
Self Adjusting Loudspeaker System

December 2013

Group 503

Project report



AALBORG UNIVERSITY

STUDENT REPORT

Department of Electronic Systems

Fredrik Bajers Vej 7

DK-9220 Aalborg Ø

<http://es.aau.dk>

Title:

Self Adjusting Loudspeaker System

Theme:

Analog and digital systems in interaction with the surroundings.

Projekt Period:

Fall semester 2013

Project Group:

503

Authors

Henrik Aarup Vesterager

Jacob Møller Hjerrild Hansen

Søren Bøgeskov Nørgaard

Malthe Bilgram Jensen

Igor Aleksandrovich Syrytsin

Lasse Thomsen

Constantin de La Seiglière

Supervisor:

Søren Krarup Olesen

Number of Prints: 2

Number of Pages: 147

Date for Submission: December 19, 2013

Abstract:

In this report a self adjusting loudspeaker system is developed. The purpose of a self adjusting loudspeaker system is to equalize reverberations and standing waves in a room such that the speaker output is as close to the original audio source.

The product consists of two parts: A Base Station and a Mobile Station. The Base Station is connected to the users audio player. It's purpose is to do filtering of the audio signal and output the equalized signal into a speaker system.

The Mobile Station is recording the audio in the room and transmits it to the Base Station so it knows how to equalize the audio. The transmission from Mobile Station to Base Station is done via Bluetooth.

The system has been implemented in three blocks:

1. The program running on the Base Station is programmed in C using the ALSA API.
2. The filter is designed as a C library for the Base Station program.
3. The wireless communication is designed so it have a receiver and a transmitter. The receiver connected directly to the sound input of the Base Station.

The functionalities of the implementation part are verified through multiple tests.

A prototype has been developed and is working as intended. The Base Station consists of a laptop and a USB sound card and a Mobile Station consists of a laptop and a USB sound card with a microphone connected to it. The Mobile Station sends sound data to the Base Station via wireless Bluetooth connection.

From measurements it can be seen that the system is counteracting noise that is different from the audio source.

Henrik Aarup Vesterager

Jacob Møller Hjerrild Hansen

Søren Bøgeskov Nørgaard

Malthe Bilgram Jensen

Igor Aleksandrovich Syrytsin

Lasse Thomsen

Constantin de La Seiglière

Preface

Reading Guide

The report is structured in four sections. In each section different aspects of the project will be analyzed.

- I Problem Analysis
- II Technical Solution
- III Conclusion
- IV Appendices

The Problem Analysis will be dealing with definition of the technology which will be developed and therefore it contains the necessary discussions for a Requirement Specification, which will result in guidelines and instructions for The Technical Solution. The Technical Solution contains development of the system and solutions to the issues discussed. The Discussion part contains a summary and conclusion, and reflections regarding the obtained system technology.

Appendices

Appendices are found after the main report A.1 and on the attached CD. The appendices contains all source code, an extended analysis, measurement reports and work sheets. The CD also contains a digital version of the report and the web pages used. A complete list of the CD content can be found here D. When referred to the CD the following symbol is used

 path/to/stuff

All figures, tables and equations are referred to by the number of the chapter they are used in, followed by a number indicating the number of figure, table or equation in the specific chapter. Hence each figure has a unique number, which is also printed at the bottom of the figure along with a caption. The same applies to tables and equations, the latter of which have no captions. Appendices are referred to by capital letters instead of chapter numbers.

Bibliography

At the very end of the main report, a Bibliography is listed which contains all sources of information used in the report. In the Bibliography books are indicated with author, title, publisher and year. Web pages are indicated with author and title. All information sources are referred to by the number which they feature in the list. This will look like this: [number]. If the sources reference is written just before a final dot, then it represents the source of information for the sentence in question. If the reference is placed after the dot it represents the source of information for the section in question.

Contents

1	Introduction	1
1.1	System Description	1
1.2	Initiating Problem	2
I	Problem Analysis	3
2	Technology Definition	4
2.1	Block Description	4
2.2	Measurement of Room Frequency Response	5
2.3	Microphone System	7
2.4	Standards for Dividing the Audible Frequency Range into Bands	7
2.5	Filter Analysis	8
2.6	Computer Platform	12
2.7	Propagation Delay	13
2.8	Size of the Audio Buffer	13
2.9	Wireless Communication	13
3	Problem Statement	15
4	Requirement Specification	16
4.1	General Description	16
4.2	Functionality Description	16
4.3	Description of Interfaces	17
4.4	Specific Requirements	17
5	Test Specification	19
5.1	General Tests	19
5.2	Filter	19
5.3	Controller	20
5.4	Mobile Station (general)	20
5.5	Wireless	20
5.6	Microphone System	21
II	Technical Solution	24
6	Main Program	26
6.1	Thread Test Code	27
7	Audio Interface	31
7.1	Code	32
7.2	Low Pass Filter	38
7.3	Low Pass Filter	39
7.4	Test of ALSA PCM Interface With a Filter	41

8	Filters	47
8.1	Band Pass Design	47
8.2	Choices of Design	47
8.3	Design Application	48
8.4	Using MATLAB to Obtain Filter Coefficients	53
8.5	C Code Implementation—Cascaded Biquads	55
8.6	Test	59
8.7	Implementation of Treble Filter	62
9	Wireless	64
9.1	Bluetooth Module RN52	64
9.2	Circuit	65
9.3	Control of the RN52	66
9.4	Tests	66
10	Implementation	68
10.1	Main Program Implementation With ALSA	68
10.2	Implementation of the ALSA With the Filter Library	75
10.3	Main Program With ALSA PCM Interface and Bluetooth	81
10.4	Signal Amplifier for Bluetooth module	81
III	Conclusion	83
11	Acceptance Test	84
12	Conclusion	85
IV	Appendices	86
A	Measurement Reports	87
B	Worksheets	115
C	Extended Analysis	121
D	Source Code	131
	Bibliography	145

1 Introduction

The sound quality of reproduced sound is not only dependent on the electronic circuits and speakers. When listening to playback audio inside a room, the sound that reaches the listener is a product of room dimensions, surfaces and furniture etc. Therefore, the frequency characteristics are varying with the location of the listener and the speakers, relative to the room. The aim in this project is to design an electrical audio system, which continuously analyzes the frequency response for a loudspeaker inside a room. Once this frequency response has been found, the system will make it as flat as possible, by equalizing the output signal from the system to the speaker. This is done to improve the sound reproduction and thereby improve the sound quality for the listener. This design can be used in augmented reality or in private communication systems, by users of loudspeaker systems that are moving physically while playing games, listening to music or watching video. Using the self adjusting loudspeaker system will then help improve the experience with audio systems inside a room. Since the focus is to design a system for people that are changing location, by moving around, the system should be wireless. Furthermore it has been chosen to use a Raspberry Pi with a Linux implementation as the Base Station.

1.1 System Description

The general purpose of the system is to compensate for any changes in the sound as the listening person is moving around in the livingroom, so that the listening person always has the same sound experience. The ideal soundsystem for home use should work with a stereo sound system, see Figure 1.1.

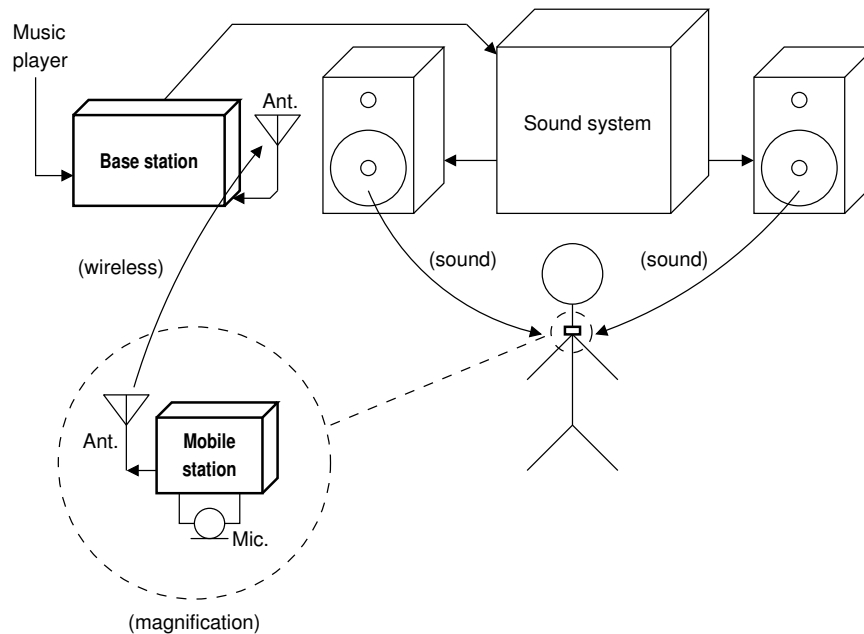


Figure 1.1: Real life drawing of project.

The measuring system must work at normal audio frequencies, 20 Hz up to around 20 kHz. The principle in a measurement could be as follows: The audio interface/computer plays music which is sent to the loudspeakers. The microphone picks up the audio signal and sends it through a wireless connection back to the audio interface/computer. The software on the computer tunes the various filters in near real time because a small time delay is introduced, so that a satisfying frequency characteristic is obtained. An example of a compensation situation, can be seen on Figure 1.2. In this example the red curve is

compensating for the recorded sound and resulting in a flat frequency response. In this case an inverse of the recorded frequency response is made and added to the recorded frequency response.

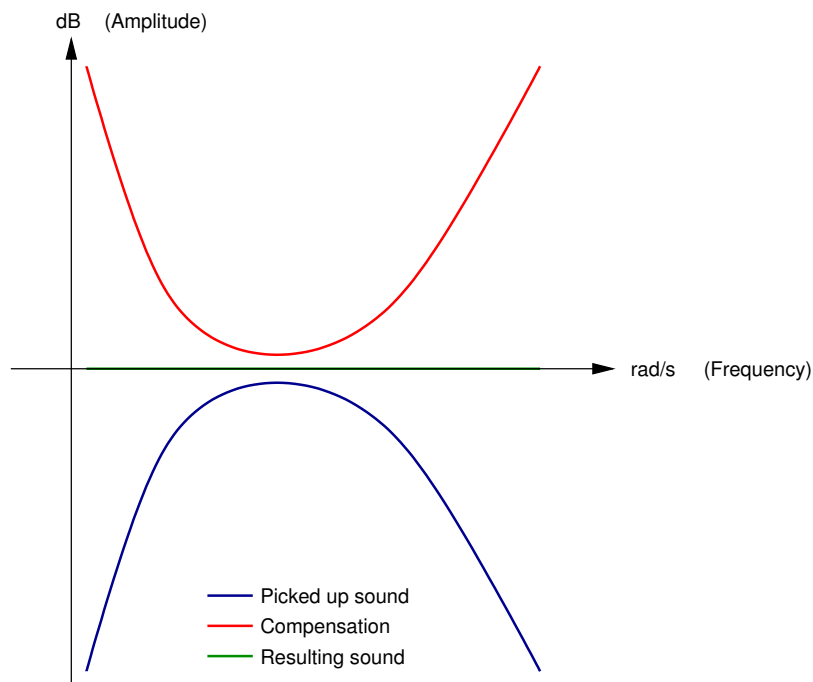


Figure 1.2: Concept of compensation. The blue frequency curve is first picked up. The computer then adjusts its filter as the red curve, and the resulting curve is a flat curve—the green one.

1.2 Initiating Problem

The initiating problem aims to initialize an analysis of the problem to clarify the functionality and requirements of the system. The initializing problem is as follows:

Which considerations should be taken into account when building a wireless self adjusting loudspeaker for use with augmented reality?

I

Problem Analysis

2 Technology Definition

2.1 Block Description

In this chapter the technology will be defined, throughout the analysis. This is done by investigating which parts that is included in an audio system, which is able to carry out a frequency analysis and apply level regulations based on the frequency analysis. Also a measurement of the frequency response in a room is carried out. This will lead to the problem statement.

To give an overview of the system, it is initially divided into four modules. This is done to outline the interfaces between these modules. On this block diagram the interfaces are noted with a V and a P. The V is for analogue voltage and the P means air pressure variations. Each module will be further described in detail in the following sections, which also leads to the definition of what each system will contain and how this will be developed. (e.g. The computer will contain AD/DA converters, filters and realtime regulator software).

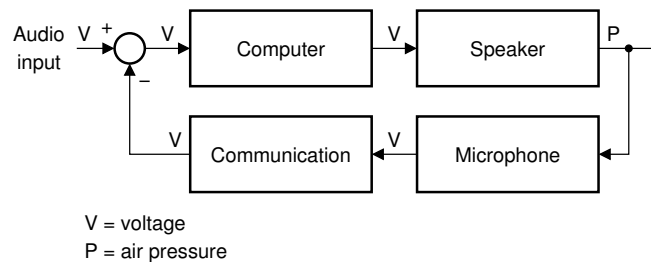


Figure 2.1: Overview of the system

2.1.1 Computer

The computer is a real time system, that compares the output from the microphone system to the reference input signal which is fed to the speaker. This is done to equalize the output to the speaker, so it is as close as possible to the reference in the listeners position. The idea is to pass the signals through a real time digital filter to have a dynamic system.

The computer could have two analog inputs and one analog output, where the inputs consists of a AUX sound input and a microphone output through the communication block. The analog output is used for the speaker. There are many ways of implementing this computer into the system, one way is to use a raspberry pi or some sort of small controller with the right requirements.

2.1.2 Speaker

The speaker block is composed by an amplifier and a loudspeaker. The amplifier receives an analog input signal from the computer. Then the loudspeaker produce sound in response to the electrical audio signal input. The analog input of the loudspeaker is in Volt and the air pressure output is in Pascal.

2.1.3 Microphone

This module will be in charge of measuring the sound from the speakers.

The microphone will sense the sounds from the speaker and pass this signal to the computer through a pre-amplifier and a communication module. The microphone will be located on a person, so there could

be a continuous change in the speakers output, based on the input received from the microphone.

2.1.4 Communication

This module handles all communication between the computer and the microphone system. This could be a simple wired- or a wireless communication.

The computer will need some sort of communication device, to receive the signal sent from the microphone system and likewise will the microphone system need a device to transmit. The data could be converted into digital at the microphone system and sent digitally or the signal could be sent directly analog and the analog to digital conversion would happen in the computer.

This was an overview of the complete system, how it is suppose to be carried out. Therefore the main interfaces is now defined and a feedback system has been initialized. To define the specific parameters in the feedback system, the output which is the sound in a room will first be investigated further in the following section.

2.2 Measurement of Room Frequency Response

Sound quality may be defined as the ability to reconstruct the original signal from its natural appearance. When developing an electrical system to achieve this, all the interferences in the whole chain of the system has to be found. The main sources of interference are listed below.

- Propagation through air.
- Propagation through electrical circuits.

Because this project focuses on the development of the electrical circuits, which will equalize the frequency response with respect to a reference signal, the acoustic reflections will briefly be described in this section. A measurement of the frequency response has been carried out. This was done by recording the playback of pink noise, in different locations of a room with a height of 4 m, a width of 4 m and a length of 6 m. The recordings were conducted with a Zoom H4 HD-recorder (AAU no. 64694) and one speaker of type Genelec 1032A. The placement of speaker and microphone is shown in Drawing 2.3. Datasheets for speaker and microphone can be found on the CD [16].

/Datasheets

The results are illustrated in Figure 2.2 and Figure 2.4. This measurement, shows that the frequency content is distributed unequally. These unwanted fluctuations are mostly placed in the lower end of the spectrum. In the interval below 30 Hz all recordings lacks magnitude, due to the limitations in the frequency response of the speaker.

From 35 Hz to 100 Hz all recordings shows fluctuations of around 10 dB above the reference, which should be compared to the higher end attenuation of the same recordings. In the interval between 120 Hz to 300 Hz some signals are above the reference by 10 dB, though some signals are below by 6 dB. Above 1 kHz the signal depicted in blue is a bit above the reference, while the green signal is attenuated. Both of these should be compared to the lower end of the respective signal, because each graph in the plots is scaled, so that the max frequency gain of every signal equals 0 dB. All of the signals shows further attenuation over 15 kHz, due to speaker and microphone limitations. These measurements have been done to show that there are undesired fluctuations in the propagation through the air. It should also be mentioned here, that every speaker has directivity characteristics, which affect the listeners experience in each position. As the listener is exposed to this unequal frequency response like the ones measured, it is fair to conclude that the listener could use some help in order to adjust the frequency spectrum. As a side note it can be seen that the greatest amplifications is in the low end of the frequency band, which makes sense in comparison with room dimensions and the standing waves phenomenon.

