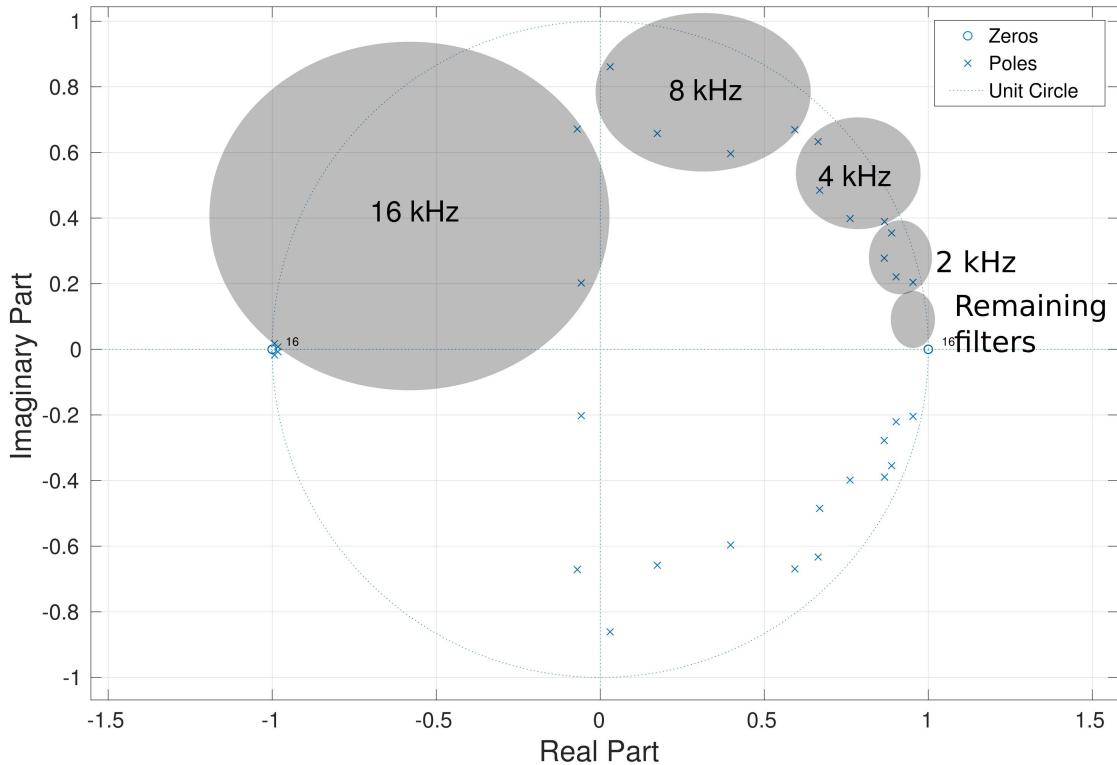


Figure 5.15: Poles and zeros of the 2 kHz to 16 kHz bands.

In figure 5.15 the 16 kHz band has to poles almost at half the sampling frequency (at -1). This causes the shape of the right side of the passband of the 16 kHz filter. As the center frequency of the bands gets closer to 0, the poles gets both more squished together and closer to the unit circle at 1. Both of these properties leads to instability of the filters. Thus another way of implementing the filters is necessary.

5.3.4 Down- and upsampling

From appendix C it shown that the z in the transfer functions corresponds to $e^{j\omega T_d}$. From this relation it can be noted that by scaling the sampling time T_d of the input of a transfer function $H(z)$ by a factor, the frequencies will be scaled by the inverse factor. This means that by multiplying the sampling time by 2, the frequency response of $H(z)$ will be shifted an octave down.

This technique is known as downsampling.

Since all the bandpass filters are an octave apart, you could for example sample the input of the 8 kHz filter found in section 5.3.3 at half the sampling rate of the input, and make the filter behave similarly to the wanted 4 kHz filter. As the z-domain transfer function will remain the same, the poles and zeros are unchanged as well. This means that this new 4 kHz filter will have the exact same pole-zero plot as shown in figure 5.12 expect that the point at -1 corresponding to half the original sampling frequency. The sampling frequency of 22.05 kHz will now correspond to half of this, namely 11.025 kHz.

This can eliminate the problems related to poles being to close to each other and to the unit

circle. Another advantage is that a filter made from downsampling a filter an octave above will need only half as many computations since only every other sample is read. Therefore all the lower bands of the equalizer could essentially be made by taking a stable bandpass filter and downsampling it by a factor of 2^n where n is an integer.

It is of course always useful to lessen the number of computations the system has to do, to filter one input buffer.

Since the computations of the filtering is approximately halved when a filter is downsampled to an octave below, quartered when downsampled two octaves et.c, it is most efficient to have every filter downsampled as much as possible while still ensuring that the filters have the desired shape. The highest frequency filter that looks closest to the original butterworth shape is the 4 kHz band. For this reason it has been decided to implement every filter below the 4 kHz band, as a downsampled version of the 4 kHz filter exemplified in section 5.3.3. The amplitude responses of these filters can be seen in figure 5.16.

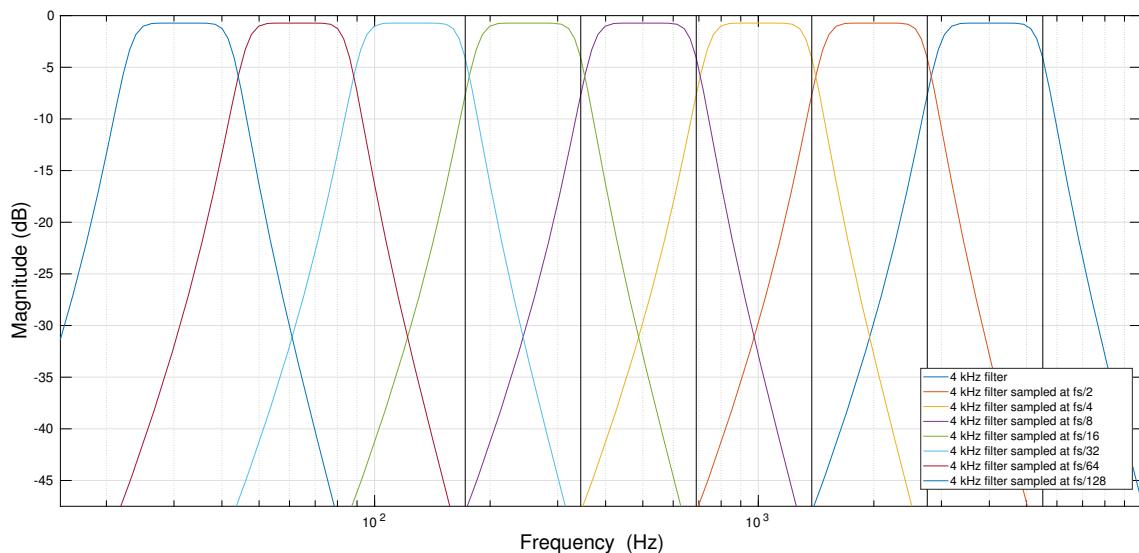


Figure 5.16: The first 8 filters implemented using the same transfer function [Bigger](#)

The gain at the point where neighboring filters meet is still about 6 dB. So there is no reason to assume that this configuration of the filters won't be able to live up to the requirements about the passband gain.

One issue with changing halving the sampling frequency is that the Nyquist frequency is halved as well. That means that the frequencies over 11.05 kHz will cause aliasing in the 2 kHz filter, 5.525 kHz will cause aliasing in the 1 kHz filter and so on. This means that the input to the downsampled filters will have to go through and anti-aliasing lowpass filter before being downsampled to ensure that the Nyquist requirement is met. This will simply be implemented as a second order Butterworth⁵ filter.

As stated before the 2 kHz filter should attenuate frequencies over 11.05 kHz. The anti-aliasing filter is consequently designed as a 4th order Lowpass filter with a cutoff frequency of 11.05 kHz.

⁵Butterworth filters are once again chosen since a flat passband response is desired.