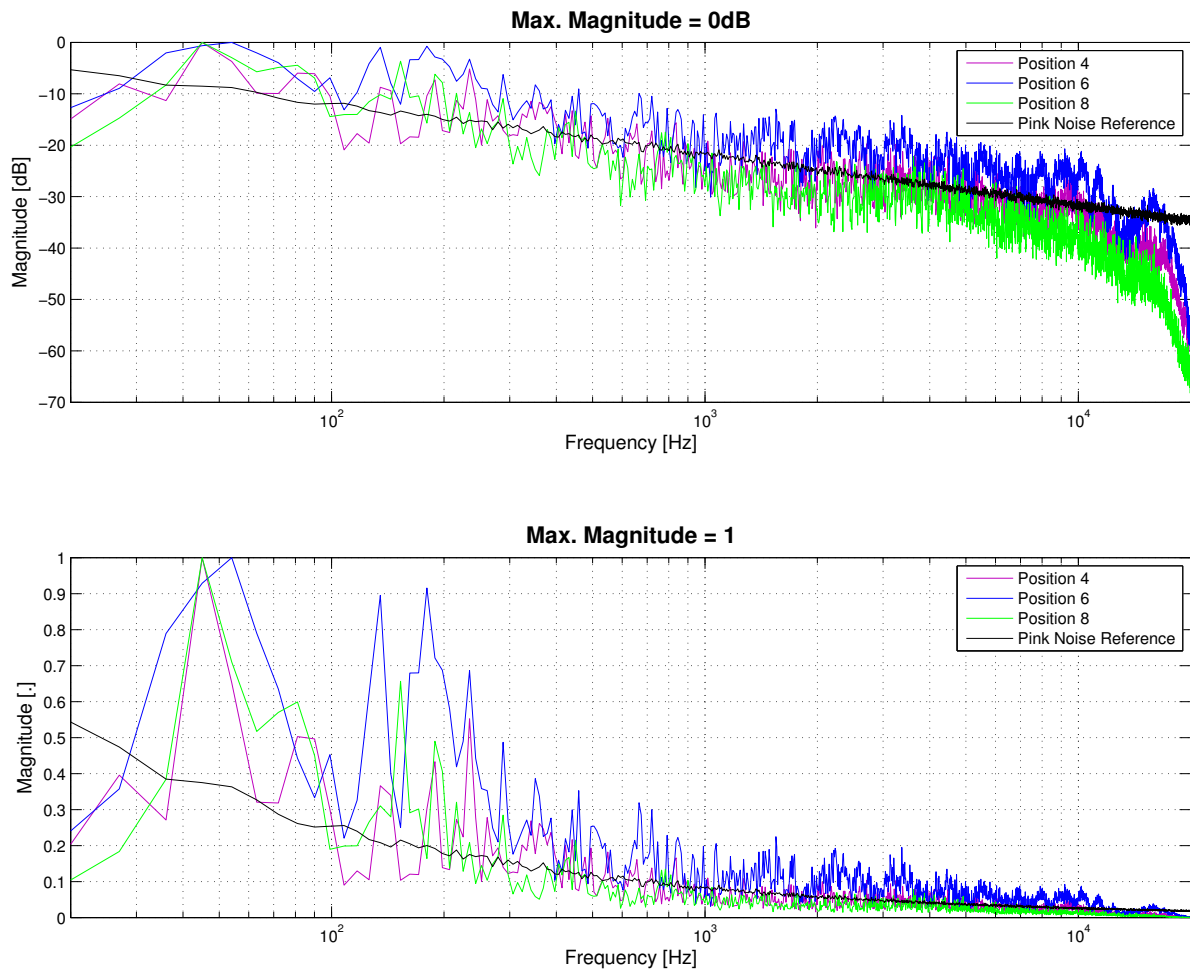


Figure 2.2: Frequency characteristics in three positions of a room, compared to the reference signal, which is pink noise signal.

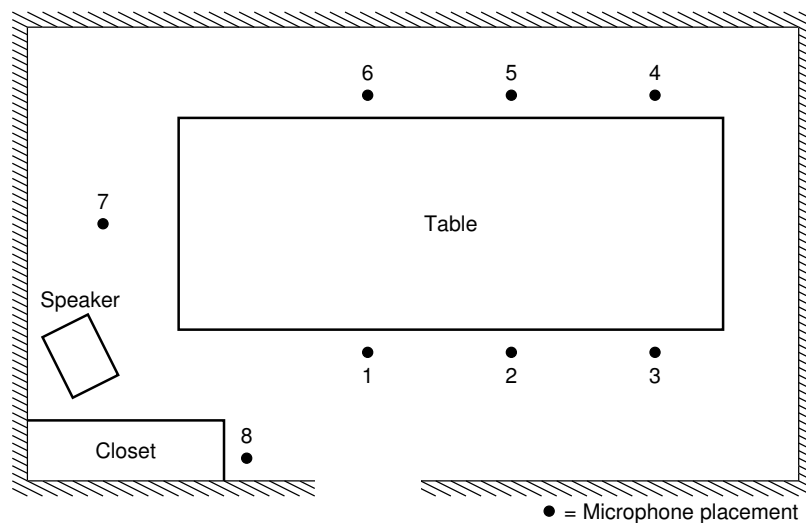


Figure 2.3: Positions of the speaker and the positions of the microphone, during recording of pink noise.

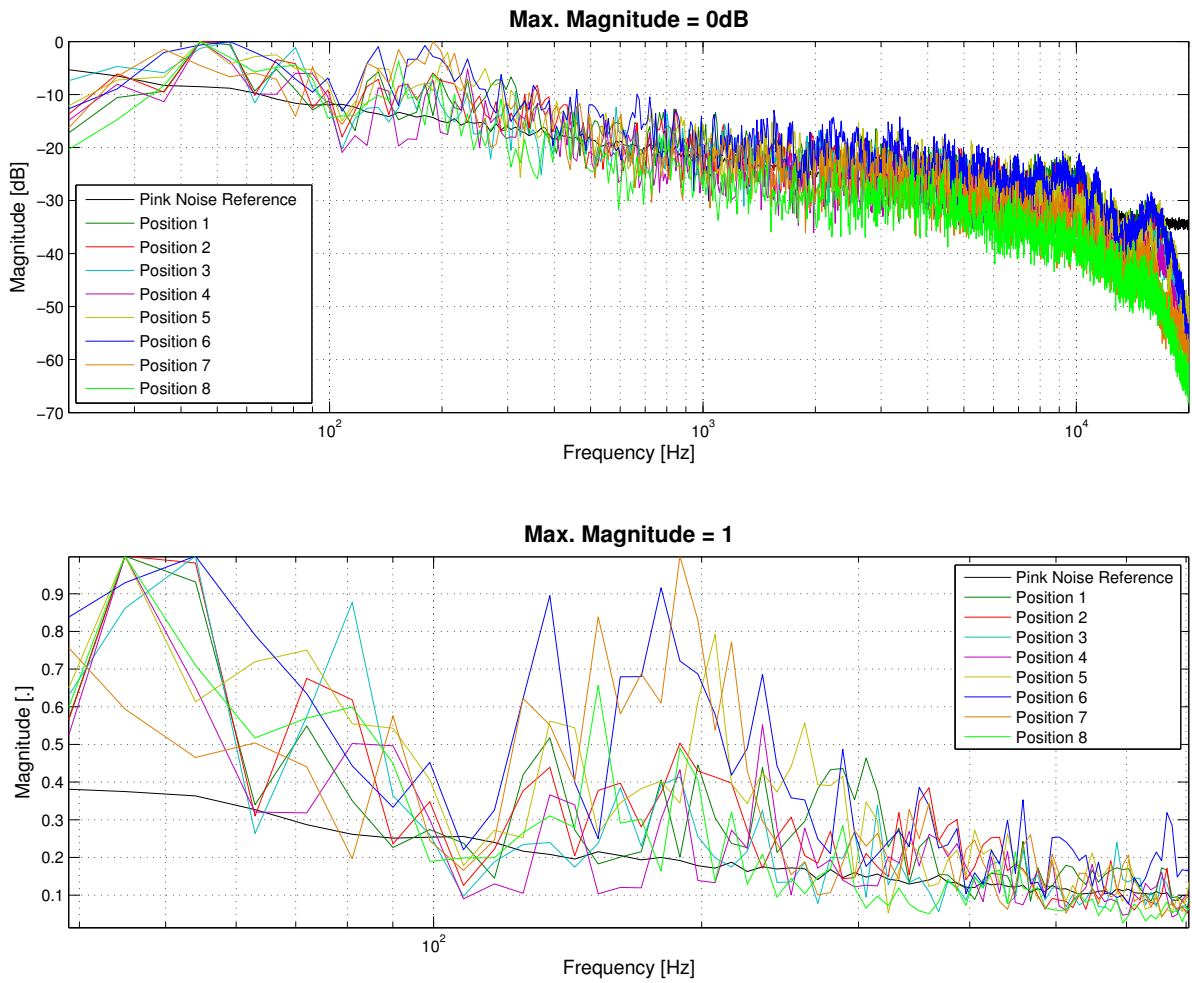


Figure 2.4: Frequency response of 8 measurements and the pink noise reference, in the low end of the spectrum.

2.3 Microphone System

The Microphone system (Mobile Station) is placed on the listener. It will communicate to the Base Station, through the wireless transceiver. Therefore the signal output has to be compatible with the input requirements for the ADC of the WL-transceiver. The expected SPL is below the human threshold of pain and will be above normal background noise of 60 dB SPL. To produce an exact representation of the reference signal, the frequency response is supposed to be linear in the audible range. The microphone device needs to be mounted on a listener, that moves around in a living room. Therefore it is preferred to use a small and non power consuming microphone. Furthermore the directivity pattern, should somehow be approximately close to that of the listeners ears.

Different microphone technologies exist, which are being used for different applications. Moving coil, ribbon, piezo, and the capacitor are some examples. Some more theory is explained in Appendix C.2 and Appendix C.3.

2.4 Standards for Dividing the Audible Frequency Range into Bands

When detailed information about a complex sound is required, the frequency range from 20 Hz–20 kHz can be divided into bands. This is done with band pass filters. In professional audio equipment, the bandwidth of each band is mostly designed as one third octave or one full octave [18]. In the following section the necessary equations in standard design will be described shortly, only for base 2 numbers, derived from DS/EN 21260. The standard DS/EN 61260[25] defines how the audible frequency spectrum should be divided into bands of selectable bandwidth.

The exact midband frequency f_m is defined as in Equation 2.1.

$$f_m = 2^{\frac{x}{b}} f_r \quad (2.1)$$

where

- f_r is the reference frequency, and is supposed to be 1 kHz
- x is an positive or negative integer, indexing the number of bands away from the reference frequency.
- b is the bandwidth-designator (for one third octave b is 3).

When the bandwidth-designator b is even:

$$f_m = 2^{\frac{2x+1}{2b}} f_r \quad (2.2)$$

The edge frequencies of the pass band are defined as in Equation 2.3

$$f_{\text{lower}} = 2^{\frac{-1}{2b}} f_m \quad \wedge \quad f_{\text{upper}} = 2^{\frac{1}{2b}} f_m \quad (2.3)$$

where

- f_{lower} is the lower band edge frequency of the respective band.
- f_{upper} is the upper band edge frequency of the respective band.

The transition limits is given in tables of relative attenuation.

The tables has been left out, but the data from the tables is shown graphical in Figure 2.5. The filter

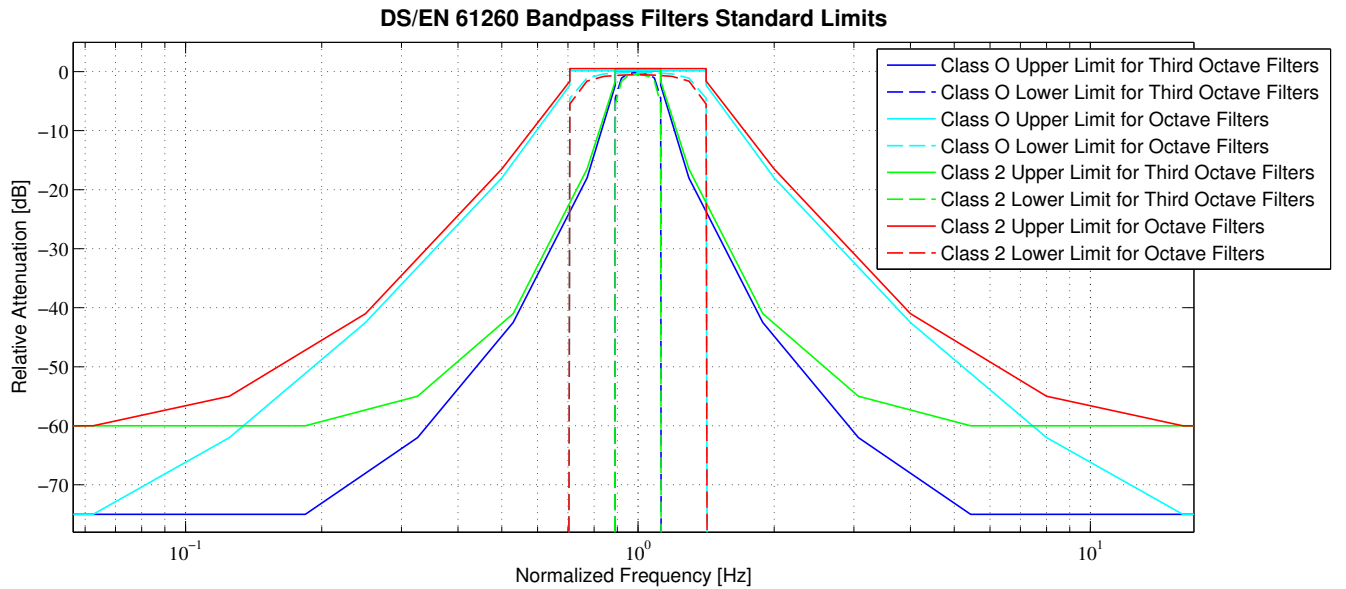


Figure 2.5: DS/EN 61260 band pass standart filters

designs are categorized into three classes. Only the two extreme classes are shown in Figure 2.5. They are shown for both the third octave and one octave designs, at normalized frequency. It can be seen that some ripple is allowed in the pass band. And importantly one can notice by looking at the transition, that the slope seems to be close to 60 dB per decade.

2.5 Filter Analysis

This section will investigate which filter to design, by complying to the DS/EN 61260 Standard. It will be easy to decide the Butterworth type, since it has the property of zero ripple in the pass band, which

is preferred in audio application. Therefore, by using MATLAB, three designs will be compared to the DS/EN 61260, Class 0. It has been decided from the data in Figure 2.5, to design LP-prototypes of 2nd, 3rd and 4th orders, and transform these into band pass filters, to see how they comply with the standard. The results are shown in Figure 2.7, 2.8 and 2.9.

For this approach to be used, the filters have to be parallel and summed together, as seen on Figure 2.6. This way, all the filters can be developed individually and each with it's own gain coefficient.

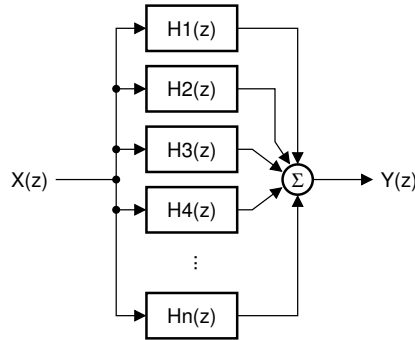


Figure 2.6: Filter construction. All filters are running in parallel and are summed together.