It is assumed that a 4th order filter has a steep enough cutoff that the unwanted frequency won't cause significant aliasing.

The analog transfer function is found by replacing s with $\frac{s}{\omega_c}$ in equation 5.3.10.

$$\begin{aligned} H_{aa}(s) &= H_{LPP}\left(\frac{s}{2\pi 11.05}\right) = \frac{1}{\left(\frac{s}{2\pi 11.05}\right)^4 + 2.613\left(\frac{s}{2\pi 11.05}\right)^3 + 3.414\left(\frac{s}{2\pi 11.05}\right)^2 + 2.613\left(\frac{s}{2\pi 11.05}\right) + 1} \\ &= \frac{2.324 \cdot 10^7}{s^4 + 181.4s^3 + 1.646 \cdot 10^4 s^2 + 8.746 \cdot 10^5 s + 2.324 \cdot 10^7} \end{aligned} \quad (5.3.26)$$

The same way as in section 5.3.3 the transfer function is transformed into two 2nd order difference equations. This will be skipped since the procedure is completely the same.

The resulting difference equation by doing the bilinear transform are:

$$y_{aa1}[n] = 0.3077x[n] + 0.6154x[n-1] + 0.3077x[n-2] - 0.005152y_{aa1}[n-1] - 0.4465y_{aa1}[n-2] \quad (5.3.27)$$

$$y_{aa2}[n] = 0.3077x[n] + 0.6154x[n-1] + 0.3077x[n-2] - 0.003703y_{aa2}[n-1] - 0.03957y_{aa2}[n-2] \quad (5.3.28)$$

The next anti-aliasing should have a cutoff frequency an octave below. Since the output of the first anti-aliasing filter is already band limited to 11.05 kHz the next anti-aliasing filter could be made by downsampling the same filter by 2. This makes sense since the signal already need to be downsampled before going into the bandpass filter. This way all the anti-aliasing and downsampling can be made in one go with a cascade structure as shown in figure 5.17.

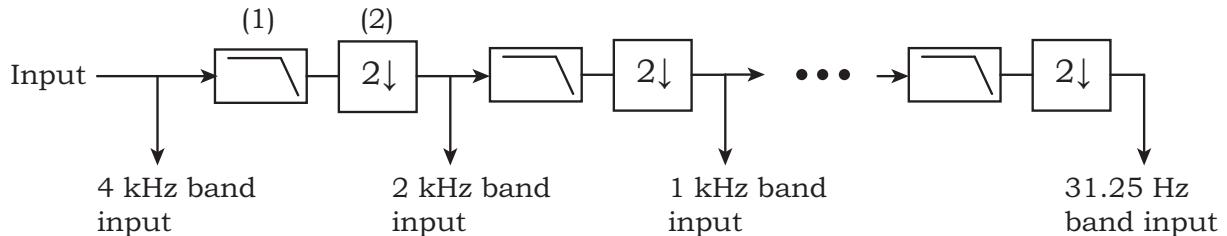


Figure 5.17: Downsampling of the input using cascaded lowpass filters with a cutoff of half the sampling frequency (1) and downsampling by a factor of 2 (2).

The input to the 31.25 kHz band will be downsampled by a factor of 2 seven times which is equivalent to downampling by a factor of $2^7 = 128$. It may seem like a waste to let the input go into up to 7 lowpass filters, but since the filter calculations are approximately halved every time the signal is downsampled by 2, the illustrated structure will need to do fewer calculations compared to having 7 parallel anti-aliasing filters with downampling afterwards.

When the input has been sent through the different bandpass filters, they will need to be added together once again. This means that the signals needs to be upsampled to 44,1 kHz again.

Upsampling is usually done by first zero-filling, and then lowpass filtering the signal with a cutoff frequency of half the sampling rate and a gain equal to the upsampling rate. This process is called interpolation. **Why this works might come in an appendix.**

Interpolating by a factor of $m \cdot k$ is the same as interpolating by m and then afterwards interpolating by k . Since all output signals need to be interpolated by some factor of 2^n , the interpolation and summation of the outputs can be done in a similar cascaded structure to how the downsampling is done. This is illustrated in figure 5.18.

Since the lowpass filters all just needs to be lowpass filters with a cutoff frequency of half the sampling frequency, the anti-aliasing filters described by the transfer function in equation 5.3.26 can be re-purposed as interpolation filters.

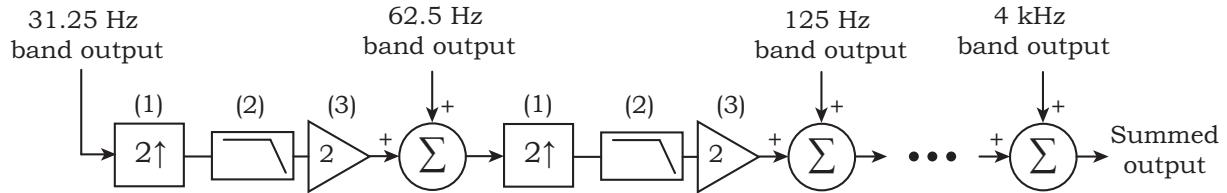


Figure 5.18: Interpolation and summation of the bandpass filtered output signals, using Upsampling (1), lowpass filtering (2) and a gain of 2 (3).

Two downsampled bands next to each other will always be downsampled by a factor of n and $2n$. By initially just upsampling the signal with the lowest frequency by a factor of 2, it is then possible to immediately add the resulting signal to output of the band an octave higher since they will now have the same sampling frequency. This step is repeated until all the outputs have been summed together and upsampled to 44.1 kHz.

High Frequency bandpass filters

The bandpass filters centered around 8 kHz and 16 kHz will not be made by frequency shifting the 4 kHz filter, since this in this case would increase the filter calculations if they were to be made by upsampling to twice and four times the sampling rate. They will rather just be made with two additional transfer functions.

The 8 kHz bandpass filter will be made the same way as the 4 kHz filter by simply determining the s -domain transfer function and doing the bilinear transform to convert it to the z -domain. Thereafter the z -domain transfer function is converted to four difference equations. This is again skipped over since the process has already been exemplified in section 5.3.3.

For the 16 kHz filter it is a lot easier to implement it as a highpass filter since there is already some lowpass filtering on the ADC of the sound card to prevent aliasing of frequencies over 22 kHz. So the filter will be implemented as a Butterworth highpass filter with a cutoff frequency at the lower edge frequency of the band $f_{L10} = 11.68$ kHz. Since the digital 8 kHz band has a steeper cutoff curve than expected because of the bilinear transform, as seen in figure 5.14, the 16 kHz filter is adjusted slightly to achieve a flat passband ripple between the bands. This is done by changing the cutoff frequency of the highpass filter to 12 kHz and setting the filter order to 6. Once again the filter is implemented digitally using the same method as described in section 5.3.3.

This results in the highpass filter to have approximately as steep a cutoff curve as the 8 kHz band, and also that the two curves meet at -6 dB. All this can be illustrated in figure 5.19

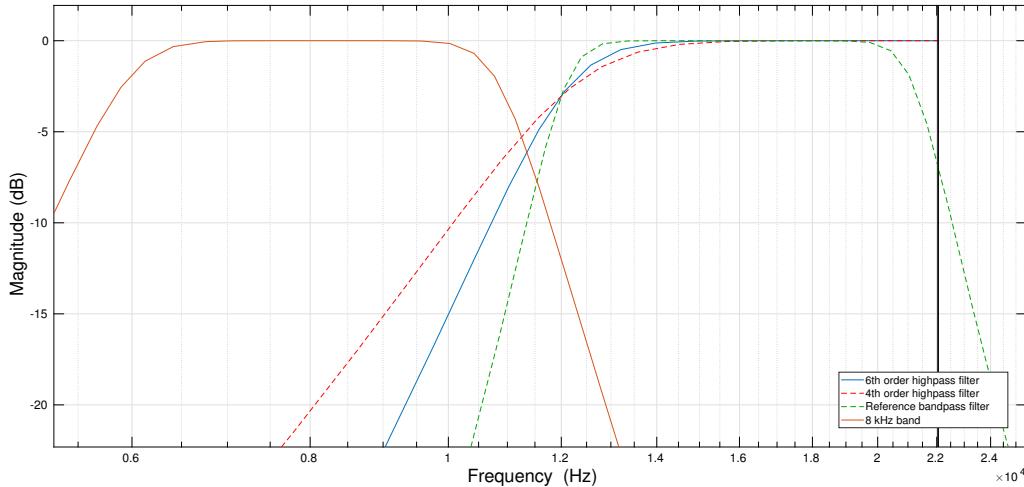


Figure 5.19: Different implementation of the 16 kHz band compared to the 8 kHz band [Bigger](#)