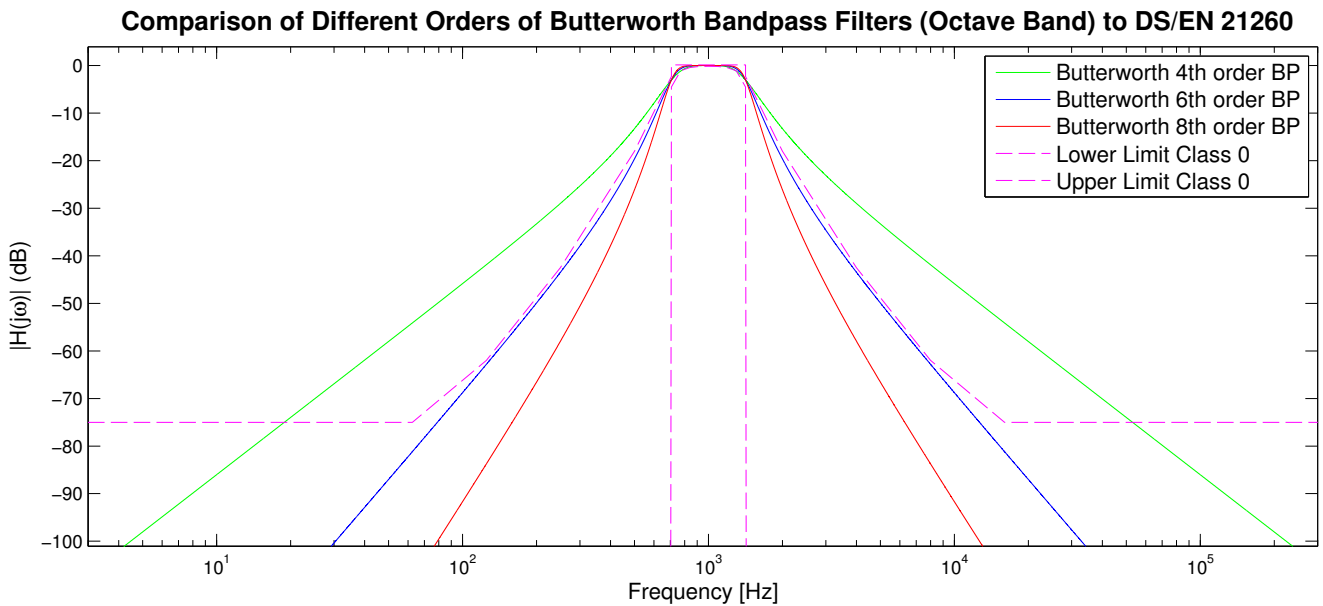
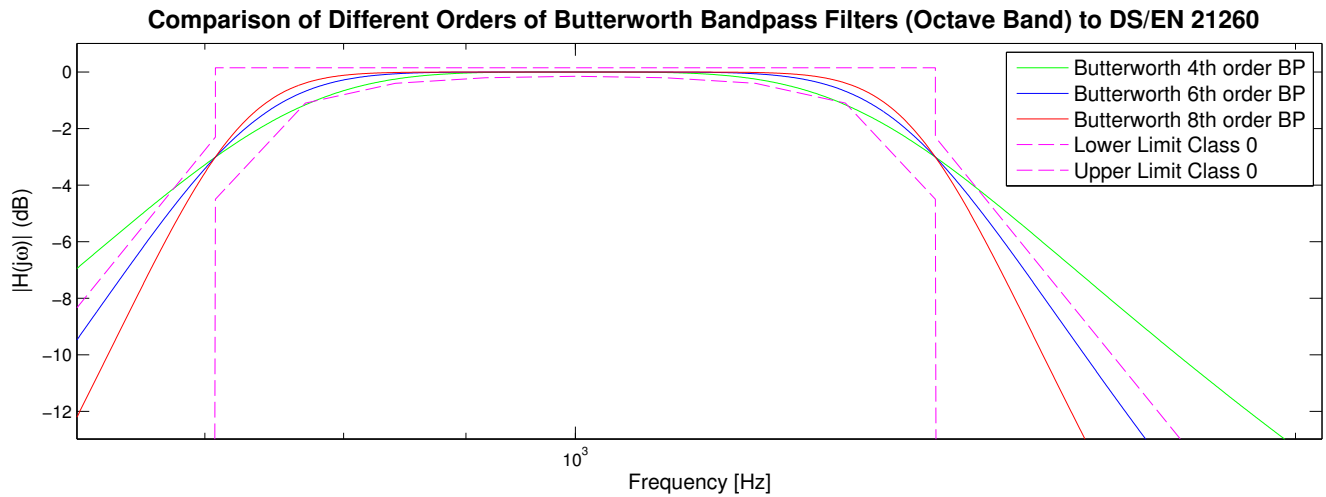
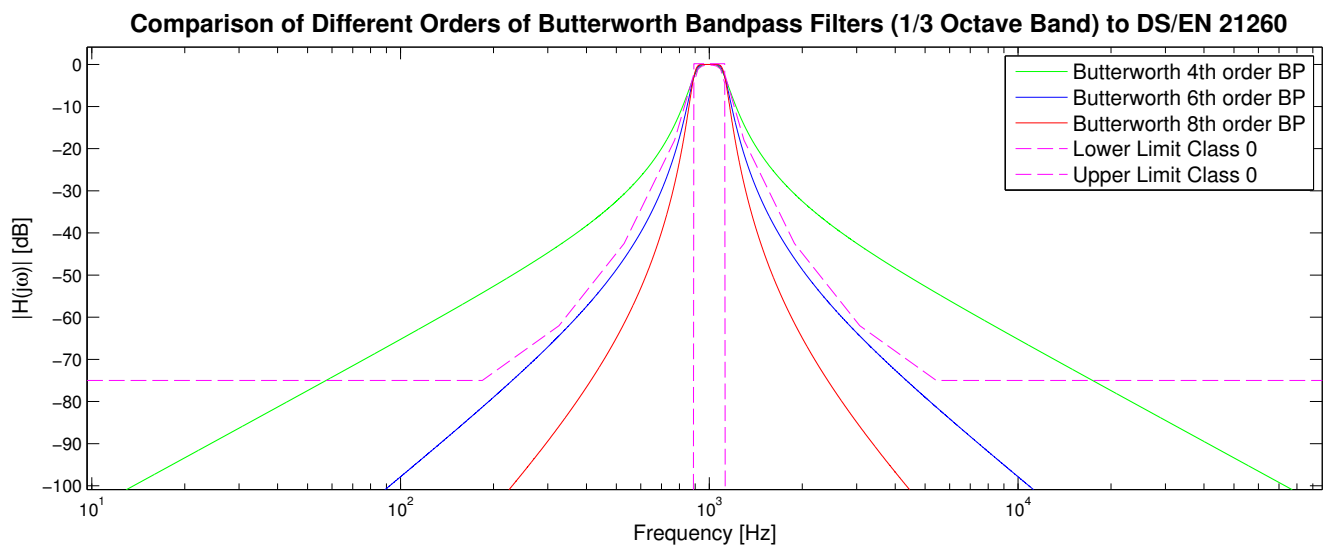


Figure 2.7: Comparison of different orders of Butterworth band pass filters to the DS/EN 61260 standard

It is seen that the 4th order BP-filter is non-compliant with the standard, while both the 6th and the 8th order filters are. This makes it possible to design a compliant filter, with a BP-transformation of the 3rd order LP-filter. Therefore the properties of this type is furthermore examined in conjunction with the frequency spectrum of the recordings of pink noise signal from Section 2.2. This is carried out by using the design procedure, described in Section 2.4 to design the 6th order BP-filter in both full octave design and third octave design, and graph them in the audible range on top of the plot of the measurements of recorded frequency response. For frequencies up to 1 kHz, see Figure 2.10.

This is done to extract a more precise link between the measured interference and the ability to equalize the spectrum, with a 6th order Butterworth design.

The results has to be compared to some test or calculation results from the the computer platform, and definitions from the limitations in the realtime system that handles the signal processing, before it is appropriate to settle on a design for the filter part.



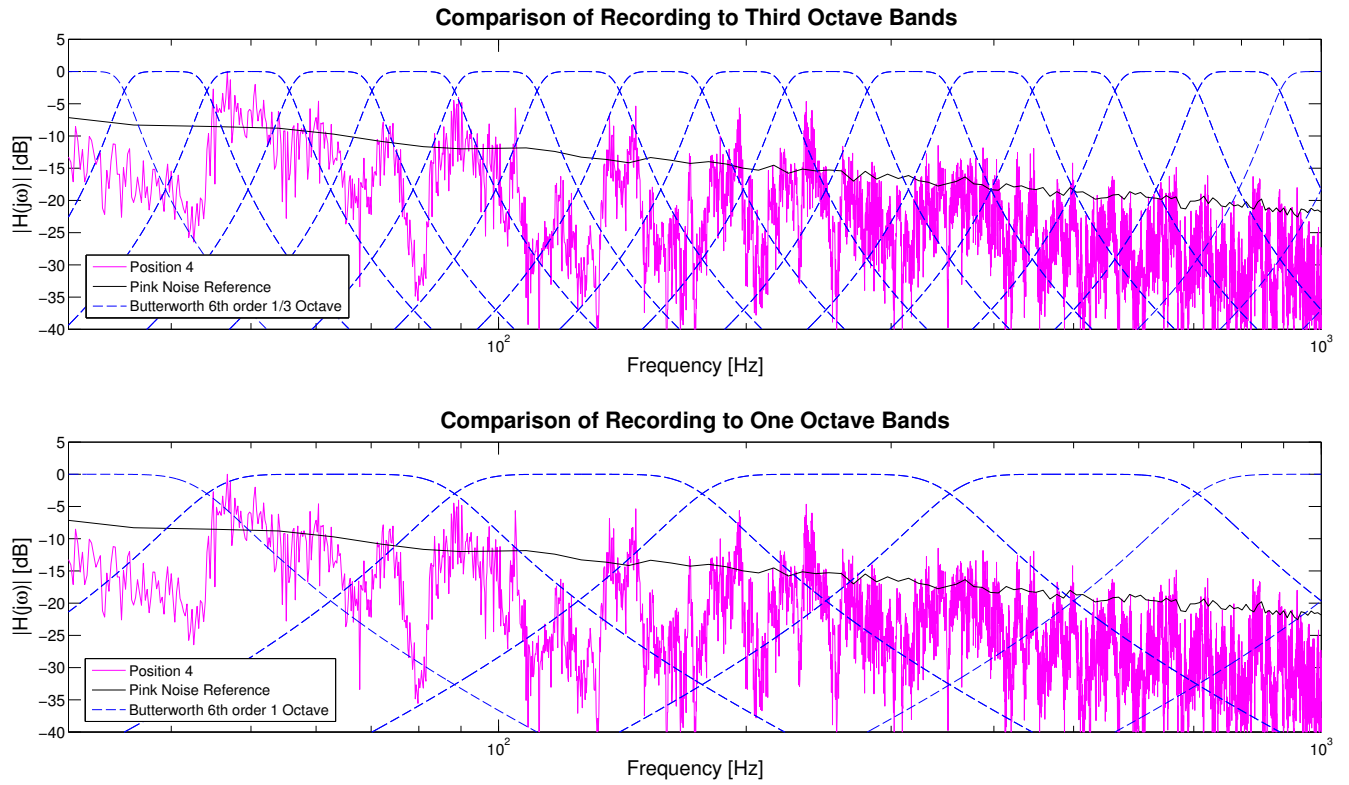


Figure 2.10: Comparison of noise recordings to filters of third octave and full octave. The frequency axis is represented in the lower end, and the recording is in full sample rate of 44,1 kHz

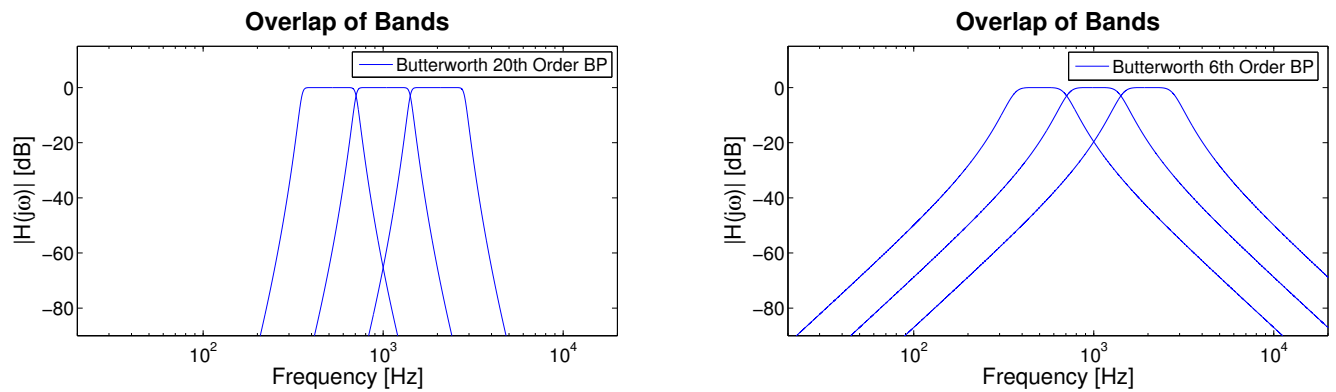


Figure 2.11: Overlap of 20th order compared to 6th order band pass design

2.6 Computer Platform

For the base unit of the system, it has been decided to use a Raspberry Pi. The reason for this is that it contains quite a lot of computation power and is widely supported. In this section, the features of the Raspberry Pi are briefly described in the context of the project.

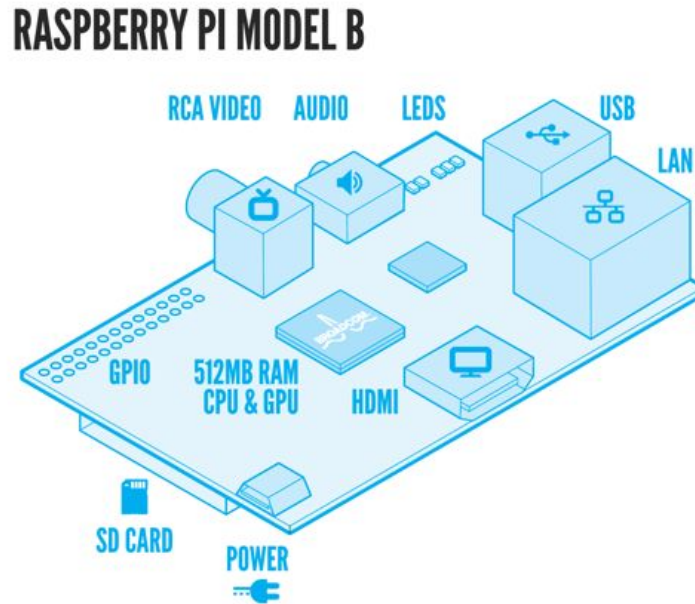


Figure 2.12: Raspberry pi model B.[21]

The initial idea behind the Raspberry Pi was to design a small and cheap computer for kids. This has resulted in a small computer based on a Broadcom BCM2835 ARM chip running 700 MHz. It is mainly build like a desktop computer containing USB ports, video output, LAN connector, and audio output as seen in Figure 2.12. Apart from these standard connectors, the unit contains a set of general purpose I/O ports (GPIO) which can be used to connect your own devices to it without using USB or other standard connectors.

The Raspberry Pi is build to use Linux as operating system. Debian is the recommended distribution, but every Linux distribution compatible with it's ARM processor is usable. Other operating systems like FreeBSD and Plan9 are also supported.[8][2]

As mentioned above, the Raspberry Pi contains a standard audio output, but by default does have an audio input. To input audio, the official website suggests using a USB microphone.[21]

2.6.1 Audio Inputs

Though it is desired to input an analog audio reference, the Raspberry Pi does not contain a standard analog input for audio. For this purpose the GPIO can be used. This requires an appropriate ADC for audio purposes. The optimal solution will also be able to handle the input conversion for the signal from the microphone. The standards for audio inputs from DIN45500/IEC61938-1 are listed in Table 2.1. The expected audio music input anno 2013 has been digitally produced to the radio. This product will

Source	Z_{out}	Required Z_{in}	Nominal v_{out}	Maximal v_{out}	Maximal v_{out}
Line	2,2 k Ω	≥ 22 k Ω	0,5 V _{RMS}	2,8 V _{RMS}	4 V _{peak}
CD	2,2 k Ω	≥ 22 k Ω	2,0 V _{RMS}	2,0 V _{RMS}	2,8 V _{peak}

Table 2.1: Signal standards for connecting to external audio player

most likely, have a nominal signal amplitude close to the max value, which should be close to $2,8 V_{\text{peak}}$. Furthermore the connectivity to an external audio player is often handled by 3,5 mm TRS phone connector on e.g. a mobile phone, which is obvious to consider when designing an external input to the Base Station.

2.7 Propagation Delay

The system needs to retain data in some internal structure, so it can be retrieved and replenished readily. To be able to design an efficient signal flow and buffer size, the signal propagations are compared in Figure 2.13. Figure 2.13 shows the symbolic data flow, independent of whether the signal is analog or digital. The important aspect is that the content of two buffers needs to be aligned in time. Initially when the

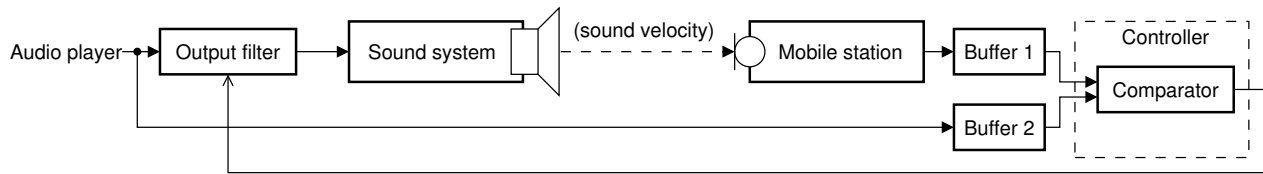


Figure 2.13: Flow diagram to show the delay in air propagation, relative to the wired signals.

audio signal has passed the filter, it will be output to the speaker and also to the data buffers for the controller. Meantime the data, that corresponds to the initial point in time, will reach buffer two first, and some delayed time t_d later it will reach buffer one. This time relation is described in Eq. 2.4. It is relative to the distance from microphone to speaker, due to sound velocity, which is described in Eq. C.4.

$$t_d = \frac{d}{v_{\text{air}}} \quad (2.4)$$

where

- t_d is delay time.
- d is the distance from speaker to microphone.
- v_{air} is the velocity in the air, approximately 340 m/s.

This situation could be fixed at system startup, by adding a test tone, with properties to make the buffers able to respond with a counter result. This situation could also vary the size of each buffer. The difference in buffer size $\text{Buffer 2} - \text{Buffer 1} = \Delta\text{buf}$ will then be.

$$\Delta\text{buf} = t_d \times \text{bitrate} \quad (2.5)$$

2.8 Size of the Audio Buffer

To process the audio signal, the Base Station will contain the audio in a buffer system. There are some limitations in this aspect, which is the interval in which it is possible to sample without aliasing. When sampling it is necessary to define the lowest sampling frequency, f_c . The lower limit has to be compliant with the Nyquist-Shannon Sampling theorem. Next the minimum audio buffer size must be defined. The limit can be defined from the lowest input frequency which is desired to sample, and in this respect it's important that the buffer is able to contain the signal (e.g. 20 Hz). If the lowest frequency is 20 Hz then the period time is $T_s = \frac{1}{20 \text{ Hz}} = 50 \text{ ms}$. The minimum buffer size $\geq x \cdot \frac{\text{samples}}{T_s}$, where x is the maximum data size of each sample. If the sampling frequency f_c is 44,1 kHz there will be $44,1 \text{ kHz} \cdot 0,05 \text{ s} = 2205$ samples, if each sample is 4 B then the minimum buffer size $\geq 8,82 \text{ kB}$.

2.9 Wireless Communication

When designing electronic devices it is often useful to utilize wireless communication. A lot of electronic systems have wireless communication implemented, whether it is for a car, mobile phone, computer, or television. These implementations comes in many forms, with different protocols. In this project the wireless communication will be used between the mobile and Base Station, so that the user can change position, freely, inside a room.

2.9.1 Bandwidth

Wireless communication in electronic devices are usually using the radio spectrum between 30 MHz and 30 GHz [1]. The spectrum is allocated into licensed and unlicensed bands, whereas the licensed bands could be for AM radio, FM radio, or the 3G. Using licensed bands without being licensed is strictly illegal, and can result in large fines, therefore it is important to choose a band that is public available worldwide.

The word bandwidth is rather ambiguous, since it can be the difference between the upper and lower frequencies (spectrum) or it can be the network throughput. In this report bandwidth will refer to the spectrum and not network throughput.

2.9.2 Needed throughput

Since the Mobile Station needs to stream audio wireless, which is in the range of 20 Hz to 20 kHz, it needs a certain throughput to be able to stream this. Because of Nyquist–Shannon sampling theorem[19] it's obvious that the sample rate should be equal or greater than 40 kHz. Regular CD's uses a sample rate of 44 100 Hz and usually a bit depth of 16 bit.[9] The bitrate can be calculated as [26].

$$\text{bitrate} = \text{sample rate} \times \text{bit depth} \times \text{channels} \quad (2.6)$$

The theoretically needed throughput to stream audio in CD quality would then be

$$\begin{aligned} \text{bitrate} &= 44\,100 \text{ Hz} \cdot 16 \text{ bit} \cdot 2 \\ \text{bitrate} &= 1\,411\,200 \text{ b/s} \approx 1,4 \text{ Mb/s} \end{aligned}$$

This is the needed throughput, and it might not be necessary to have the audio in CD quality. Lossy compression, can be used to compress the audio using an algorithm and then save throughput. Such algorithms can be implemented as hardware solutions. By using a hardware solution large computational delays can be avoided.

2.9.3 Range

The range limitation of wireless communication depends on the specific technology used and in this section the specific technology will take base in the IEEE 802.11 standard. The IEEE 802.11 standard is communicating in the 2,4 GHz, 3,6 GHz, 5 GHz, and 60 GHz frequency bands and depending on the protocol, the indoor communication range can reach up to 70 m.

Typical wireless connections using the IEEE 802.11 protocols such as wifi and Bluetooth communication can reach an indoor range of up to 35 m. Wifi uses the same frequency band as Bluetooth at 2,4 GHz, but uses higher power which results in a higher bit rate and better range from the Base Station. Both the wifi and Bluetooth technology are developed for home and outdoor use, which means that both technologies are suitable for the communicating aspect of this project.

The range is also dependent on the frequency, as the wavelength changes when the frequency changes, also described in the speaker section, Appendix C.1.[27]

2.9.4 Digital vs. Analog

When looking at different wireless technologies, it's important to take a look at the difference between analog and digital technologies. The sound quality for digital signals compared to analog signals depends on the resolution (bits per sample) of a given sound. The higher resolution the more accurate reproduction of the sound. This means in theory that analog sound quality would be better because of the reproduction of the sound, but nowadays the resolution of digital systems are very high and the difference is not audible. When talking about signal noise, the digital technology is the winner. In digital the signal only consists of zero's and one's, which means that noise only gets noticed if it passes the level where a digital zero turns into a digital one. In the case of analog technology, noise can be heard at any level because of the analog signal [24].

The price of digital technologies compared to analog technologies varies a lot. Analog technologies have been the cheapest solution for many years, but the digital technologies are beginning to catch up.[7]

3 Problem Statement

Throughout the problem analysis the initiating problem have been answered in Part I. The collected knowledge is used to setup criteria for developing an appropriate solution, and describe what requirements should be met.

How is a self adjusting loudspeaker system designed, which is able to counteract the effect of the frequency characteristics of reflections, by equalizing the signal to a speaker, so that the sound in the air is similar to the audio source—even when the listener is moving around the room?

4 Requirement Specification

To be able to answer the Problem Statement in Chapter 3, the requirements to the product must be described. The technical part of this report will be based upon this chapter.

Reading Instructions The requirements in this chapter are derived from the analysis. The requirements are split in four groups:

1. Section 4.1 is a general description of the system.
2. Section 4.2 describes the system functionalities.
3. Section 4.3 describes the system's interfaces.
4. Section 4.4 describes the specific requirements that the product has.

4.1 General Description

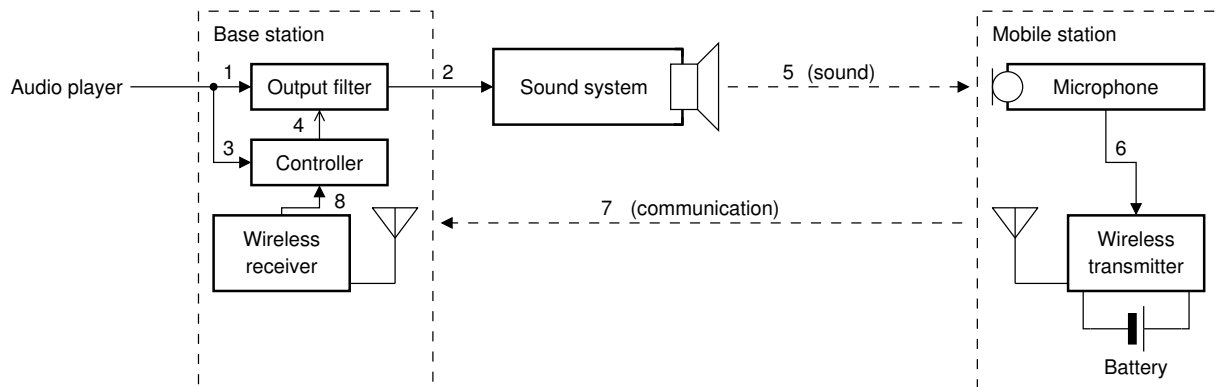


Figure 4.1: System block diagram

The overall system consists of a Base Station and a Mobile Station as it can be seen on the block diagram in Figure 4.1. In order to get an overview of the functionality of the system, the interfaces and their directions are added to the block diagram. In the following there will be a description that outlines the general functionality of the system.

4.2 Functionality Description

From the block diagram in Figure 4.1 the following functionality will be described. It will describe the main functions of the system and also give an indication of the main system interfaces.

At first, the Base Station receives a signal from the audio player. This signal is used as a reference input to the controller and as input to an adjustable output filter. When the input has passed the output filter it is send to the sound system which outputs sound to the room. The Mobile Station which is placed on the listening person, uses a microphone to detect the sound in the room. The sound is recorded by the microphone and send through a wireless communication system to the Base Station station. At the Base Station the received signal is used as an input to the controller. The controller compares the recorded signal to the reference signal from the audio player. The purpose of the controller is to compensate any changes between the audio player input and the recorded signal, so that the listener always gets the same experience. The controller will do this by applying information to the output filter, which changes the characteristics of the filter.

4.3 Description of Interfaces

From the block diagram Figure 4.1, it is possible to determine the interfaces, which will be described in the following section. On the block diagram Figure 4.1 the interfaces are enumerated corresponding to the numbers below.

1. **Audio Player and Filter** An external music player inputs a analog line signal of max $2,8 V_{RMS}$.
2. **Between Filter and Sound System** The filter outputs a standard analog line signal of max $2,8 V_{RMS}$.
3. **Audio Player and Controller** An external music player inputs an analog line signal of max $2,8 V_{RMS}$. The controller must convert this signal to digital data of same format as the output from WL receiver to Controller.
4. **Between Controller and Filter** The controller knows how to manipulate the output filter and sends this correction data to the filter.
5. **Sound System and Microphone** The speaker outputs a certain sound pressure level of max $120 dB_{SPL}$. The background noise will be minimum $60 dB_{SPL}$.
6. **Microphone and WL-transmitter** The microphone will convert sound pressure variations to voltage cf. Section C.2. The output voltage is maximum $2,8 V_{RMS}$.
7. **WL-transmitter and WL-reciever** The wireless transmitter and wireless receiver communicates by the use of modulated radio waves.
8. **WL-reciever and Controller** The digital data will be passed directly into the controller as an analog signal of maximum $2,8 V_{RMS}$

These are the main interfaces. These blocks will be further divided into smaller blocks in the development process, described in Chapter II.

4.4 Specific Requirements

Below is a list of the specific requirements for the system. It defines the requirements for the blocks in the block diagram in Figure 4.1. The requirements are numbered so they can be referred to later.

4.4.1 Base Station

1. The Base Station must have a line input for a sound system, compliant with the standard in Table 2.1.
2. The Base Station must have a line output to a sound system, compliant with the standard in Table 2.1.

4.4.2 Filter

3. Sample rate of audio input (for audio throughput): $> 40 \text{ kHz}$.
4. Bandwidth of regulation: $40 \text{ Hz} - 1 \text{ kHz}$ (since the largest changes are in this region, see Figure 2.2).
5. The frequency range should be covered by $1/3$ octave bands (see Figure 2.10).
6. The band pass filters are minimum of Class 2 (see Figure 2.5).
7. The band pass filters' midband frequencies are calculated from a reference of 1 kHz and should be located as given by Equation 2.1 and Equation 2.2 $\pm 2 \%$.
8. The edge frequencies are proportional to the center frequencies and should be located as described in Equation 2.3 $\pm 2 \%$.
9. Maximum ripple within the frequency band: $0,5 \text{ dB}$.

4.4.3 Controller

These requirements are based on measurements in Figure 2.4.

10. Sample rate of the audio player and microphone (for regulation): $> 2 \text{ kHz}$.

11. A signal attenuated more than 15 dB at a frequency must not be boosted more than 15 dB to avoid amplification of “nothing”.
12. Must be able to attenuate boosted frequencies > 20 dB.
13. Must be able to adjust filters in < 500 ms.

4.4.4 Mobile Station (General)

14. Maximum physical size: $100 \text{ mm} \times 50 \text{ mm} \times 15 \text{ mm}$.
15. Maximum weight: 200 g corresponding to the weight of a modern smartphone[17].

4.4.5 Wireless Transceiver

The following requirements are based on the problem analysis Section 2.9.

16. Use a worldwide public available frequency band.
17. Bitrate $> 1,4 \text{ Mbit/s}$ corresponding to Equation 2.6.
18. Transceiver range $> 10 \text{ m}$.
19. Power consumption in standby $< 50 \text{ mW}$.

4.4.6 Microphone System

The microphone system contains an amplifying part. Thus, these requirements are including this.

20. Sensitivity value of $2,8 V_{\text{RMS}}$ at $120 \text{ dB}_{\text{SPL}}$ [11].
21. A linear output signal in the range $40 \text{ Hz} - 1 \text{ kHz} \pm 0,5 \text{ dB}$ is required out of the microphone amplifier.
22. $\text{SNR} \geq 50 \text{ dB}$.
23. $\text{THD} \leq 0,5 \%$ in the operating range.
24. Operation voltage: $3,3 - 10 \text{ V}$.
25. Current consumption: $< 10 \text{ mA}$.
26. Output impedance: Less than 10 times the input impedance of the sound card input.
27. Input impedance: Greater than 10 times the output impedance of the WL-transmitter.

Now the requirements for the system has been defined from the analysis, therefore the next chapter defines how to test these requirements.

5 Test Specification

5.1 General Tests

Test of Requirement 1—Line Input

This requirement is tested using an oscilloscope. The power will be set to max and it is possible to read the peak values on the oscilloscope and then compare them to the values set in Table 2.1.

The input impedance is tested according to the method used in the testspecification for Requirement 27.

Test of Requirement 2—Line Output

This requirement is tested using an oscilloscope. The power will be set to max and it is possible to read the peak values on the oscilloscope and then compare them to the values set in Table 2.1.

The output impedance is tested according to the method used in the testspecification for Requirement 26.

5.2 Filter

Test of Requirement 3—Sample Rate of Audio Input

The sound card is set to a sample frequency of 44,1 kHz and the processor is running at 700 MHz. As the sampling rate is defined in the main program there is no need to test this requirement.

Test of Requirement 4—Filter Bandwidth

In the design, the octave bands are placed in the desired frequency region. Therefore, this requirement is not tested.

Test of Requirement 5—Number of Bands

The filter is designed around a certain amount of bands. Therefore, this requirement is not tested.

Test of Requirement 6—Filter Class

Measurements will be done on a frequency analyzer. The measurements will be plotted with the upper and lower limits of the Class 2 filter. If the requirements are met, the measured signal will be in between these two limits.

Test of Requirement 7—Midband Frequencies

Frequency analysis measurements will show a peak with a certain bandwidth for each filter. The center of this band is calculated as

$$f_c = \sqrt{f_{\text{lower}} \cdot f_{\text{upper}}} \quad (5.1)$$

where

- f_{lower} is the lower edge frequency at -3 dB.
- f_{upper} is the upper edge frequency at -3 dB.

Test of Requirement 8—Edge Frequencies

The edge frequencies, f_{lower} and f_{upper} , are measured using a frequency analyzer as described above.

Test of Requirement 9—Pass band Ripple

From the previously mentioned measurements, the pass band ripple is determined from the lowest filter's f_{lower} and the highest filter's f_{upper} . Within this band, the highest amplitude, A_{max} , and the lowest

amplitude, $A_{\min}$ is determined in decibels. The pass band ripple is then

$$R_{\text{dB}} = A_{\max} - A_{\min} \quad (5.2)$$

5.3 Controller

Test of Requirement 10—Audio Player Sample Rate

The sound card is set to a sample frequency of 44,1 kHz and the processor is running at 700 MHz. As the sampling rate is defined in the main program there is no need to test this requirement.

Test of Requirement 11—Maximum Boost at Frequencies

A measurement like that in Section 2.2 is carried out with the Base Station playing—without the controller activated—to show areas where the difference with regards to the reference signal is larger than that given of the requirement.

The controller is then activated and should at most boost dampened frequencies the amount set in the requirement.

Test of Requirement 12—Minimum Dampening at Frequencies

The minimum dampening should be visible on the previous test of Requirement 11.

Test of Requirement 13—Reaction Time

The reaction time is tested by adding a reference signal that is pink noise. A noise source is applied, consisting of an abrupt switch from a 440 Hz to a 619 Hz sine (triton interval—very disharmonic so no overtones should be an issue). This should cause the filters to change because as 440 Hz is in the 397 Hz-filter range and 619 Hz is in the 630 Hz-filter range, according to Requirement 7 and 8.

Two signals are recorded:

1. The output (going to the loudspeaker).
2. The microphone recording.

and a spectrogram of each signal is plotted. The output signal should show the sine shift on top of the pink noise and the microphone should show the noise with the sine shift subtracted. If the sine shift in the output happens at time t_{out} and the sine shift in the microphone happens at time t_{mic} then the reaction time is

$$t_{\text{reaction}} = t_{\text{mic}} - t_{\text{out}} \quad (5.3)$$

5.4 Mobile Station (general)

Test of Requirement 14—Maximum Physical Size

The size of the Mobile Station is measured by a measuring tape and compared to the requirement.

Test of Requirement 15—Maximum Weight

The Mobile Station is weighed on a weighing scale and compared to the requirement.

5.5 Wireless

Test of Requirement 16—Frequency band

The unit is designed around this requirement. Therefore, this is not tested.

Test of Requirement 17—Bitrate

A file of a known size is sent from transmitter to receiver while measuring the time it takes to arrive. The bitrate is then calculated as

$$\text{bitrate} = \frac{\text{size of file in bits}}{\text{time it takes to arrive in seconds}} \quad (5.4)$$

Test of Requirement 18—Transceiver Range

Transceiver range is tested in two steps.

1. Transmit. The Base Station is put stationary. The Mobile Station is set to transmit data to the Base Station and is moved away until the wrong data arrives at the Base Station. The distance is measured.
2. Receive. The Base Station is put stationary. The Base Station is set to transmit data for the Mobile Station to receive. The Mobile Station is moved away until wrong data is received. The distance is measured.

Test of Requirement 19—Power Consumption in Standby

The power supply for the wireless unit is run through an ampere meter. The wireless unit is put in standby mode and the current consumption is measured.

5.6 Microphone System

In this chapter is it defined how to test these requirements. Every requirements has a specific description, to make sure the requirements are testable.

Test of Requirement 20—Microphone Sensitivity

Microphone sensitivity is tested by putting the microphone next to a sound pressure meter—both of which are placed in front of a speaker. The speaker plays a 1 kHz sine wave, and is turned up until the sound pressure meter shows the wanted sound pressure. The output voltage of the microphone system is now measured with a voltage meter.

Test of Requirement 21—Microphone Amplifier Frequency Response

The microphone amplifier frequency response is measured by a frequency analyzer. The frequency analyzer does a sweep of frequencies and measures the amplitude at each frequency as the signal is sent through the microphone amplifier. The output amplitude of the frequency analyzer is set to the maximum output amplitude of the microphone itself. The resulting frequency response can now be compared and should be within the limits given by the requirement inside the interval given by the requirement.

Test of Requirement 22—Signal to Noise Ratio

The system is turned up to full volume. The amplitude of the noise floor, V_{noise} , is measured. A 1 kHz sine wave is put on the input at full volume and it's amplitude, V_{signal} , is measured. The signal to noise ratio is then defined as

$$\text{SNR} = 20 \log \frac{V_{\text{signal}}}{V_{\text{noise}}} \quad (5.5)$$

Test of Requirement 23—Total Harmonic Distortion

The total harmonic distortion is measured by the frequency analyzer tool that is also used to test Requirement 21—Microphone Amplifier Frequency Response.

Test of Requirement 24—Operational Voltage

The amplifier is designed around a operational voltage that fits this requirement. Therefore, this need not be tested.

Test of Requirement 25—Current Consumption

The current consumption is measured by putting a ampere meter between the power supply and the microphone system, so that all current goes through the ampere meter. A sine wave is put on the input of the amplifier with the amplitude being the maximum output of the microphone. The current consumption is observed as the frequency is swept from 20 Hz to 20 kHz.