The resulting transfer function can be found to be:

$$H_{10}(z) = \frac{0.01912 - 0.1147z^{-1} + 0.2869z^{-2} - 0.3825z^{-3} + 0.2869z^{-4} - 0.1147z^{-5} + 0.01912z^{-6}}{1 + 0.5248z^{-1} + 0.8805z^{-2} + 0.2576z^{-3} + 0.1413z^{-4} + 0.01777z^{-5} + 0.002313z^{-6}} \quad (5.3.29)$$

Which can be transformed to three difference equations.

$$y_1[n] = 0.2674x[n] - 0.5348x[n-1] + 0.2674x[n-2] - 0.2204y_2[n-1] + 0.5919y_2[n-2] \quad (5.3.30)$$

$$y_2[n] = 0.2674x[n] - 0.5348x[n-1] + 0.2674x[n-2] - 0.1629y_2[n-1] - 0.1763y_2[n-2] \quad (5.3.31)$$

$$y_3[n] = 0.2674x[n] - 0.5348x[n-1] + 0.2674x[n-2] - 0.1415y_3[n-1] - 0.02217y_3[n-2] \quad (5.3.32)$$

Programming of the filters

Now that the filters are calculated and transformed into discrete time domain, it is possible to program these in C. First the structure of the code is examined. Each filter is split into two second order highpass and two second order lowpass filters, that can then be programmed so the output of one filter becomes the input of the next. When down sampling aliasing also becomes a problem so anti aliasing filters have been designed as well. When the downsampled signals needs to be upsampled again to the original sampling frequency, the anti-aliasing filters are reused as interpolation filters. The program flow for the implementation of the first nine filters can be seen in figure 5.20. The filters at 8 kHz and 16 kHz is not included here because they have different coefficients. But they are essentially just other filters in parallel with the rest.

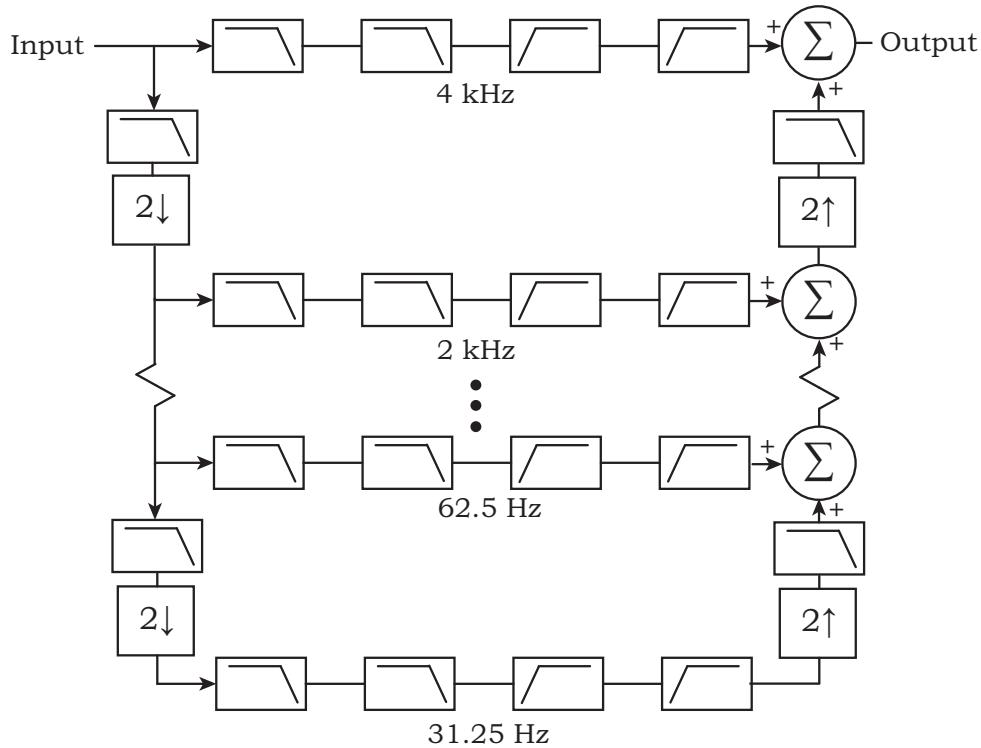


Figure 5.20: Sketch over the program flow for the filters

First off the input from the buffer is filtered using the 4 kHz bandpass. The output here is then saved in a buffer so it can be added together with the other filter outputs. Next the input is lowpass filtered using the previously described anti aliasing filter. Afterwards it is downsampled by a factor 2 and sent to the next filter. This process is then repeated until the input has been downsampled and filtered through the 31.25 Hz filter. The output from the lowest bandpass filter is then upsampled and low pass filtered, before it is added with the output of the next filter. This is again repeated until the original sampling frequency is obtained. First off the setup for the function that handles the downsampled bandpass filtering can be seen in code example 5.7.

Code example 5.7: Function for the dwonsampled bandpass filters

```

1 void bandpassFilter(int m, double ratio) {
2     double output;
3     int k = 0;
4     int dsRatio = pow(2,m);
5     ...
6 }
```

The downsampled bandpass function is getting two inputs, `m` and `ratio`. The input `m` is telling the function which filter is currently being processed. Here a zero would be the 4 kHz filter, a one would be 2 kHz and so on. The `ratio` input is used to scale the output so the controller can turn up or down for a specific bandpass filter. At the start of the function some variables are declared. The double `output` is used for the ratio calculations. The integer `k` is a variable used for controlling the loop that handles the filters. Next the downsampling ratio is calculated and saved in the integer `dsRatio`. This is done with the `pow` function that returns two to the power of `m`. If the input have been downsampled two times and there will only be a fourth of

the number of samples to process, so the number of samples is calculated as the size of a frame divided by the downsampling ratio.

Code example 5.8: Function for the downsampled bandpass filters

```

1 while(k != frameSize/dsRatio) {
2     if(m == 0) {
3         x[m][0] = filterInputBuffer[k];
4     } else {
5         x[m][0] = aaFilterOut[k];
6     }
7     // The filter calculations
8     y[m][0][0] = 0.1635*(x[m][0] - 2*x[m][1] + x[m][2]) + 1.328*y[m][0][1] - 0.8418*y[m][0][2];
9     y[m][1][0] = 0.1635*(y[m][0][0] - 2*y[m][0][1] + y[m][0][2]) + 1.733*y[m][1][1] - 0.9027*y[m][1][2];
10    y[m][2][0] = 0.1635*(y[m][1][0] + 2*y[m][1][1] + y[m][1][2]) + 1.338*y[m][2][1] - 0.6825*y[m][2][2];
11    y[m][3][0] = 0.1635*(y[m][2][0] + 2*y[m][2][1] + y[m][2][2]) + 1.523*y[m][3][1] - 0.739*y[m][3][2];
12    output = ratio * y[m][3][2] * pow(-1,m);

```

short explanation of what `y[x][y][z]` is. datatype, array lenght etc

The next part of the function is a while loop that repeats the filter calculations until the available samples have been processed. This can be seen in code example 5.8. The variables `x` and `y` are stored as global arrays of doubles so they can be saved between periods. The structure of the variables are a series of arrays where the first signifies the bandpass number from 0 to 7, the next is equivalent to the filter number from 0 to 3 and the last is for the previous values from 0 to 2. In the start of the while loop the input for the filter is updated. If it is the 4 kHz filter the input will come directly from the input buffer else it will be the downsampled output from the anti aliasing filters. Now that the parameters for the filter and the inputs have been clarified, it is possible to do the actual filter calculations. Line eight from the code example is equivalent to the first highpass filter. The output of the first filter depends on the input, the previous inputs and the previous outputs of the filter. For all the filters there are some coefficients that the previous outputs needs to be multiplied with. The output from the first filter is then used as the input for the next filter. This procedure is repeated until all the filters have been processed. After the filter calculations are done there is the variable that the output needs to be multiplied with, depending on the controller calculations. It has been determined that the anti-aliasing and interpolation filters causes bands next to each other to be reverse phase. Therefore every other bandpass filter is multiplied by -1 using the `pow()` function.