Test of Requirement 26—Output Impedance

The output impedance is measured by putting a sine wave on the input of the amplifier. The procedure for getting the output impedance is then as follows (see Figure 5.1).

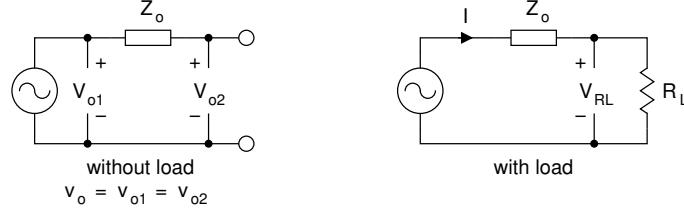


Figure 5.1: Procedure for measuring output impedance.

1. Without a load, the open circuit voltage V_o is measured. As no load is connected, V_o is the same on “both sides of the output impedance” as shown in Figure 5.1.
2. A load, R_L , is put on the output as shown in Figure 5.1. *The size of the R_L is chosen so small that a clear difference can be seen when the load is present compared to the open loop.*
3. The voltage drop, V_{RL} is measured. The current in the circuit can now be calculated in two ways, using the voltage drop over either R_L

$$I = \frac{V_{RL}}{R_L} \quad (5.6)$$

or the voltage drop over Z_o

$$I = \frac{V_o - V_{RL}}{Z_o} \quad (5.7)$$

From Equation 5.9 the current, I , can now be determined.

4. From Equation 5.10 and V_o previously obtained, the output impedance, Z_o , can be obtained

$$Z_o = \frac{V_o - V_{RL}}{I} \quad (5.8)$$

Test of Requirement 27—Input Impedance

The input impedance is measured by putting a sine wave on the input of the amplifier. The procedure for getting the input impedance is then as follows (see Figure 5.2).

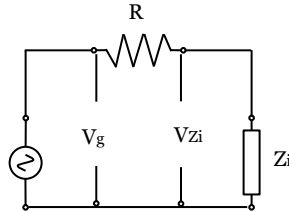


Figure 5.2: Procedure for measuring input impedance.

1. The voltage V_g measured before the known resistor as shown in Figure 5.2.
2. A known resistor, R , is put on the input as shown in Figure 5.2.
3. The voltage drop, V_{Zi} is measured. The current in the in the circuit can now be calculated, using the voltage drop over Z_i

$$I = \frac{V_g - V_{Zi}}{R} \quad (5.9)$$

The input impedance Z_i can be found as:

$$Z_i = \frac{V_{Zi}}{I} \quad (5.10)$$

Now that all the requirements has been defined whit a coplementary test specifacation, it is now time to design the subsystems. These will be designed in the next chapter.

II

Technical Solution

This Part will document the development of all the technical system parts, both for the Base Station and Mobile Station. The technical development will be initialized from the block diagram overview in Figure 5.3. A detailed description follows of the main program, the audio interface system, the applied filters and the wireless system. From the development of each subsystem follows a subsystem test. Lastly the implementation of the subsystems is done before the full system is implemented.

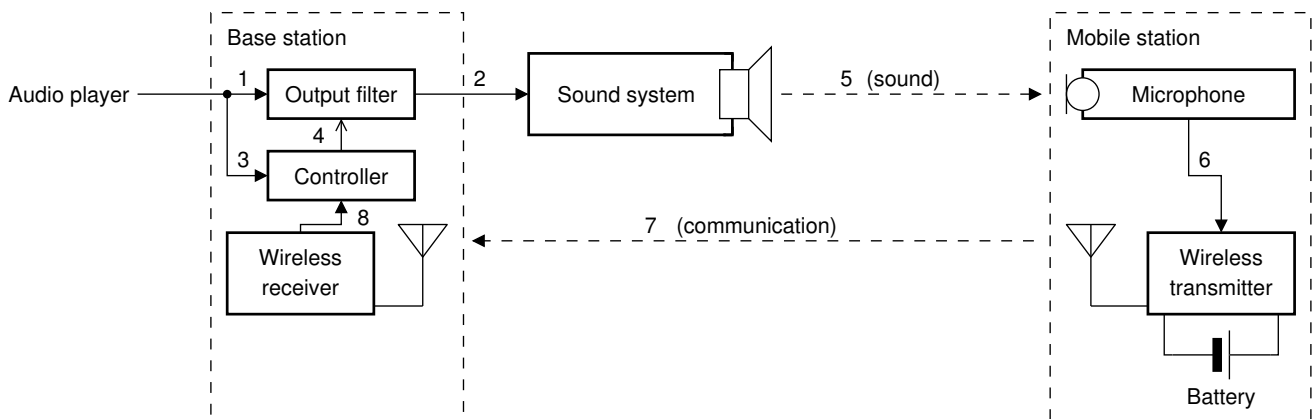


Figure 5.3: Block Diagram of the overall system.

From the block diagram in Figure 5.3 the system is well defined. It outlines all the main interfaces between every building block. It is then up to the developer to design and implement each building block, so that it fulfills the requirements specification. To give an overview of the overall code, this chapter begins with the documentation of the main program.

6 Main Program

In this chapter the development of the main program will be documented. The main program is the top level and controls the flow of all the developed functions.

As suggested in the block diagram in Figure 4.1 in the requirements specification, the playback through an output filter is done in parallel with the control calculations. This can be done on the Raspberry Pi by using a thread for playback and one for calculating the filter gains. This thread division can be seen on Figure 6.1. It should be noted, that one of the “threads” will be the main program that is then running as the main thread creating the other.

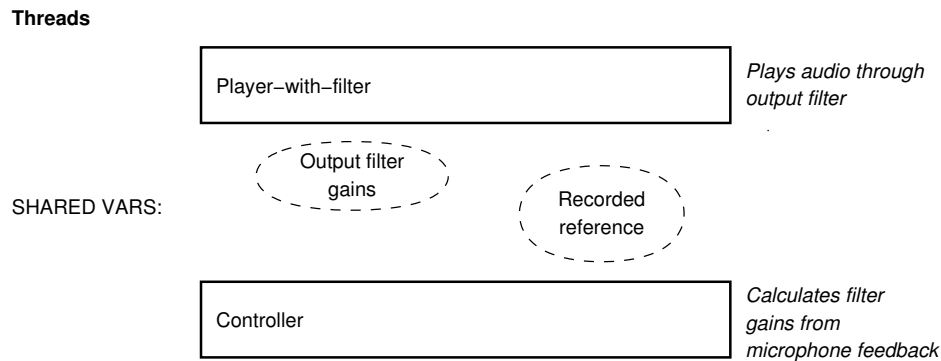


Figure 6.1: Thread division of the control, playback, and filtering process.

The thread has shared data, **SHARED VARS**.

The Player-with-filter thread is always playing back the reference signal through the compensating output filter. This has very high priority because there should not be any glitches in the sound played to the user. Since the reference signal should be used by both threads, the Player-with-filter thread does the recording and shares it with the other thread.

The Controller thread receives the microphone signal and from that and the reference signal it calculates the gain coefficient of the output filter so it makes the output signal closer to the reference.

The flow of the two threads can be seen in Figure 6.2 and 6.2.

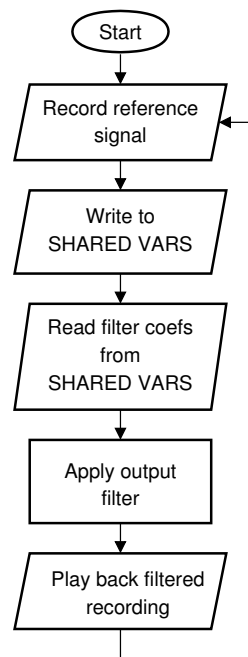


Figure 6.2: Flow of the Player-with-filter thread.

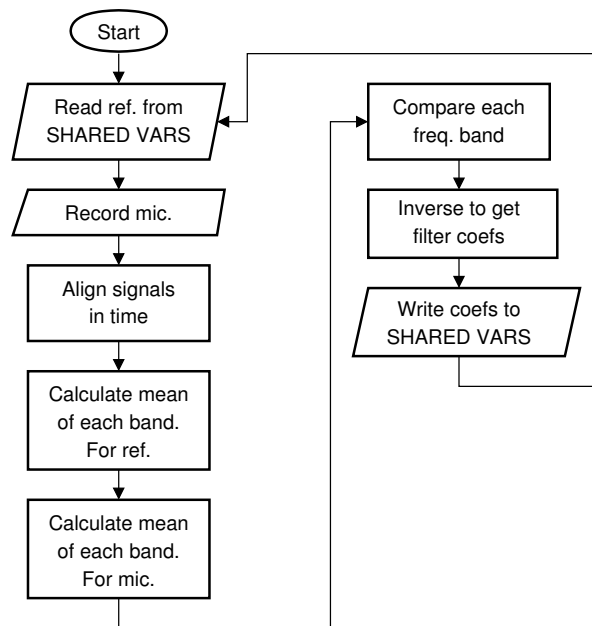


Figure 6.3: Flow of the Controller thread

6.1 Thread Test Code

To begin the development it has been decided to design the two multithreads as described in Section 6. The following code is initially designed directly from the diagram in Figure 4.1 and 6.1. The procedure is firstly to design the threads, with their corresponding semaphores. This will then be utilized for testing the data flow. This design is also used as the basis for implementing the required functionalities and data processing of the complete system. In the code there is implemented some test functions, that reads from a text file and applies a low pass filter calculations and RMS calculations. The two threads is designed with an ID number used for debugging the data flow of the threads and semaphores.

```

1  #include <stdio.h>
2  #include <pthread.h>

```

```

3      #include <semaphore.h>
4      #include <unistd.h>
5      #include <math.h>
6
7      #define BUF_SIZE 4410
8
9      /**
10     * Thread to playback audio and apply the output filter, by use of semaphor.
11     * @param input ID of thread used in test.
12     * @return void
13     */
14     void player_filter (void * input);
15     /**
16     * Thread to compare the mic- and ref-signals, and write result in semaphor.
17     * @param input ID of thread used in test.
18     * @return void
19     */
20     void controller (void * input);
21     /**
22     * Semaphor function to read data from text-file and put data in input_buffer.
23     * @return void.
24     */
25     void read_file_to_buffer ();
26     /**
27     * Function to calculate RMS value.
28     * @param ref_buffer pointer to data_buffer inside controller.
29     * @return float containing RMS value.
30     */
31     float calculate_RMS (float * data_buffer);
32     /**
33     * Function to apply lp-filter to data_buffer from controller.
34     * @param ref_buffer pointer to data_buffer inside controller.
35     * @return float containing .
36     */
37     void lp_filter (float * data_buffer);
38
39     sem_t SHARED_VARS;
40     float data[128], coefficients[13], input_buffer[BUF_SIZE];
41     int testcounter = 0;
42
43     int main ()
44     {
45         int id_number[2];
46         pthread_t t_id[2];
47
48         id_number[0] = 1;
49         id_number[1] = 2;
50         sem_init(&SHARED_VARS, 0, 1);
51         pthread_create(&t_id[0], NULL, (void *) &player_filter, (void *) &id_number[0]);
52         pthread_create(&t_id[1], NULL, (void *) &controller, (void *) &id_number[1]);
53
54         while (1) {
55
56             pthread_join(t_id[0], NULL);
57             pthread_join(t_id[1], NULL);
58         }
59         return 0;
60     }
61
62     void player_filter (void * input) {
63         while (1) {
64             int id;
65             id = *((int *) input);
66
67             // input_buffer[n] = record_reference (fs, n);
68
69             printf("Thread %d: Waiting to enter critical region of input_buffer...\n", id);
70             sem_wait (&SHARED_VARS);
71             testcounter++;
72             printf("Thread %d: write_input_buffer();", testcounter);
73             printf("%d\n\n", id, testcounter);
74             read_file_to_buffer ();

```

```

75         sem_post (&SHARED_VARS);
76
77         printf("Thread %d: Waiting to enter critical region of coefficients...\n", id);
78         sem_wait (&SHARED_VARS);
79         testcounter++;
80         printf("Thread %d: read_new_filter_coefficients();                testcounter = %d
           \n\n", id, testcounter);
81         sem_post (&SHARED_VARS);
82
83         // apply_output_filter ( coeffiecents );
84     }
85 }
86
87 void controller (void * input) {
88     while (1) {
89         int id;
90         id = *((int *) input);
91         float ref_buffer[BUF_SIZE];
92
93         printf("Thread %d: Waiting to enter critical region of input_buffer...\n", id);
94         sem_wait (&SHARED_VARS);
95         testcounter++;
96         printf("Thread %d: ref_buffer = read_input_buffer();                testcounter =
           %d\n\n", id, testcounter);
97         int i;
98         for (i = 0; i < BUF_SIZE; ++i) {
99             ref_buffer[i] = input_buffer[i];
100         }
101         sem_post (&SHARED_VARS);
102
103         // mic_buffer = read_mic();
104         // align_buffers_in_time();
105         // filter_both_signals();
106         // ref_RMS_value = calculatate_mean_for_ref();
107         float RMS_value_ref = calculate_RMS (ref_buffer);
108         // mean_value_mic = calculatate_mean_for_mic();
109         // correction_coefficients = compare_mean_magnitudes(ref_RMS_value, mean_value_mic);
110
111         printf("Thread %d: Waiting to enter critical region of coefficients...\n", id);
112         sem_wait (&SHARED_VARS);
113         testcounter++;
114         printf("Thread %d: write_coefficients();                testcounter = %d
           \n\n", id, testcounter);
115         sem_post (&SHARED_VARS);
116     }
117 }
118
119 void read_file_to_buffer () {
120     FILE *dataFile;
121     dataFile = fopen("signal.dat", "r");
122     float readBuffer;
123
124     int cnt = 0;
125     while (fscanf(dataFile, "%e", &readBuffer) != EOF) {
126         input_buffer[cnt] = readBuffer;
127         cnt++;
128     }
129     fclose(dataFile);
130 }
131
132 float calculate_RMS (float * data_buffer) {
133     int i;
134     float squared_sum, data_RMS;
135     for ( i = 0 ; i < BUF_SIZE ; i++ ) {
136         squared_sum += data_buffer[i] * data_buffer[i];
137     }
138     data_RMS = squared_sum / BUF_SIZE;
139     data_RMS = sqrt(data_RMS);
140     return data_RMS;
141 }
142
143 void lp_filter (float * data_buffer) {
144     int i;

```

```
145     float a = 0.8825, b = 0.1250;
146     float y, y_old = 0;
147     for (i = 0; i < BUF_SIZE; ++i) {
148         y = a*y_old + b*data_buffer[i];
149         y_old = y;
150     }
151 }
```

Listing 6.1: preliminary multithread code for testing of data flow

A main program flow has been developed and implemented in a main test code. This version is used for debugging the data flow of the threads. Furthermore it contains the complete structure that can be used to initialize the development of a main program. In the following chapter the development of the audio interface for the main program is carried out.

7 Audio Interface

This chapter will document the development of the audio interface for the Base Station which is done in ALSA. The development of the audio interface is done in parallel with the development of the main program, Section 6, and the band pass filter library, Section 8.1. In order to do the independent development of the audio interface a simple low pass filter is developed as well.

Since the Base Station is running Linux, it has been chosen to use the Advanced Linux Sound Architecture(ALSA) Application Programming Interface (API) to control the audio input and output flow.

ALSA consists of multiple kernel drivers, some utility programs and API. ALSA API provides low-level audio control functions for best control and performance. Also ALSA is restricted to running on a Linux kernel. ALSA has plugins, which allows extension to new devices and can be implemented entirely in software.

ALSA architecture consists of 6 interfaces:

- Control interface
- Pulse Code Modulation (PCM) interface
- Raw MIDI interface
- Timer interface
- Sequencer interface
- Mixer interface

For development of the Self Adjusting Loudspeaker System, the audio interfacing system will be developed with help of PCM interface. More information on PCM interface can be found in Section 7.0.1.

Devices in API library can be accessed with help of logical names rather than Linux device files. Format of hardware devices defined as `hw:i,j`, where `i` is the card number and `j` is device on the chosen card. For example, if you want info on your laptops sound card, it can be found by writing `aplay -l` for info on playback devices and `arecord -l` for info on recording devices in Linux console. The output can look as seen in Listing 7.1

```
aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: Intel [HDA Intel], device 0: ALC269 Analog [ALC269 Analog]
  Subdevices: 0/1
  Subdevice #0: subdevice #0

arecord -l
**** List of CAPTURE Hardware Devices ****
card 0: Intel [HDA Intel], device 0: ALC269 Analog [ALC269 Analog]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

Listing 7.1: Info on Intel laptop sound card

As can be seen in Listing 7.1 the Intel [HDA Intel] card have a device and two sub-devices. So if is necessary to access an analog playback device on the first sound card, one can do it by loading a sound card 0 and device 0, as `hw:0,0`. The device 0 have two sub-devices for playback and for recording, as it can be seen in Listing 7.1 playback sub-device have number 0 and record sub-device have number 1.

7.0.1 PCM Interface

PCM is a digital representation of an analog signal. The digital representation of this analog signal takes multiple samples of the amplitude. The samples are taken at a regular interval and the analog data samples are changed to binary data. The samples are very dependent on time, which means that an accurate clock is necessary. If the encoding or decoding clock is not stable, the frequency drift of the unstable clock will have consequences for the output quality of the device. This does not mean that a little difference between the encoding and decoding clock will be a major error. But if the clock is not stable, it will be a larger issue, since a drifting clock will cause distortion in, for example, audio signals.

PCM is used in ALSA as digital audio signal, so generally it can be said that the PCM interface is used to process the recorded analog audio to a digital signal, which can be stored and then reconverted for later use, if that is what the user wants.

7.1 Code

The following sections will contain explanations to the three programs `playback.c`, `capture.c` and `capture_playback.c`. A typical program that uses ALSA PCM interface can be written by a pseudo code of this sort:

- Open interface for capture or playback
- Allocate space for hardware parameters
- Set hardware parameters
- Write hardware parameters
- Prepare interface for playback/capture
- Free hardware parameters space
- Read or write PCM data to the interface in the loop

All of these functions have been written in the library `mypcm.h`.

The overview of the library can be seen on the Listing 7.2. It consists of the ALSA PCM functions with error handling.

```

1  #include <alsa/asoundlib.h>
2
3  #define PCM_DEVICE "plughw:0,0"
4
5
6  /**
7   * Open the PCM device in playback mode
8   * and writes error if device
9   * @param **pcm_handle handle to pcm
10  * @param *card audio card to play to
11  * @param stream Wanted stream
12  * @param mode Open mode
13  */
14 void open_pcm (snd_pcm_t **pcm_handle,
15               char *card,
16               snd_pcm_stream_t stream,
17               int mode)
18 {
19     unsigned int pcm;
20     pcm = snd_pcm_open(pcm_handle, card, stream, 0);
21     if (pcm < 0)
22     {
23         printf("ERROR: Can't open \"%s\" PCM device. %s\n",
24               card, snd_strerror(pcm));
25         exit(1);
26     }
27 }
28
29
30
31 /**

```

```

32  * Restrict a configuration space to contain only
33  * one access type
34  * and writes an error if interleaved mode can't be set
35  * @param **pcm_handle handle to pcm
36  * @param *params configuration space
37  */
38  void set_access (snd_pcm_t *pcm_handle,
39                  snd_pcm_hw_params_t *params)
40  {
41      unsigned int pcm;
42      pcm = snd_pcm_hw_params_set_access(pcm_handle, params,
43                                         SND_PCM_ACCESS_RW_INTERLEAVED);
44      if (pcm < 0)
45      {
46          printf("ERROR: Can't set interleaved mode. %s\n",
47                snd_strerror(pcm));
48          exit(1);
49      }
50  }
51
52  /**
53   * Restrict a configuration space to contain only one format
54   * and writes an error if format can't be set
55   * @param **pcm_handle handle to pcm
56   * @param *params configuration space
57   */
58  void set_format (snd_pcm_t *pcm_handle,
59                  snd_pcm_hw_params_t *params)
60  {
61      unsigned int pcm;
62      pcm = snd_pcm_hw_params_set_format(pcm_handle, params,
63                                         SND_PCM_FORMAT_S16_LE);
64      if (pcm < 0)
65      {
66          printf("ERROR: Can't set format. %s\n",
67                snd_strerror(pcm));
68          exit(1);
69      }
70  }
71
72  /**
73   * Restrict a configuration space to one channels count
74   * and writes an error if channels number can't be set
75   * @param **pcm_handle handle to pcm
76   * @param *params configuration space
77   * @param channels channels count
78   */
79  void set_channels(snd_pcm_t *pcm_handle,
80                  snd_pcm_hw_params_t *params,
81                  int channels)
82  {
83      unsigned int pcm;
84      pcm = snd_pcm_hw_params_set_channels(pcm_handle, params, channels);
85      if (pcm < 0)
86      {
87          printf("ERROR: Can't set channels number. %s\n",
88                snd_strerror(pcm));
89          exit(1);
90      }
91  }
92  }
93
94  /**
95   * Restrict a configuration space to contain only one rate
96   * and writes an error if rate can't be set
97   * @param **pcm_handle handle to pcm
98   * @param *params configuration space
99   * @param rate approximate rate
100  */
101  void set_rate(snd_pcm_t *pcm_handle,
102               snd_pcm_hw_params_t *params,
103               int rate)

```

```

105 {
106     unsigned int pcm;
107     pcm = snd_pcm_hw_params_set_rate(pcm_handle, params, rate, 0);
108     if (pcm < 0)
109     {
110         printf("ERROR: Can't set rate. %s\n",
111             snd_strerror(pcm));
112         exit(1);
113     }
114 }
115
116
117 /**
118  * Set hardware parameters
119  * and writes an errors if parameters can't be set
120  * @param **pcm_handle handle to pcm
121  * @param *params configuration space
122  * @param channels channels count
123  * @param rate approximate rate
124  */
125 void set_params(snd_pcm_t *pcm_handle,
126                 snd_pcm_hw_params_t *params,
127                 int channels,
128                 int rate)
129 {
130     set_access(pcm_handle, params);
131     set_format(pcm_handle, params);
132     set_channels(pcm_handle, params, channels);
133     set_rate(pcm_handle, params, rate);
134 }
135
136
137 /**
138  * Write hardware parameters
139  * and writes an error if hardware parameters can't be set
140  * @param *pcm_handle handle to the pcm
141  * @param *params configuration space
142  */
143 void write_params(snd_pcm_t *pcm_handle,
144                  snd_pcm_hw_params_t *params)
145 {
146     unsigned int pcm;
147     pcm = snd_pcm_hw_params(pcm_handle, params);
148     if (pcm < 0)
149     {
150         printf("ERROR: Can't set hardware parameters. %s\n",
151             snd_strerror(pcm));
152         exit(1);
153     }
154 }
155
156 /**
157  * Prepair audio interface for use
158  * and writes an error if it can't be prepaired
159  * @param *pcm_handle handle to the pcm
160  */
161 void prepair_interface(snd_pcm_t *pcm_handle)
162 {
163     unsigned int pcm;
164     pcm = snd_pcm_prepare(pcm_handle);
165     if (pcm < 0)
166     {
167         fprintf(stderr, "cannot prepare audio interface for use (%s)\n",
168             snd_strerror(pcm));
169         exit(1);
170     }
171 }
172
173 /**
174  * Extracts period time from a configuration space
175  * and writes an error if period time can't be extracted
176  * @param *params configuration space
177  * @return tmp approximate period duration

```

```

178  */
179  unsigned int get_period_time(snd_pcm_hw_params_t *params)
180  {
181      unsigned int tmp, pcm;
182      pcm = snd_pcm_hw_params_get_period_time(params, &tmp, NULL);
183      if (pcm < 0)
184      {
185          printf("ERROR: Can't extract maximum period time.%s\n",
186                snd_strerror(pcm));
187          exit(1);
188      }
189      return tmp;
190  }
191
192  /**
193   * Write contents of a buffer to the PCM
194   * write error when unable to write to PCM device
195   * @param **pcm_handle handle to playback
196   * @param frames period size
197   * @param *buffer pointer to buffer
198   * @param buff_size size of the buffer
199   */
200  void play(snd_pcm_t *pcm_handle,
201            signed short *buff,
202            int buffer_size)
203  {
204      unsigned int pcm;
205      pcm = snd_pcm_writewi(pcm_handle, buff, buffer_size);
206      if (pcm == -EPIPE)
207      {
208          printf("ERROR: an underrun occurred\n");
209          snd_pcm_prepare(pcm_handle);
210      }
211      else if (pcm < 0)
212      {
213          printf("ERROR. Can't write to PCM device. %s\n",
214                snd_strerror(pcm));
215          exit(1);
216      }
217  }
218
219
220
221  /**
222   * Reading from audio card to the buffer
223   * write errors too
224   * @param **pcm_handle handle to playback
225   * @param frames period size
226   * @param period length of period
227   * @param *buffer pointer to buffer
228   * @param buff_size size of the buffer
229   */
230  void record(snd_pcm_t *pcm_handle,
231              signed short *buff,
232              snd_pcm_uframes_t frames)
233  {
234      unsigned int pcm;
235      pcm = snd_pcm_readi(pcm_handle, buff, frames);
236      if (pcm != frames)
237      {
238          fprintf(stderr, "ERROR: read from audio interface failed (%s)\n",
239                snd_strerror(pcm));
240          exit(1);
241      }
242      if (pcm == -EBADFD)
243      {
244          fprintf(stderr, "ERROR: PCM is not in the right state (%s)\n",
245                snd_strerror(pcm));
246          exit(1);
247      }
248      if (pcm == -EPIPE)
249      {
250

```

```

251         fprintf (stderr, "ERROR: an overrun occurred (%s)\n",
252                 snd_strerror (pcm));
253         exit (1);
254     }
255     if (pcm == -ESTRPIPE)
256     {
257         fprintf (stderr, "ERROR: a suspend event occurred (%s)\n",
258                 snd_strerror (pcm));
259         exit (1);
260     }
261 }
262
263 /**
264  * Reading from audio card to the buffer
265  * write errors too
266  * @param **pcm_handle handle to playback
267  * @param frames period size
268  * @param period length of period
269  * @param *buffer pointer to buffer
270  * @param buff_size size of the buffer
271  */
272 void record_2_channels(snd_pcm_t *pcm_handle,
273                       signed short **bufs,
274                       snd_pcm_uframes_t frames)
275 {
276     unsigned int pcm;
277     pcm = snd_pcm_readn (pcm_handle, (void**)bufs, frames);
278     if (pcm != frames)
279     {
280         fprintf (stderr, "ERROR: read from audio interface failed (%s)\n",
281                 snd_strerror (pcm));
282         exit (1);
283     }
284     if (pcm == -EBADFD)
285     {
286         fprintf (stderr, "ERROR: PCM is not in the right state (%s)\n",
287                 snd_strerror (pcm));
288         exit (1);
289     }
290     if (pcm == -EPIPE)
291     {
292         fprintf (stderr, "ERROR: an overrun occurred (%s)\n",
293                 snd_strerror (pcm));
294         exit (1);
295     }
296     if (pcm == -ESTRPIPE)
297     {
298         fprintf (stderr, "ERROR: a suspend event occurred (%s)\n",
299                 snd_strerror (pcm));
300         exit (1);
301     }
302 }
303 }

```

Listing 7.2: PCM interface library

The ALSA PCM interface setup can be seen on the Listing 7.3 which is corresponding to the bullet list shown above.

```

1  open_pcm(&capture_handle, PCM_DEVICE, SND_PCM_STREAM_CAPTURE, 0);
2  snd_pcm_hw_params_malloc (&params);
3  snd_pcm_hw_params_any (capture_handle, params);
4  set_params(capture_handle, params, CHANNELS, RATE);
5  write_params(capture_handle, params);
6  prepare_interface(capture_handle);
7  snd_pcm_hw_params_free (params);

```

Listing 7.3: ALSA PCM setup

7.1.1 playback.c

This program is designed to read contents of a wave file to the buffer and then play buffer with a chosen PCM device. At the beginning the interface is opened for playback, afterwards some hardware parameters are set; `buf`, `buf_size`, `*playback_handle`, etc. Some parameters are user defined, to set the amount of seconds `playback.c` should run etc., as seen in Listing 7.4.

```
1 rate = atoi(argv[1]);
2 channels = atoi(argv[2]);
3 seconds = atoi(argv[3]);
```

Listing 7.4: User set parameters

While there is PCM data to be processed, PCM data will be written from the buffer. After all the PCM data have been processed the interface will shut down.

As shown on the Listing 7.5 first the number of frames has been read from the PCM parameters, then period has been found and then for-loop loop runs for chosen number of seconds.

```
1 snd_pcm_hw_params_get_period_size(params, &frames, 0);
2 period =(float) get_period_time(params);
3 snd_pcm_hw_params_free(params);
4
5 for (i = (seconds * 1000000) / period; i > 0; i--)
6 {
7     read(0,buf,SIZE2);
8     play(playback_handle,buf,frames);
9 }
```

Listing 7.5: main playback loop

7.1.2 capture.c

The `capture.c` program is designed to capture PCM data and store it in a buffer. When the interface is started some hardware parameters will be set; `buf`, `*capture_handle` and `*params`. As seen in Listing 7.6 a for-loop is running forever until the user suspends the program and PCM data samples will be stored in the buffer. Then the buffer content is written to the file or on the screen.

```
1 for (;;)
2 {
3     record(capture_handle,buf,SIZE);
4     for (i=0; i < SIZE; ++i)
5         printf ("%d\n",buf[i]);
6 }
```

Listing 7.6: For loop capturing LOOP number of samples from the PCM interface and writing them into the buffer `buf`

7.1.3 capture_playback.c

The `capture_playback.c` program is a combination of `playback.c` and `capture.c`, which means this program will both capture and playback PCM data. As the two previous programs, some hardware parameters will be declared when the program starts; `buf`, `*capture_handle`, `*playback_params`, etc. As in the `capture.c` program, the PCM data will be progressed through a for loop, as seen in Listing 7.8. The difference in the for loop is that a `play()` function have been added. As it says, this plays the captured PCM data each time the loop is run. Also, there has been made two handles—one for playback and one for capture—and they have been prepared as shown on the Listing 7.7, it's basically the Listing 7.3 duplicated one time for the other handle.

```
1 open_pcm(&capture_handle,PCM_DEVICE,SND_PCM_STREAM_CAPTURE,0);
2 open_pcm(&playback_handle,PCM_DEVICE,SND_PCM_STREAM_PLAYBACK,0);
3 //-----
4 snd_pcm_hw_params_malloc (&playback_params);
5 snd_pcm_hw_params_malloc (&capture_params);
6 //-----
7 snd_pcm_hw_params_any (playback_handle, playback_params);
```

```

8 | snd_pcm_hw_params_any (capture_handle, capture_params);
9 | //-----
10 | set_params(playback_handle, playback_params, CHANNELS, RATE);
11 | set_params(capture_handle, capture_params, CHANNELS, RATE);
12 | //-----
13 | write_params(playback_handle, playback_params);
14 | write_params(capture_handle, capture_params);
15 | //-----
16 | prepare_interface(capture_handle);
17 | prepare_interface(playback_handle);
18 | //-----
19 | snd_pcm_hw_params_free (playback_params);
20 | snd_pcm_hw_params_free (capture_params);

```

Listing 7.7: PCM interface setup for playback and capture

```

1 | for ( ; ; )
2 | {
3 |     record(capture_handle, buf, SIZE);
4 |     play(playback_handle, buf, SIZE);
5 | }

```

Listing 7.8: For loop determining how many samples will be aptured

7.2 Low Pass Filter

To test the capture and playback part through a filter, we start out by designing a simple digital filter. The theory and literature used to design such a filter is taken from [19].

In order to design a simple digital filter a first order digital low pass Butterworth filter is chosen with a cutoff frequency at 1 kHz and a sampling frequency of 44,1 kHz.

7.2.1 Design of Low Pass Filter

The first order filter is consists of a resistor in series with a capacitor see Figure 7.2

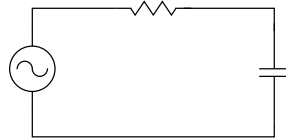


Figure 7.1: Digital low pass Butterworth filter

The standard transfer function for a first order filter can be written as follows Equation 7.8:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b}{1 - az^{-1}} \quad (7.1)$$

The b and a coefficients can be calculated as Equation 7.9:

$$\begin{aligned} b &= \frac{T}{RC} \\ a &= e^{\frac{-T}{RC}} \end{aligned} \quad (7.2)$$

where

$$\begin{aligned} T &= \frac{1}{f_s} \\ &= \frac{1}{44,1\text{E}3} \\ &= 22,67 \text{ uS} \end{aligned} \quad (7.3)$$

To design the first order filter, the value of either the capacitor or the resistor have to be chosen in order to calculate the other component and the a and b coefficients. So first of the resistor is chosen to have the value of $1\text{ k}\Omega$ and then the capacitor can be calculated from the following Equation 7.11:

$$\begin{aligned} C &= \frac{1}{2\pi R f_c} \\ &= \frac{1}{2\pi \cdot 1\text{E}3 \cdot 1\text{E}3} \\ &= 159,16\text{ nF} \end{aligned} \quad (7.4)$$

Now that we now the values of C , R and T , the a and b coefficients can be calculated using the previous Equations 7.9:

$$\begin{aligned} b &= \frac{T}{RC} \\ &= \frac{22,67\text{E}-6}{1\text{E}3 \cdot 159,16\text{E}-9} \\ &= 0,142 \end{aligned} \quad (7.5)$$

$$\begin{aligned} a &= e^{\frac{-T}{RC}} \\ &= e^{-b} \\ &= e^{-0,142} \\ &= 0,867 \end{aligned} \quad (7.6)$$

The transfer function $H(z)$ can now be written as follows 7.13:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0,142}{1 - 0,867 \cdot z^{-1}} \quad (7.7)$$

$H(z)$ can now be implemented in the capture playback C code, by writing the transfer function as a differential equation.

7.2.2 Low Pass Implementation

In the following the low pass filter is implemented in the capture playback C code. Line 9 shows the differential equation for the transfer function $H(z)$ (Eq. 7.10):