Code example 5.9: Function for the downsampled bandpass filters

```

1     // Updating the variables
2     x[m][2] = x[m][1]; x[m][1] = x[m][0];
3     y[m][0][2] = y[m][0][1]; y[m][0][1] = y[m][0][0];
4     y[m][1][2] = y[m][1][1]; y[m][1][1] = y[m][1][0];
5     y[m][2][2] = y[m][2][1]; y[m][2][1] = y[m][2][0];
6     y[m][3][2] = y[m][3][1]; y[m][3][1] = y[m][3][0];
7     // Saving the data in an array for the output
8     outputArray[m][k] = output;
9     k++;
10    }
11    return;
12 }

```

The last part of the function can be seen in code example 5.9. Since the filter need the previous

inputs and outputs in order to do the calculations, the variables that are needed are updated one at a time. After the variables is updated the output for the rest of the system need to be updated as well. Last in the function the counter is updated for the next run trough of the while loop. **With these lines of code the function for filtering and add justing a specific bandpass filter. <- grammar** The same principles are used for the 8 kHz and 16 kHz filters, but here there is no downsampling used.

For downsampling and anti aliasing a separate function have been made. This can be found in code example D.1 in the appendix. The code concerning the anti aliasing filter have the same structure as the code for the bandpass filters, so this will not be explained further. The downsampling can be found in code example 5.10.

Code example 5.10: Downsampling data

```

1 if(i%2 == 0) {
2     aaArray[i/2] = aaOut[n][0];
3 }
```

The downsampling is done by only saving every other sample in the output array for the anti aliasing filter. To save on memory the same array is being used for the output of the anti aliasing filter, as it is only needed one time in the bandpass filters. Afterwards the program moves to the next filter and need a new set of downsampled input values.

When the data stream have been downsampled and filtered, both using the anti aliasing filter and the bandpass filter, it have to be upsampled again. This is done using zero filling and lowpass filtering. The full function of the interpolation filter can be seen in code example D.2. The zero filling is done by only taking a sample for the filter input every other run through. The rest will be set to zero. This can be seen in code example 5.11. When the output is saved it is multiplied by a factor equal to the upsampling factor. In this case it is 2.

Code example 5.11: Upsampling data

```

1 if(i%2 == 0) { //Input
2     if(n == 7) {
3         iIn[n][0] = filterOut[n][i/2];
4     } else {
5         iIn[n][0] = intFilterOut[n+1][i/2] + filterOut[n][i/2];
6     }
7 } else {
8     iIn[n][0] = 0;
9 }
10 intFilterOut[n][i] = 2 * iOut2[n][0]; //Output
```

Because the data is being zero filled for every other sample the sampling frequency is increased by a factor of two. Since the downsampling is done with the a factor of 2 every time, the same principle can be used here. This means that the first data to be upsampled is the data from the filter with a center frequency of 31.25 Hz, since this is the most downsampled. When the data have been through the filter it has the same sampling frequency as the 62.5 Hz band output. These can then be added together and the same filter is used again, just with the double sampling frequency. This process in repeated until all the downsampled filter data have been upsampled to the original sampling frequency.

All the above functions have to be called every time a period of samples needs to be equalized. The input buffer needs to be sent through both the 16 kHz filter, the 8 kHz filter and the downsampling structure shown in figure 5.20. The function for doing all this filtering is shown in code example 5.12.

Code example 5.12: Function for doing all filters

```

1 void doAllFilters(double ratio[10]) {
2     double tmpOut;
3     filter16k(ratio[9]);
4     filter8k(ratio[8]);
5     for(int i = 0; i < 8; i++) {
6         bandpassFilter(i, ratio[7-i]);
7         antiAliasing(i);
8     }
9     for(int j = 7; j != 0; j--) {
10        interpolation(j);
11    }
12    for(int k = 0; k < frameSize; k++) {
13        tmpOut = intFilterOut[0][k] + filterOut[9][k] + filterOut[8][k] + filterOut[0][k];
14        ...
15    }
}

```

The function basically just calculates the output of every bandpass filter one after another. The summation of the downsampled outputs are all done by using the `interpolation()` function on all the downsampled filters. After the interpolation loop is finished all the outputs of the downsampled bands will be summed together in the array `intFilterOut[0]`. This is added together with the output of the non-downsampled filters and stored in the variable `tmpOut`.

For the audiointerface to be able to accurately play this equalized buffer, the buffer needs to be cast from doubles to shorts. This can be seen in code example 5.13

Code example 5.13: Casting to 16 bit integers

```

1 ...
2     if (tmpOut <= 32767 && tmpOut >= -32767) {
3         totalFilterOut[k] = (int)(tmpOut);
4     }
5     else if (tmpOut > 32767){
6         totalFilterOut[k] = 32767;
7     }
8     else {
9         totalFilterOut[k] = -32767;
10    }
11 }
12 return;
13 }

```

All that is being done here is essentially that `tmpOut` is checked to see if it's in the range of a 16 bit integer. If it is, it is cast to the nearest int, and if it isn't the `totalFilterOut[k]` is just set to the nearest valid short. The output buffer should then simply be sent to the audio interface to be played.

5.4 Design of controller

The purpose of the controller is to regulate the gain of the different bands in the equalizer, and thereby regulate the frequency response of the sound. The controller is making regulations based on audio samples from the Roland sound card and measured audio samples from the microphone unit.

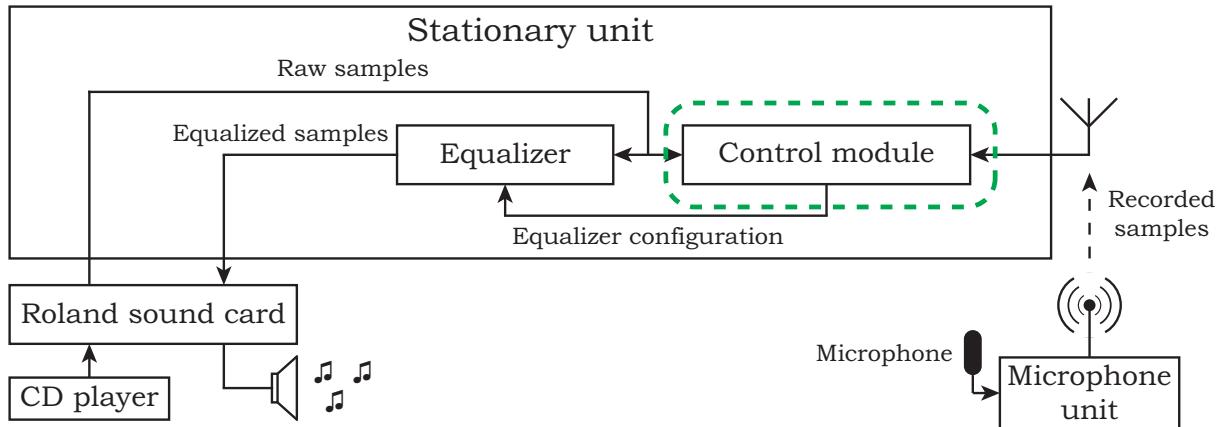


Figure 5.21: You are here

The best way to compare the reference audio and the measured audio would be to make a Discrete Fourier Transform(DFT) of both signals. By doing this a precise level of each individual frequency is known and by comparing the DFTs the most accurate adjustments of the equalizers band. However comparing the two DFTs may prove difficult, since there is an unknown, and possible variable, time delay between the reference audio and the measured audio. Therefore, comparing two signals in the frequency domain would require considerably much computation in order to find out which frequencies to compare. Another disadvantage of the DFT is that too much information is gathered. Since the equalizer only has 10 bands, a DFT is simply **OVERKILL**.