```

1 void lp_filter(signed short *input_buf, signed short *output_buf, float gain)
2 {
3     int i;
4     float a = 0.86709;
5     float b = 0.14261;
6     float old_output = 0;
7     for (i = 0; i < SIZE; ++i)
8     {
9         output_buf[i] = gain*(a * old_output + b * input_buf[i]);
10        old_output = (signed short)output_buf[i];
11    }
12 }
```

Listing 7.9: low pass filter implementation

The differential equation in the code The filter is run in a for loop at **SIZE** times which indicate the buffer size. The USB sound card used, is a 16 bit digital sound card and therefore the variables type for the input and output buffer is chosen to be a 16 bit short.

7.3 Low Pass Filter

To test the capture and playback part through a filter, we start out by designing a simple digital filter [19]. In order to design a simple digital filter a first order digital low pass Butterworth filter is chosen with a cutoff frequency at 1 kHz and a sampling frequency of $44,1\text{ kHz}$. The procedure is to first design an analog filter and then convert it into a digital filter.

7.3.1 Design of Low Pass Filter

The first order filter consists of a resistor in series with a capacitor shown in Figure 7.2

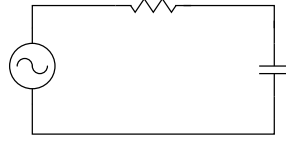


Figure 7.2: Digital low pass Butterworth filter

The standard transfer function for a first order filter can be written as follows Equation 7.8:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b}{1 - az^{-1}} \quad (7.8)$$

The b and a coefficients can be calculated as Equation 7.9:

$$\begin{aligned} b &= \frac{T}{RC} \\ a &= e^{\frac{-T}{RC}} \end{aligned} \quad (7.9)$$

where

$$\begin{aligned} T &= \frac{1}{f_s} \\ &= \frac{1}{44,1\text{E}3} \\ &= 22,67 \text{ uS} \end{aligned} \quad (7.10)$$

To design the first order filter, the value of either capacitor or resistor have to be chosen in order to calculate the other component and the a and b coefficients. So first of the resistor is chosen to have the value of $1 \text{ k}\Omega$ and then the capacitor can be calculated from the following Equation 7.11:

$$\begin{aligned} C &= \frac{1}{2\pi R f_c} \\ &= \frac{1}{2\pi \cdot 1\text{E}3 \cdot 1\text{E}3} \\ &= 159,16 \text{ nF} \end{aligned} \quad (7.11)$$

Now that we know the values of C , R and T , the a and b coefficients can be calculated using the previous Equations 7.9:

$$\begin{aligned} b &= \frac{T}{RC} & a &= e^{\frac{-T}{RC}} \\ &= \frac{22,67\text{E}-6}{1\text{E}3 \cdot 159,16\text{E}-9} & &= e^{-b} \\ &= 0,142 & &= e^{-0,142} \\ & & &= 0,867 \end{aligned} \quad (7.12)$$

The transfer function $H(z)$ can now be written as follows 7.13:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0,142}{1 - 0,867 \cdot z^{-1}} \quad (7.13)$$

$H(z)$ can now be implemented in the capture playback C code, by writing the transfer function as a difference equation.

7.3.2 Low Pass Implementation

In the following the low pass filter is implemented in the capture playback C code. Line 9 shows the difference equation for the transfer function $H(z)$ (Eq. 7.10):

```

1 void lp_filter(signed short *input_buf, signed short *output_buf, float gain)
2 {
3     int i;
4     float a = 0.86709;
5     float b = 0.14261;
6     float old_output = 0;
7     for (i = 0; i < SIZE; ++i)
8     {
9         output_buf[i] = gain*(a * old_output + b * input_buf[i]);
10        old_output =(signed short)output_buf[i];
11    }
12 }
```

Listing 7.10: low pass filter implementation

The difference equation in the code The filter is executed in a for loop at `SIZE` times which indicate the buffer size. The USB sound card used, is a 16 bit and therefore the variables type for the input and output buffer is chosen to be a 16 bit short.

7.4 Test of ALSA PCM Interface With a Filter

This section documents the implementation and different tests of the simple filter designed in Section 7.3 with ALSA PCM interface. Implementation can be divided into three steps:

1. Implement a filter with a `capture` program
2. implement a filter with a `playback` program
3. implement a filter with a `capture_playback` program

When this is finished it will be possible to move these programs into the `main_program` which is implemented with multi-threaded processes. To test if the filter is working it is chosen to use a simple finger click sound, which has been recorded by the program and then plotted with help of MATLAB.

Capture Program The implementation of the program can be seen in Listing 7.11. The filter is applied to the buffer `buf` and the output of the filter goes to the buffer `out_buf`. Data from both buffers are printed to a comma separated file and plotted by use of MATLAB. The last argument of the function `lp_filter()` is a gain which can be changed as necessary.

```

1 for (;;)
2 {
3     record(capture_handle, buf, SIZE);
4     lp_filter(buf, out_buf, 1);
5     for (i=0; i < SIZE; ++i)
6         printf ("%d,%d\n", buf[i], out_buf[i]);
7 }
```

Listing 7.11: Main-loop

In the first test the gain factor is set to 0,5 and in the function `lp_filter()` some code from the filter code has been commented out, so only a simple gain is applied. This is done by multiplying all the samples by 0,5, which is shown in the Figure 7.3.

In the second test the LP-filter has been applied with a gain factor of 0dB or 1, which can be seen on Figure 7.4. It looks like some of the frequencies are attenuated more than others and the signal looks more clean and is without high frequency noise.

In the final test, the filter with a gain factor of 0,5 has been applied, and as expected all signal from 0 Hz to 1000 Hz has been attenuated by factor of 0,5 but frequencies below 1000 Hz have been attenuated even more than in the previous example because of low pass filter characteristics. The result of this test can be seen on Figure 7.5

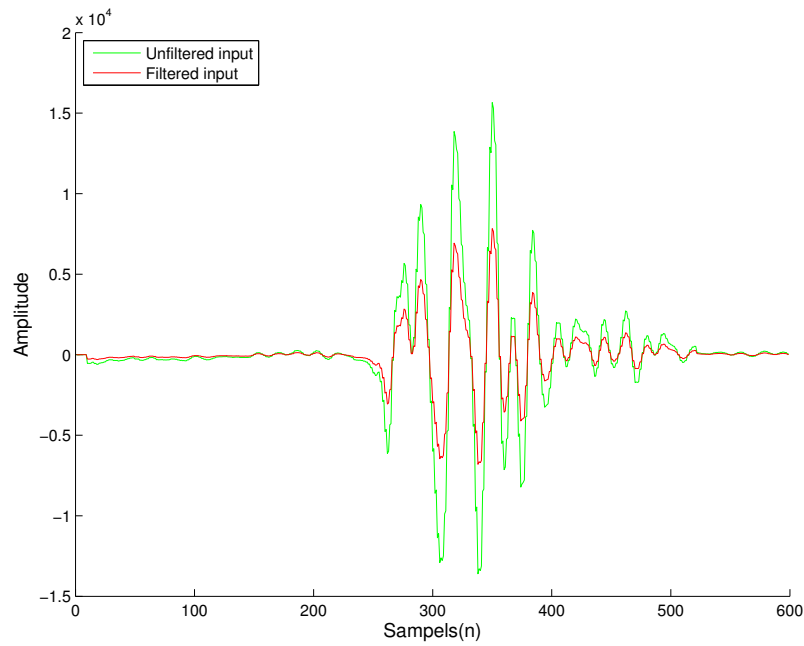


Figure 7.3: A simple gain factor of 0,5 applied on every sample of the PCM data

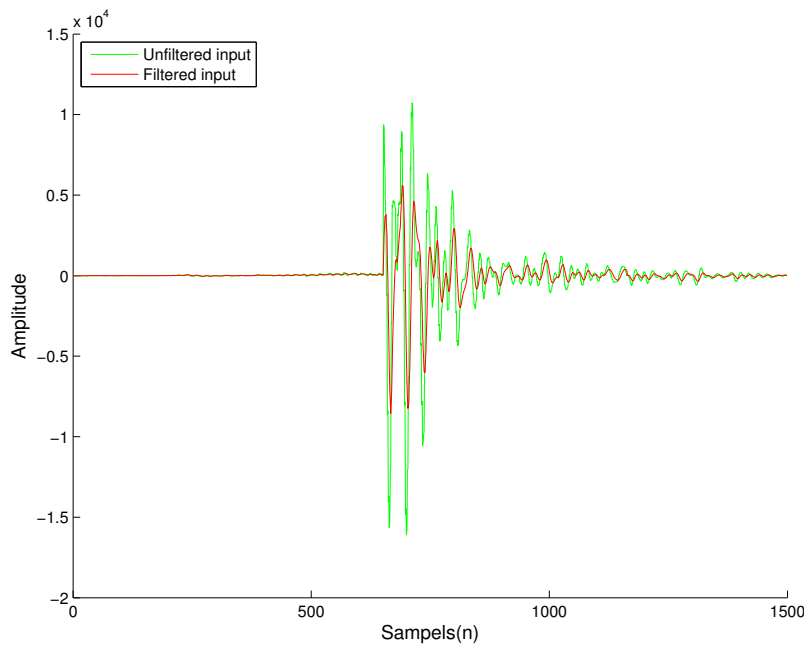


Figure 7.4: Filter applied with a gain of 1 on the PCM data

Playback Program The implementation of the program can be seen in Listing 7.12. The filter is applied to the buffer `start_buf` and its output goes to the buffer `buf`. Data from both buffers are printed to a comma separated file and then plotted in MATLAB. The last argument of the function `lp_filter()` is a gain which can be changed as necessary. Before the filter is run on the input, read data has been copied to the `start_buf`. After the data is recorded to the text file, the function `play()` plays the contents of the buffer `buf` which is then output from the filter function.

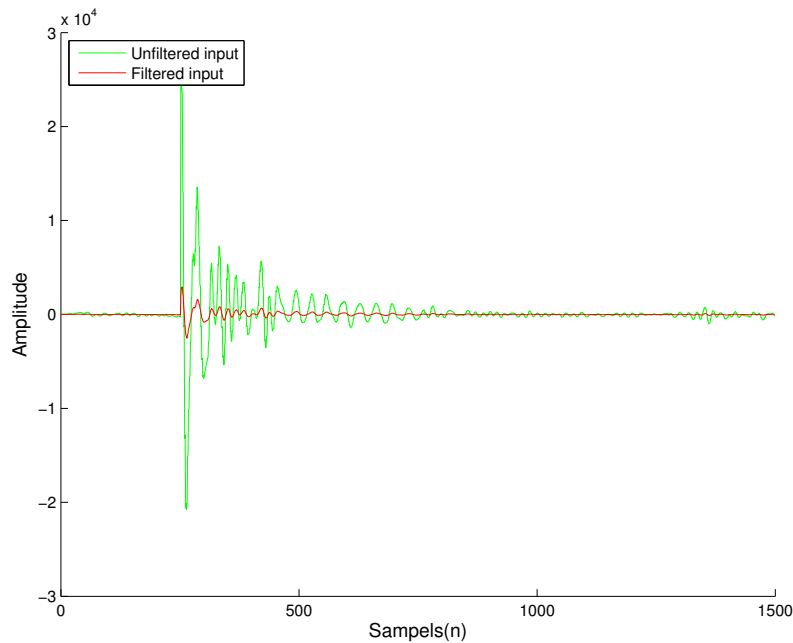


Figure 7.5: Filter applied with a gain of 0,5 on the PCM data