Another method is to calculate the Root Mean Square(RMS) of the individual equalizer bands for both the reference audio and the measured audio. By calculating the RMS-value the level of a equalizer band is known, and the necessary information to adjust the equalizer is known. Therefore calculating the RMS-value is preferred to calculating the DFT.

The RMS-value of a discrete time signal can be calculated as:

$$RMS = \sqrt{\frac{1}{N} \cdot \sum_{n=0}^{N-1} x[n]^2} \quad (5.4.1)$$

Here, N, is the total amount of samples to calculate the RMS over, and x[n] is the current sample.

When comparing the reference audio with the measured audio, it is assumed that the measured audio have a RMS-offset that is the same for all bands, since it is assumed that the microphone has a somewhat flat frequency response. The constant difference in RMS is due to the distance between the loudspeaker and the microphone. Since the distance is unknown and possible varying if the position of the microphone is moving, some correction is needed when calculating the RMS-value of the different bands.

To make this correction, the RMS-value of all bands are calculated for both the reference audio and the measured audio. This is done before equalizing so that the total RMS is calculated of the raw input. Thereafter the total RMS-value of the reference is divided by the total RMS of the measured audio, which gives the ratio between the two. If the ratio is over 1 the overall volume of the reference must bigger than the measured audio and vice versa.

These calculations are only for a single channel, but since two channels is used, it is necessary to do the total RMS calculations for the other channel as well. Thereafter the average RMS of the two channels is calculated, and this average will be used to correct the offset. This is **not cheating since the microphone is recording mono**.

The ratio between the total RMS of the reference and the microphone is expected to vary **a lot** due to the delay between calculating the RMS of the reference audio and the measured, or due to the user being noisy. To get a more steady ratio, a lowpass filter is designed with a cutoff frequency of 0,2 Hz. This filter makes the total ratio approach a value steadily, and the ratio is resilient to a sudden change in either of the RMS-values. The implementation of the calculation of the total RMS-ratio can be seen in code example 5.14.

Code example 5.14: Calculation of the total RMS-ratio

```
1 // Code will be inserted when I have cleaned it, and made variables with logical names. Just imagine a
   for-loop and a lowpass filter in the meantime.
```

When calculating the RMS-value, the newly arrived buffer with samples needs to be split into separate buffers where only frequencies of the specific bands are present. Luckily the equalizer does exactly that, so while equalizing, the RMS-value is calculated by squaring and summing the output samples, and when the equalizer is done, the total sum is divided by the number of samples where after the square is returned.

I need to be sure about how to multichannel before writing about it.

After the RMS-values of the different bands, and the total RMS, have been calculated for both the reference audio and the measured audio, they can be compared so that new gain factors for the equalizers bands can be found.

This is done by calculating the ratio between the RMS-values of the individual bands, where the total RMS-ratio of all the bands are multiplied with the RMS-value of the measured audio, so difference in volume is corrected. If the reference RMS is higher than the measured RMS while taking the total RMS ratio into account, it must indicate that the environment is dampening the band, and it must therefore be increased. The opposite must apply if the reference RMS is lower than the measured. The new RMS-ratio is thereafter lowpass filtered in order to make slow changes in the gain of the individual bands. This is done since it is assumed that sudden changes in the equalizers bands **would sound absolutely horrible**.

Having a equalizer band with a gain factor of more than one could potentially make samples over- or underflow, and therefore it is necessary to adjust the gain factor of all the bands. This is done by finding the biggest gain factor and thereafter dividing all the gains by this value. The biggest gain factor is now one, and the rest are scaled accordantly. The method can be seen in code example 5.15. Here **ratios[]** is an array of the type **double** that contains the gain factors of the different equalizer bands.

Code example 5.15: Scaling of gain factors

```

1 void ratioCompare(){
2     double biggestRatio = 0;
3     for(int i = 0; i < 10; i++){
4         if(ratios[i] > biggestRatio){ //Finding the biggest gain factor
5             biggestRatio = ratios[i];
6         }
7     }
8     for(int i = 0; i < 10; i++){ //Scaling gain factors accordingly
9         ratios[i] = ratios[i]/biggestRatio;
10    }
11 }
```

Og derefter virkede det hele bare helt perfekt, og alle var glade og tilfredse.

5.5 Recap of the whole system

Now that of the sub modules for the system have been designed, it is possible to put the pieces together. A familiar illustration of the whole system can be seen on figure 5.22.

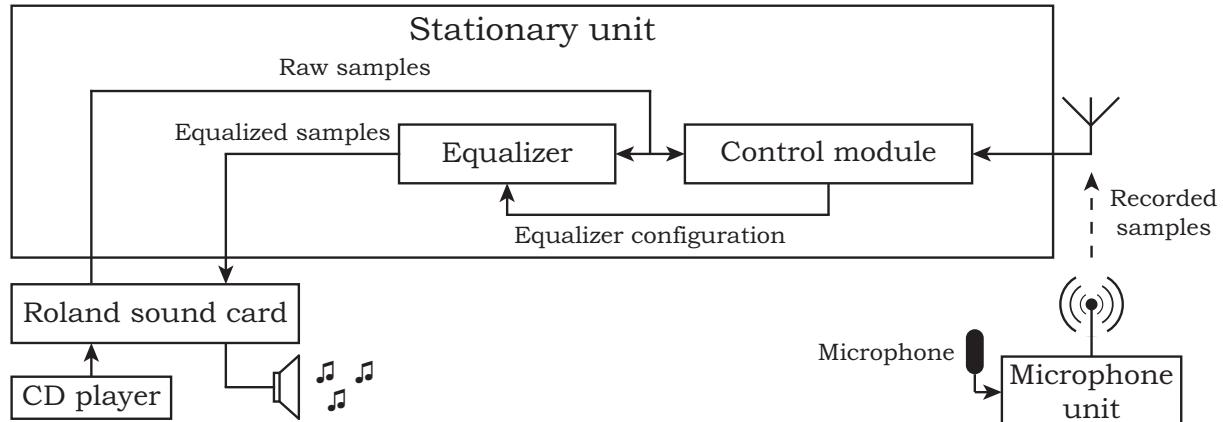


Figure 5.22: Illustration of the whole system.

The system starts with an analog audio input for the Roland sound card. This signal is converted to a digital signal and send to the stationary unit. Here it is send though the equalizer where it is filtered, depending on the current settings for the bandpass filters. The filtered digital sound is then send back to the Roland sound card where it is converted to an analog signal and played in a loudspeaker. The sound is then picked up by the microphone unit and transmitted via WiFi to the stationary unit. The recorded sound is then send to the control module where it is compared to the original sound signal, so new settings for the equalizer can be calculated. The filters are then adjusted accordingly and the whole process starts over again.

With the system pieces put together it is now possible to test the modules and the whole system to see if it can live up to the requirements.

Glossary

Ad hoc Ad hoc is a wireless networks that does not use any preexisting infrastructure.. 21

ALSA Advanced Linux Sound Architecture. A set of software modules on the Linux kernel, that handles interaction with sound cards.. 8

Downsampling A way of changing the sample rate of a signal by only taking every n sample of some integer n.. 36

Equalizer An electronic device for adjusting the frequency response of a system.. 4

FIR Filter Finite impulse response filters. Filters whose impulse responses are zero after a certain point. see. 7

Frequency response The way a system output behaves in terms of magnitude and phase, as a function of the frequency of the input. 1

IIR Filter Infinite impulse response filters. Filters whose impulse responses continues forever. This property is known from the impulse response of analog filters. see. 7

Raspberry Pi Series of single board computers, often used for small projects.. 9

Bibliography

- [1] Last seen: 21/09/17. URL: https://www.raspberrypi.org/app/uploads/2014/07/rsz_b-.jpg.
- [2] *Advanced Linux Sound Architecture*. Last seen: 13/09/17. URL: https://wiki.archlinux.org/index.php/Advanced_Linux_Sound_Architecture.
- [3] AKM. *AK5371 microphone*. Last seen: 23/10/2017. URL: <https://media.digikey.com/pdf/Data%20Sheets/AKM%20Semiconductor%20Inc.%20PDFs/AK5371.pdf>.
- [4] *Bluetooth Core Specification v5.0*. Bluetooth SIG Proprietary. 2016. URL: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=421043&_ga=2.131475212.2088466289.1505901779-621389646.1505901779.
- [5] International Electrotechnical Commission. *Audio Recording - Compact Disc digital audio system*. 1999.
- [6] *ISO 266:1997 - Acoustics - Preferred frequencies*. ISO. 1997.
- [7] Bjorn Roche. *Analog vs. Digital EQ*. Last seen: 20/09/17. URL: <http://www.xowave.com/doc/effect/eq/analogvsdigital.shtml>.
- [8] Roland. *UA-25EX Sound card*. Last seen: 18/10/2017. URL: http://cdn.roland.com/assets/media/pdf/UA-25EX_OM.pdf.
- [9] Margaret Rouse. *definition: microcontroller*. Last seen: 13/09/17. URL: <http://internetofthingsagenda.techtarget.com/definition/microcontroller>.
- [10] Gary W. Siebein. "Architectural acoustics". In: *Access Science* (2014). URL: <http://www.accessscience.com.zorac.aub.aau.dk/content/architectural-acoustics/048700>.
- [11] *Sound Card Components*. Last seen: 13/09/17. URL: <https://www.pctechguide.com/sound-cards/sound-card-components>.
- [12] Steve Taranovich. *Integration Choices: Analog Filters vs. Digital Filters*. Last seen: 21/09/17. URL: https://www.planetanalog.com/author.asp?section_id=3065&doc_id=560512.
- [13] Floys Toole. "Sound-reproducing systems". In: *Access Science* (2014). URL: <http://www.accessscience.com.zorac.aub.aau.dk/content/637630>.
- [14] *What is an equalizer?* Last seen: 20/09/17. URL: http://www.yamahaproaudio.com/global/en/training_support/selftraining/pa_guide_beginner/equalizer/.
- [15] J. William Whitehart. *Signal Processing*. Last seen: 21/09/17. URL: <https://www.accessscience.com/content/signal-processing/757755#757755s011>.
- [16] Wikipedia. *Equalization (audio)*. Last seen: 20/09/17. URL: [https://en.wikipedia.org/wiki/Equalization_\(audio\)](https://en.wikipedia.org/wiki/Equalization_(audio)).
- [17] Wikipedia. *Figure about audio bit debt*. Last seen: 13/09/17. URL: https://en.wikipedia.org/wiki/Audio_bit_depth#/media/File:4-bit-linear-PCM.svg.
- [18] Wikipedia. *Figure about Nyquist-Shannon sampling theorem*. Last seen: 12/09/17. URL: <https://en.wikipedia.org/wiki/File:CPT-sound-nyquist-theorem-1.5percycle.svg>.

- [19] Wikipedia. *Nyquist–Shannon sampling theorem*. Last seen: 12/09/17. URL: https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem.
- [20] Tracy V. Wilson. *How Sound Card Works*. Last seen: 13/09/17. URL: <http://computer.howstuffworks.com/sound-card3.htm>.

Wireless bit rate approximation A

Purpose

The purpose of this test is to find an approximation of the bit rate between the microphone unit and stationary unit, and to test the bit rate, on different distances. As specified in 4.6, a bit rate of at least 2 Mb/s (0.25 MB/s) is needed.

Procedure

To test the wireless connection speed 10.000 packets of 40 kB, a total of 400 MB each, are send from the client to the host using the UDP protocol. This is then timed so that the connection speed can be calculated. The time is measured using a stop watch.

Materials

Items used for the test:

- Raspberry Pi 3 Model B Vi 3
- Raspberry Pi Zero W
- Stopwatch
- Measuring Tape

For the test C code is used to send the packages [UDP-Client_Speedtest](#) and [UDP-Server_Speedtest](#)

$$\text{Message length} = 10.000 \text{ Int32} = 40.000 \text{ B} \quad (\text{A.0.1})$$

$$\text{Messages send} = 10.000 \quad (\text{A.0.2})$$

$$\text{Data send} = 10.000 \cdot 40.000 = 400 \text{ MB} \quad (\text{A.0.3})$$

$$\text{Distance} = 0,5, 1 \text{ and } 5 \text{ m} \quad (\text{A.0.4})$$

Results

Test	1	2	3	4	5	
Measured time	97	96	93	71	95	[s]
Distance	1	1	1	0,5	5	[m]
Connection speed	4,12	4,17	4,3	5,63	4,44	[MB/s]

Table A.1: Data from tests

Sources of error

Since the timing solely relies on human reaction time, and since the program is not sending the same way as it will do in the finalized prototype, this test will only give an idea of what speeds can be expected.

Conclusion

The bit rate between the two Raspberry Pi units is tested to at least 16 times faster than the requirement of 0.25MB/s, even taking into account any sources of error in the test, or uncertainties such as distance the connection speed is still perceived as plenty.

Test of package loss rate

B

Purpose

The purpose of this test is to find how much package loss can be expected if any. The code used for this test only checks whether the sum of the data corresponds to the checksum send.

Little to no package loss is expected, this is due to the fact that the two Raspberry Pi's have a dedicated line of communication. The only disturbance would be interference from other devices on the same frequency as the ad hoc network on 2.4 GHz.

Procedure

To make certain that every bit is tested, every value possible in a signed short is sent. Furthermore these values are sent in messages of 2048 bytes each, a checksum for each message is calculated and set at the end of the message. To send all the values 64 messages are sent before the procedure is repeated. The test ran for about 16 hours, with a distance between the two units of about 5 meters.

Materials

- Raspberry Pi 3 Model B V1.3 (stationary unit)
- Raspberry Pi Zero W (microphone unit)
- Measuring tape

For the test C code is used to send the packages [UDP-Client_packageloss](#) and [UDP-Server_packageloss](#)

Results

About 250 GB of messages was transmitted between the two units, no package loss was recorded during this test.

Sources of error

Since there is no acknowledgement in the UDP protocol it is possible that the microphone unit has transmitted a package without the stationary unit being ready and so there would be a package loss. In the final prototype the microphone unit will take far longer to sample the sound than it will take to process it, therefore this will not be a problem.

Conclusion

There have been no package loss during the 16 hours in which the test have been conducted.

The bilinear transform



The bilinear transform is obtained by looking at the Laplace transform of a discrete time function i.e a continuous time function multiplied by an impulse train $\sum_{n=0}^{\infty} \delta(t - nT_d)$ where T_d is the sample time $1/f_s$.

$$x(t) \cdot \sum_{n=0}^{\infty} \delta(t - nT_d) \quad (\text{C.0.1})$$

As the impulse train is zero at every $t \neq nT_d$ the equation can be rewritten as

$$\sum_{n=0}^{\infty} x(t)\delta(t - nT_d) = \sum_{n=0}^{\infty} x(nT_d)\delta(t - nT_d) \quad (\text{C.0.2})$$

The discrete time function $x[n]$ is defined as the value of $x(t)$ at every sampling time, which is $x(nT_d)$ for every integer n . That means the equation can be written as

$$\sum_{n=0}^{\infty} x[n]\delta(t - nT_d) \quad (\text{C.0.3})$$

By taking the Laplace transform of this you will get

$$\begin{aligned} X_d(s) &= \int_0^{\infty} \sum_{n=0}^{\infty} x[n] \cdot \delta(t - nT_d) \cdot e^{-st} dt \\ &= \sum_{n=0}^{\infty} x[n] \int_0^{\infty} \delta(t - nT_d) \cdot e^{-st} dt \end{aligned} \quad (\text{C.0.4})$$

As the delta function has a value of 1 at exactly $t = nT_d$ the integral reduces to just e^{-snT_d} .

$$X_d(s) = \sum_{n=0}^{\infty} x[n]e^{-snT_d} \quad (\text{C.0.5})$$

This is exactly the same as the z-transform

$$X(z) = \sum_{n=0}^{\infty} x[n]z^{-n} \quad (\text{C.0.6})$$

where $z = e^{sT_d}$

Isolating for s you get

$$\begin{aligned} z &= e^{sT_d} \iff \\ sT_d &= \ln(z) \iff \\ s &= \frac{1}{T_d} \ln(z) \end{aligned} \quad (\text{C.0.7})$$

Thereby a transfer function in the analog s-domain can be transformed to the digital z-domain by simply replacing s with $\frac{1}{T_d} \ln(z)$. However since it is desired to keep the transfer functions linear after the transformation, $\ln(z)$ is approximated to $2\frac{z-1}{z+1}$. The variable s can then be given as

$$s \approx \frac{2}{T_d} \frac{z-1}{z+1} \quad (\text{C.0.8})$$

By substituting s in an analog transfer function with this expression you will therefore get an equivalent transfer function in the z-domain.

Frequency warping

One problem with the bilinear transform is that the frequency axis of a discrete transfer function does not exactly match the frequency axis of the analog transfer function it is obtained from. This can be seen by substituting s and z with their corresponding frequency axis' in the transformation.

$$j\omega_a = \frac{2}{T_d} \frac{e^{j\omega_d T_d} - 1}{e^{j\omega_d T_d} + 1} = \frac{2}{T_d} \frac{1 - e^{-j\omega_d T_d}}{1 + e^{-j\omega_d T_d}} \quad (\text{C.0.9})$$

By simplifying this you obtain:

$$j\omega_a = \frac{2}{T_d} \frac{e^{j\omega_d T_d/2} \cdot e^{-j\omega_d T_d/2} - e^{-j\omega_d T_d/2} \cdot e^{-j\omega_d T_d/2}}{e^{j\omega_d T_d/2} \cdot e^{-j\omega_d T_d/2} + e^{-j\omega_d T_d/2} \cdot e^{-j\omega_d T_d/2}} \quad (\text{C.0.10})$$

$$= \frac{2}{T_d} \frac{e^{-j\omega_d T_d/2} \cdot (e^{j\omega_d T_d/2} - e^{-j\omega_d T_d/2})}{e^{-j\omega_d T_d/2} \cdot (e^{j\omega_d T_d/2} + e^{-j\omega_d T_d/2})}$$

$$= \frac{2}{T_d} \frac{2j \cdot \sin(\omega_d T_d/2)}{2 \cdot \cos(\omega_d T_d/2)}$$

$$= \frac{2j}{T_d} \tan\left(\frac{\omega_d T_d}{2}\right) \leftrightarrow$$

$$\omega_a = \frac{2}{T_d} \tan\left(\frac{\omega_d T_d}{2}\right) \quad [\text{rad/s}] \quad (\text{C.0.11})$$

and the inverse:

$$\omega_d = \frac{2}{T_d} \arctan\left(\frac{\omega_a T_d}{2}\right) \quad [\text{rad/s}] \quad (\text{C.0.12})$$

This means that if you are looking for a digital filter with a specific behavior at a specific frequency w_d , this filter can be obtained from an analog filter with this behavior at $w_a = \frac{2}{T_d} \tan\left(\frac{\omega_d T_d}{2}\right)$.

The important frequencies in an analog system e.g. the cutoff frequencies of a filter should for this reason be pre-warped using the expression from equation C.0.11, if it is desired for the digital system to behave in the same way.

C-code used in the project

D

Code example D.1: Function for the anti aliasing filters and downsampling

```

1 void antiAliasing(int n) {
2     int i = 0;
3     int dsRatio = power(2,n);
4     while(i != frameSize/dsRatio) {
5         if(n == 0) {
6             aaIn[n][0] = inputArray[i];
7         } else {
8             aaIn[n][0] = aaArray[i];
9         }
10
11        aaOut[n][0] = 0.2132*aaIn[n][0] + 0.4264*aaIn[n][1] + 0.2132*aaIn[n][2] + 0.3392*aaOut[n][1] -
12            0.192*aaOut[n][2];
13        aaIn[n][2] = aaIn[n][1]; aaIn[n][1] = aaIn[n][0];
14        aaOut[n][2] = aaOut[n][1]; aaOut[n][1] = aaOut[n][0];
15        // Downsampling
16        if(i%2 == 0) {
17            aaArray[i/2] = aaOut[n][0];
18        }
19        i++;
20    }
21    return;
}

```

Code example D.2: Function for the interpolation filters and upsampling

```

1 void interpolation(int n) {
2     int dsRatio = power(2,(n-1));
3     int i = 0;
4     while(i < frameSize/dsRatio) {
5         if(i%2 == 0) {
6             if(n == 7) {
7                 iIn[n][0] = filterOut[n][i/2];
8             } else {
9                 iIn[n][0] = intFilterOut[n+1][i/2] + filterOut[n][i/2];
10            }
11        } else {
12            iIn[n][0] = 0;
13        }
14        // Filtering
15        iOut[n][0] = 0.3077*iIn[n][0] + 2*0.3077*iIn[n][1] + 0.3077*iIn[n][2] - 0.005152*iOut[n][1] -
16            0.4465*iOut[n][2];
17        iOut2[n][0] = 0.3077*iOut[n][0] + 2*0.3077*iOut[n][1] + 0.3077*iOut[n][2] - 0.003703*iOut2[n][1] -
18            0.03957*iOut2[n][2];
19        // Saving output
20        intFilterOut[n][i] = 2 * iOut2[n][0];
21        iIn[n][2] = iIn[n][1]; iIn[n][1] = iIn[n][0];
22        iOut[n][2] = iOut[n][1]; iOut[n][1] = iOut[n][0];
23        iOut2[n][2] = iOut2[n][1]; iOut2[n][1] = iOut2[n][0];
24        i++ ;
25    }
26    return;
}

```

Matlab scripts for equalizer calculations E

Code example E.1: Calculation of filter edge frequencies and transfer functions

```

1 w0 = [31.25 62.5 125 250 500 1E3 2E3 4E3 8E3 16E3] * 2 * pi;
2
3 ratio = 0.73;
4 order = 4;
5
6 for i = 1:10 % The calculations are done for each bandpass filter
7     WL(i) = w0(i) * ratio;
8     WH(i) = w0(i) * 1/ratio;
9
10    [a(i,:), b(i,:)] = butter(order, [WL(i) WH(i)], 's'); % When 'butter()' receives 2D vector
11        as 2nd input, it will automatically make a bandpass filter with WL(i) and WH(i) as edge
12        frequencies and with an order of twice the first input.
13    H(i) = 0.92 * tf(a(i,:), b(i,:));
14 end

```

Code example E.2: Bilinear transform of the 8 kHz filter

```

1 fs = 44100;
2
3 WLd = WL(9)/fs;
4 WHd = WH(9)/fs;
5
6 OL = 2*fs*tan(WLd/2); % prewarped edge frequencies
7 OH = 2*fs*tan(WHd/2);
8
9 %optaining S-domain transferfunction
10 order = 4;
11 [a, b] = butter(order, [OL OH], 's');
12 H = tf(a, b); % Optaining the prewarped transferfunction
13
14 %optaining z-domian transferfunction(s)
15 HD = c2d(H,1/fs,'tustin'); % Performing the bilinear transform on the prewarped transfer
16 function.
17 [N,D] = tfdata(HD); % Optains the numerator and denominator of the transfer function
18 N = cell2mat(N);
19 D = cell2mat(D);
20 zeroes = roots(N);
21 poles = roots(D);
22 zplane(zeroes,poles);

```

Code example E.3: Divison into 4 biquads

```

1 ts = 1/fs;
2 z = tf('z',ts);
3 K = 0.007172; % Constant factor of original transfer function.

```

```
4 | c = K^(1/4);  
5 |  
6 | TF1 = c * ((z - zeroes(1))*(z - zeroes(2)))/((z - poles(1))*(z - poles(2)));  
7 | TF2 = c * ((z - zeroes(5))*(z - zeroes(6)))/((z - poles(3))*(z - poles(4)));  
8 | TF3 = c * ((z - zeroes(3))*(z - zeroes(4)))/((z - poles(5))*(z - poles(6)));  
9 | TF4 = c * ((z - zeroes(8))*(z - zeroes(7)))/((z - poles(7))*(z - poles(8)));  
10| TFT = TF1*TF2*TF3*TF4;
```

Test of equalizer F

Purpose

The purpose of this report is to test whether or not the equalizer module described in section 5.3 is able to meet the requirements specified in section 4.4.

The test is divided into **X** different tests.

Materials for all measurements

A simple test bench for the equalizer has been written in c. The test program simply loads a sound file and puts the samples through the equalizer 512 samples at a time. The output of the equalizer are stored in a file for analysis. **This program will of course be sent with the report.**

F.1 Test of amplitude response

Purpose

This test is conducted to see if the equalizer can meet the requirements to it's amplitude response specifically requirement 4.3 that says that the gain from 30 Hz to 20 kHz should be between ± 1 dB, when all filters are in a neutral position.

Procedure

To test that the gain is the same for the whole passband, a "sine sweep" (or "chirp") is sent through the equalizer. The sine sweep covers frequencies from 10 Hz to 22 kHz. It had a amplitude of 15000¹ and takes 50 seconds to sweep from 10 Hz to 22 kHz. Since it is desired to measure all the bands an equal amount of time, the sweep is made logarithmic. All the bands are set to have no attenuation factor.

After the filtering the output signal is stored in a file.

Results

The results are plotted in figure F.1 using MATLAB

¹This value is semi-arbitrarily chosen because it is about half the maximum value of a signed short.

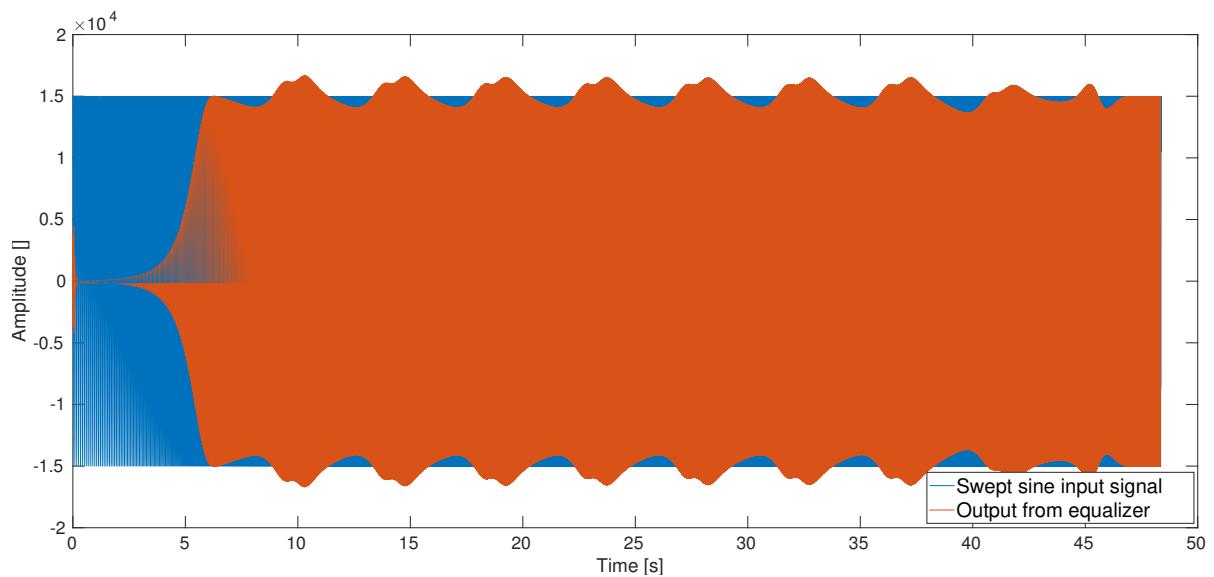


Figure F.1: Input and output of equalizer with no attenuation

It can be seen low frequencies are attenuated a lot. This make sense since the lowest frequency band attenuates all frequencies under about 22 Hz. Theres no attenuation at the higher frequencies since the 16 kHz band was made as a high pass filter. There are some high frequency signals at the very start of the measurement which can be seen more closely in figure F.2.

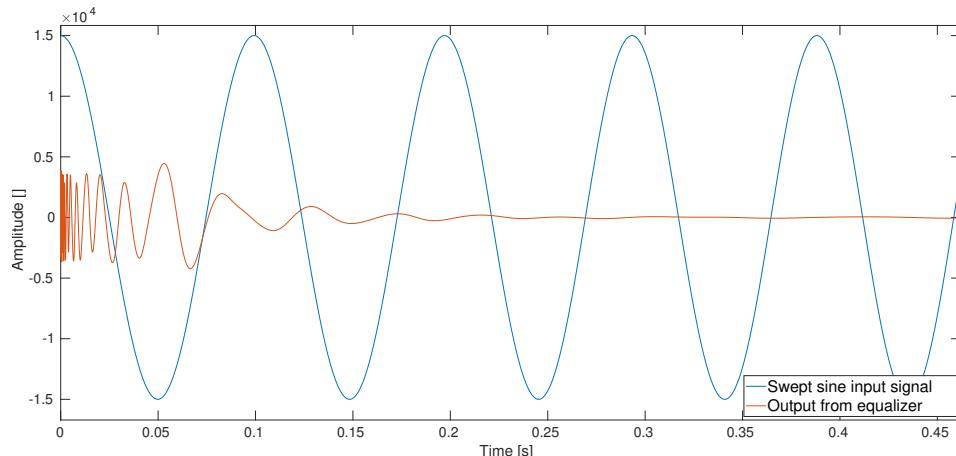


Figure F.2: A closer look at figure F.1 at the start of the measurements

The input signal is a cosine and starts at 1. Because of this and since the output signal has lots of different frequencies, it assumed that the signal at the start is something similar to an impulse response and that it can be safely ignored.

Analysis

From the measured data the highest and lowest amplitude between 30 Hz and 20 kHz has been obtained. From these the highest and lowest gains in dB are calculated

$$A_{o,min} = 13.67 \cdot 10^3 \quad (\text{F.1.1})$$

$$\iff G_{min} = 20 \cdot \log_{10} \left(\frac{A_{o,min}}{A_i} \right) = 20 \cdot \log_{10} \left(\frac{13.67 \cdot 10^3}{15 \cdot 10^3} \right) = -0.806 \text{dB}$$

$$A_{o,max} = 16.68 \cdot 10^3 \quad (\text{F.1.2})$$

$$\iff G_{max} = 20 \cdot \log_{10} \left(\frac{A_{o,max}}{A_i} \right) = 20 \cdot \log_{10} \left(\frac{16.68 \cdot 10^3}{15 \cdot 10^3} \right) = 0.922 \text{dB}$$

The equalizer is therefore able to meet the requirement in the range 30 Hz to 20 kHz.

The lowest frequency that fits inside the ± 1 dB range can also be found to be 23.96 Hz. This is measured by finding where the amplitude crosses $10^{\frac{-1}{20}} \cdot 150000 = 13368.7$. The period time of the signal is then found at this point, and the frequency is then calculated. At high frequencies the amplitude stays within the range all the way up to the highest measured frequency at 22 kHz.

Conclusion

Based on these measurements it is concluded that the gain ripple in the passband is ± 0.922 dB. It is also concluded that the range in which the ripple is ± 1 dB is from 24 Hz to 22 kHz.

F.2 Test of band attenuation

Purpose

The purpose of this test is to measure if the equalizer is able to attenuate every band by 20 dB.

Procedure

F.3 Test of phase response

F.4 Test of computation time

F.5 Conclusion

Everything works, let's go on holiday!