```

1  for (i = (seconds * 1000000) / period; i > 0; i--)
2  {
3      read(0,buf,SIZE2);
4
5      for (k=0; k < SIZE2; ++k)
6          start_buf[k]=buf[k];
7
8      lp_filter(start_buf,buf,1);
9
10     for (k=0; k < SIZE2; ++k)
11         printf ("%d,%d\n",start_buf[k],buf[k]);
12
13     play(playback_handle,buf,frames);
14 }

```

Listing 7.12: for-loop in the main function of the program

In first test the gain factor is at 0,5 and in the function `lp_filter()` some of the filter code has been commented and simple gain equation has been applied. It is multiplying all the samples by 0,5, which can be seen on the Figure 7.6.

In the second test the real filter has been applied with a gain factor of 0dB or 1, which can be seen on Figure 7.7. It looks like some of the frequencies are attenuated more than others and the signal looks more clean and is without high frequency noise.

In the last test filter with a gain factor of 0,5 has been applied, and as expected all signals from 0 Hz to 1000 Hz have been attenuated by factor of 0,5 but frequency below 1000 Hz attenuated even more than in the previous example because of low pass filter characteristics. The result of the test can be seen on Figure 7.8

capture_playback Program To test the `capture_playback` program it is chosen to use a frequency analyzer, which makes a sine sweep over the span of frequencies and it plots the magnitude and phase result. Laptop with the `playback_capture` program with a filter connected to the frequency analyzer. The result can be seen on Figure 7.9. As it can be seen the frequency response has ripple in both stop band and pass band. Later it was found out that the ripple came from an internal microphone amplifier with an automatic gain controller. This is why it was chosen to use an external sound card, which has a line input (*without* automatic gain controller).

It can be seen on the Figure 7.10 frequency response of the filter looks much better and cleaner, but

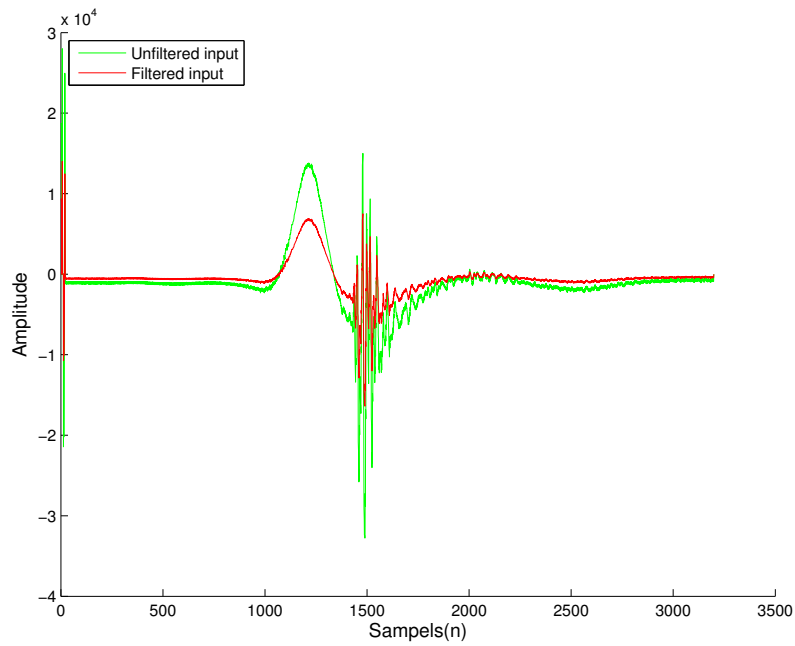


Figure 7.6: A simple gain factor of 0,5 applied on every sample of the PCM data

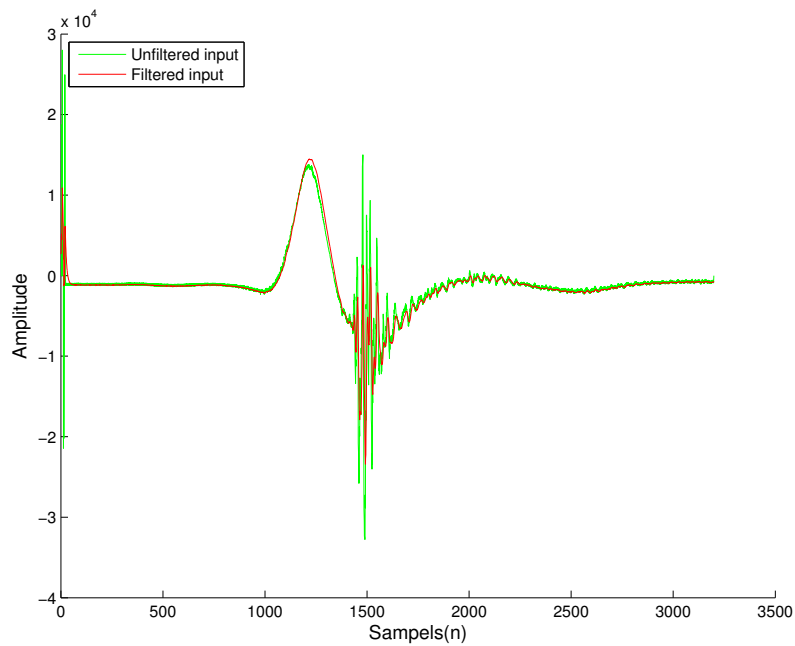


Figure 7.7: Filter applied with a gain of 1 on the PCM data

it looks like some signals at approximate 120 Hz amplified more.

On the Figure 7.11 a response of the filter with a gain factor of 0,5 can be seen. It is seen that the gain is dropping approximately of 22 dB, which makes the noise more significant in the frequency response plot.

The audio interface is developed and tested with the low pass filter and is now ready to be implemented and tested with the band pass filter library of Section 8.1 and the main program of Section 6. In the following chapter the development of the all the filters follows.

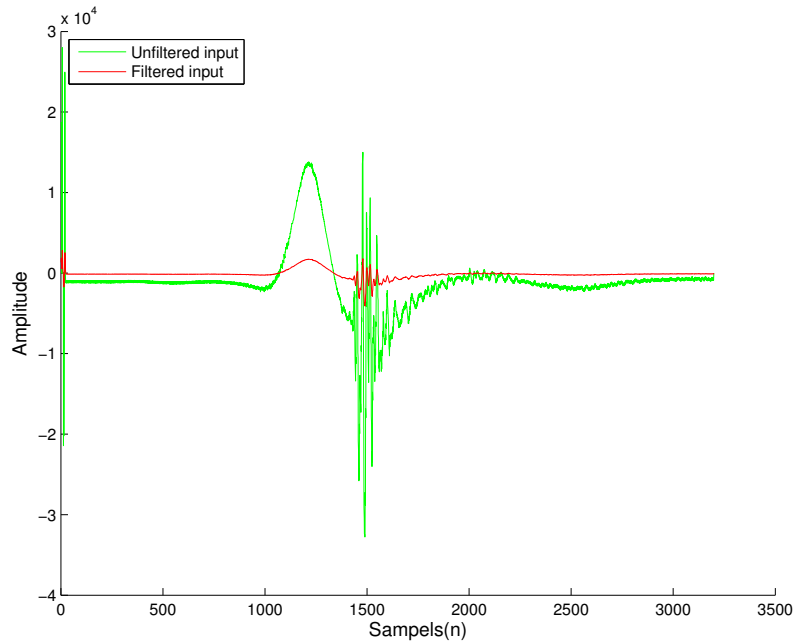


Figure 7.8: Filter applied with a gain of 0,5 on the PCM data

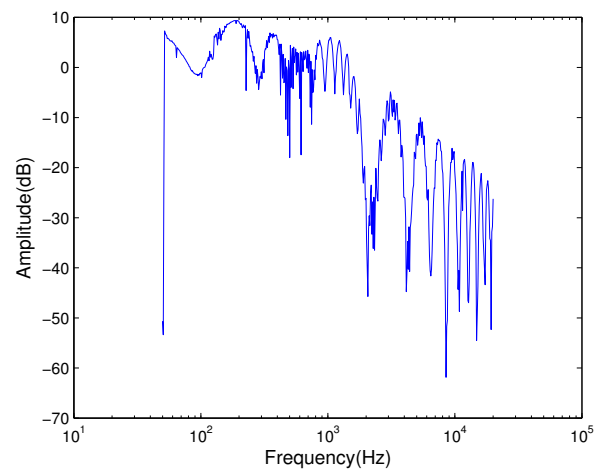


Figure 7.9: Frequency response of a capture_playback program with a filter running on the laptop sound card

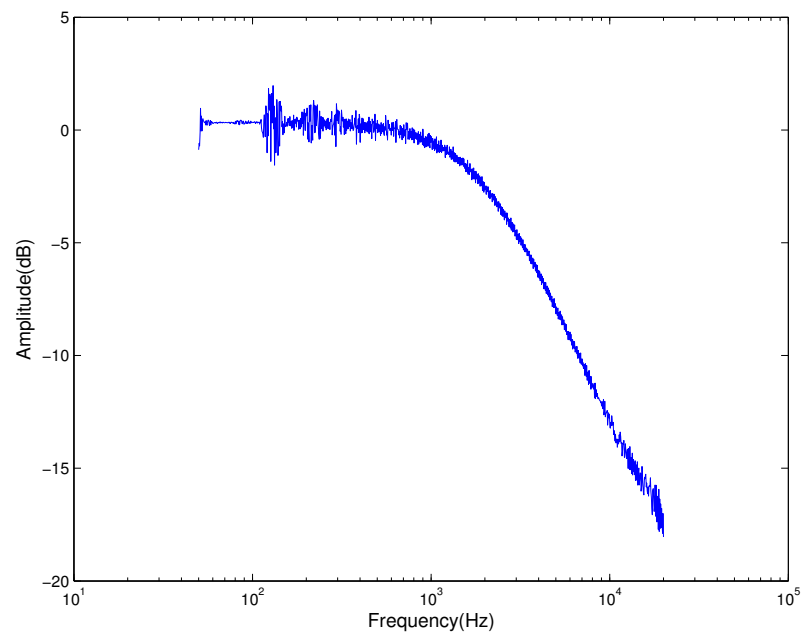


Figure 7.10: Frequency response of a capture_playback program with a filter and gain factor of 1

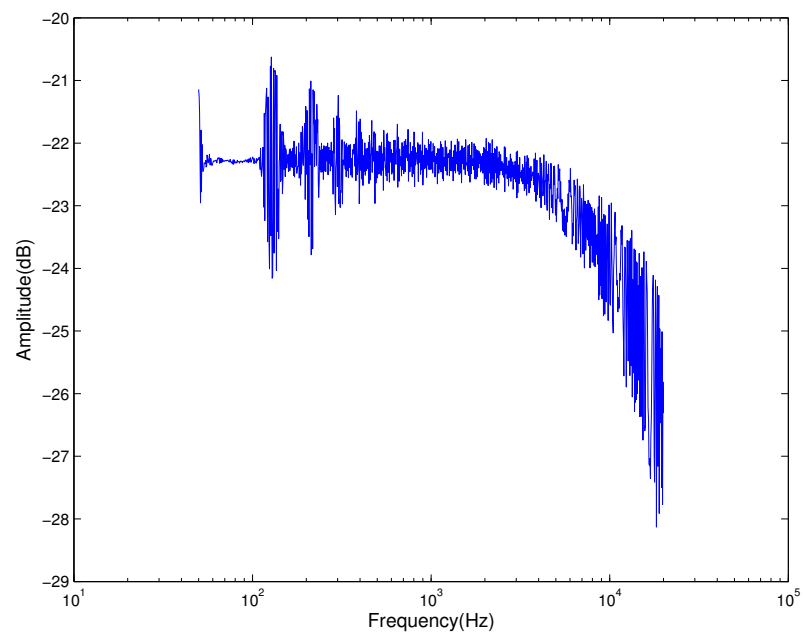


Figure 7.11: Frequency response of a capture_playback program with a filter and gain factor of 0,5

8 Filters

8.1 Band Pass Design

The purpose of this section is to develop the band pass filters which is defined in the requirement specification. It has been specified to develop third octave bands from 1 kHz down to 40 Hz, which is compliant with the standard as investigated further in Section 2.5.

Center Frequency 1 [Hz]	Center Frequency 2 [Hz]	Center Frequency 3 [Hz]
1000	794	630
500	397	315
250	198	158
125	99	79
62,50	50	39

Table 8.1: The center frequencies for one third octave bands, due to the DS/EN 61260 standard. (These values are rounded off to no decimals here.)

The exact center frequencies are defined in Equation 2.1, which yields the center frequencies to be designed from a reference of 1 kHz with proportions as described in Equation 8.1, where x is a band index referenced to 1 kHz. The desired center frequencies are showed in Table 8.1.

$$f_{\text{center}} = 2^{\frac{-x}{3}} \cdot 1 \text{ kHz} \quad (8.1)$$

These filter bands will be attached in parallel. The requirements from Section 4.4.2, is summarized to:

- Maximum pass band ripple: 0,5 dB.
- The band pass filters are of Class 2 (see Figure 2.5).
- The frequency range 40 Hz–1 kHz, should be covered by 1/3 octave bands.

In the following these digital filters will be developed. Documentation in this chapter shows how the design procedure is derived and carried out.

8.2 Choices of Design

The filter analysis, Section 2.5, has shown that a sixth order Butterworth band pass filter will be appropriate for the design. All the filters are implemented on the ARM processor, as software, where the implementation choices is between an IIR or FIR filter. It is chosen to design the IIR, due to the calculation efficiency, because it requires lower order than the FIR. The IIR has the property of non-linear phase, which leads to the risk of instability due to the feedback loop. The most important aspect is the efficiency in respect to the real time analysis—therefore the choice falls on the IIR filter design.

The general procedure is to

1. Develop an analog LP-filter prototype in the laplace domain.
2. Transform it to BP-filter.
3. Transform it to the Z-domain by a method such as the impulse invariant or the bilinear method.

The impulse invariant method will cause aliasing, which is not accepted in the respect. The bilinear method causes no aliasing, but some amount of distorted frequency response. Appropriately, most of this can be fixed by pre-warping the critical frequencies. Therefore the bilinear method will be applied.

8.3 Design Application

Low Pass Design

The Butterworth LP-filter prototype may analytically be described in the normalized domain, as a squared magnitude function:

$$|H(j\omega)|^2 = \frac{1}{1 + \left(\frac{\omega}{\omega_c}\right)^{2N}} \bigg|_{\omega_c=1} = \frac{1}{1 + \omega^{2N}} \quad (8.2)$$

where

- ω is the frequency variable.
- ω_c is the cut off frequency.
- N is the filter order.

To be able to optimize with respect to the filter order, the expression of the necessary order can be defined from deriving the attenuation α , from the Butterworth gain in Equation 8.2.

$$10 \log(|H(j\omega)|^2) = -10 \log(1 + \omega^{2N}) \Rightarrow \frac{\alpha_{dB}}{10} = \log(1 + \omega^{2N}) \Rightarrow \omega^{2N} = 10^{\frac{\alpha_{dB}}{10}} - 1$$

The ratio between the stop band and the pass band edges will express the filter order, so we put up the ratio.

$$\frac{\omega_s^{2N}}{\omega_p^{2N}} = \frac{10^{\frac{\alpha_{s,dB}}{10}} - 1}{10^{\frac{\alpha_{p,dB}}{10}} - 1} \Rightarrow 2N \log\left(\frac{\omega_s}{\omega_p}\right) = \log\left(\frac{10^{\frac{\alpha_{s,dB}}{10}} - 1}{10^{\frac{\alpha_{p,dB}}{10}} - 1}\right) \Rightarrow N \geq \log\left(\frac{10^{\frac{\alpha_{s,dB}}{10}} - 1}{10^{\frac{\alpha_{p,dB}}{10}} - 1}\right) \cdot 2 \log\left(\frac{\omega_s}{\omega_p}\right)^{-1}$$

where

- ω_s is the stop band frequency.
- ω_p is the pass band frequency.
- $\alpha_{s,dB}$ is the stop band attenuation.
- $\alpha_{p,dB}$ is the pass band attenuation.

Pole Location

The poles of the Butterworth LP-filter are to be derived from the denominator zeroes of the expression $|H(j\omega)|^2$, and because $s = j\omega$ it follows that $\omega = s/j$.

$$1 + \omega^{2N} \big|_{\omega=\frac{s}{j}} = 0 \Rightarrow \left(\frac{s^2}{j^2}\right)^N = -1 \Rightarrow (-s^2)^N = -1 = e^{j\pi}$$

and by all the rotations around the unit circle, with k as the counter.

$$(-s^2)_k^N = e^{j\pi + j2\pi k} \Rightarrow s_k = \exp\left(j\left(\frac{\pi + 2\pi k}{2N} - \frac{\pi}{2}\right)\right)$$

For the third order LP there are $2N$ equally distributed poles on the unit circle. Due to stability it is only the ones in the left half plane which is of interest. These poles are then calculated from $k = N + 1$ to $2N$. For the third order filter these are located in

$$\begin{aligned} s_3 &= -\frac{1}{2} + j\frac{\sqrt{3}}{2} \\ s_4 &= -1 \\ s_5 &= -\frac{1}{2} - j\frac{\sqrt{3}}{2} \end{aligned}$$

Because the poles always come in conjugate pairs (if they are not in 1), They can be put on the standard form for second order polynomials as follows:

$$\frac{1}{(s + s_3)(s + s_4)(s + s_5)} = \frac{1}{(s + s_4)(s^2 + 2\text{Re}\{s_3\}s + |s_3|^2)} = \frac{1}{(s - 1) \left(s^2 + 2(-\frac{1}{2})s + \sqrt{(\frac{1}{2})^2 + (\frac{\sqrt{3}}{2})^2} \right)}$$

The analog Butterworth 3rd order low pass prototype is finally defined.

$$H(s) = \frac{1}{(s + 1)(s^2 + s + 1)} \quad (8.3)$$

Band Pass Transformation

Now that the LP prototype has been defined, the band pass prototype will be developed . This is carried out by transforming the LP with a substitution of the s variable.

$$H(s) = \frac{1}{(s + 1)(s^2 + s + 1)} \Big|_{s=\frac{s^2+\Omega_0^2}{\text{BWS}}} = \frac{1}{\left(\frac{s^2+\Omega_0^2}{\text{BWS}} + 1\right) \left(\left(\frac{s^2+\Omega_0^2}{\text{BWS}}\right)^2 + \frac{s^2+\Omega_0^2}{\text{BWS}} + 1\right)}$$

$$H(S) = \frac{\text{BW}^3 S^3}{(S^2 + \Omega_0^2 + \text{BWS})(S^4 + \Omega_0^4 + 2\Omega_0^2 S^2 + \text{BWS}^3 + \text{BW}\Omega_0^2 S + \text{BW}^2 S^2)}$$

where

- Ω_0 is the center frequency.
- BW is the bandwidth.
- S is the frequency variable.

By expanding the denominator and substituting the coefficients, the Butterworth band pass polynomial is derived.

$$H(S) = \frac{\text{BW}^3 S^3}{aS^6 + bS^5 + cS^4 + dS^3 + eS^2 + fS + g} \quad (8.4)$$

where

- $a = 1$
- $b = 2\text{BW}$
- $c = (2\Omega_0^2 + \text{BW}^2) + \text{BW}^2 + \Omega_0^2$
- $d = \text{BW}\Omega_0^2 + \text{BW}(2\Omega_0^2 + \text{BW}^2) + \text{BW}\Omega_0^2$
- $e = \text{BW}^2\Omega_0^2 + \Omega_0^4 + (2\Omega_0^2 + \text{BW}^2)\Omega_0^2$
- $f = 2\text{BW}\Omega_0^4$
- $g = \Omega_0^6$

The design procedure in the analog domain can then be carried out by applying the algorithms that is defined in the filter analysis Section 2.5, Equation 2.1 and 2.2, which is shown by an example of calculation of the 794 Hz band here.

$$\Omega_0 = 2^{\frac{-1}{3}} 1 \text{ kHz} \cdot 2\pi = 794 \text{ Hz} \cdot 2\pi = 5000 \text{ rad/s}$$

$$\text{BW} = \Omega_{\text{upper}} - \Omega_{\text{lower}} = 5000(2^{\frac{1}{6}} - 2^{\frac{-1}{6}}) = 1155 \text{ rad/s}$$

The approval of this design is done by designing a MATLAB code containing a for-loop, which plots the desired filters, by using the specified algorithm. Figure 8.1 shows the resulting filters. The MATLAB code is found on the CD.

 /Source code/filter_analysis/filters_3rd_order_20_20kHz.m

It is now shown the analog straight forward procedure of band pass design. When moving to the digital domain some precautions have to be taken. Frequency deviations occur when using the bilinear transformation. This will be described in the following.

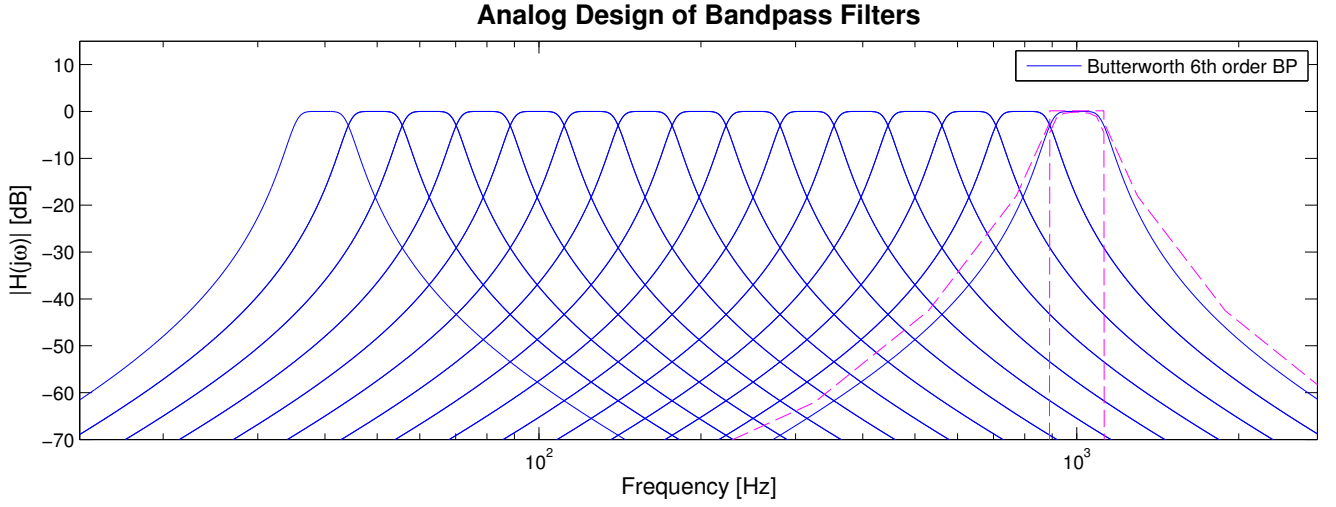


Figure 8.1: The analog design of the specified filters.

Bilinear Transformation

By having a system in the s-domain ($s = j\omega$), which is desired to map into the z-domain ($z = e^{j\omega}$), there will firstly be a short definition of the link between the two domains.

$$z = e^{j\omega} = e^{jT_s\Omega} = e^{jT_s(\Omega/j)} = e^{T_s s} \Rightarrow \ln(z) = \ln(e^{T_s s}) \Rightarrow s = \frac{\ln(z)}{T_s}$$

where

- Ω is the analog frequency variable.
- ω is the digital frequency variable.
- T_s is the sample period.

From this relationship, it is shown that the s-domain system, is represented in z-domain by the natural logarithm divided by the sample period. Therefore the bilinear transformation is a Taylor approximation of the natural logarithm of z divided by T_s . The bilinear transformation consists of only the first term in the Taylor polynomial, which is showed in Equation 8.5.

$$s = \frac{2}{T_s} \cdot \frac{z-1}{z+1} = \frac{2}{T_s} \cdot \frac{1-z^{-1}}{1+z^{-1}} \quad (8.5)$$

The bilinear transformation maps the frequencies from $-\infty < \Omega < \infty$ to $-\pi < \omega < \pi$. It preserves stability from the left half-plane to the interior of the unit circle. Due to the non-linearity of this transformation a pre-warping of the critical frequencies is necessary before the transformation is carried out. The non-linear relation between the analog and digital frequencies is expressed as.

$$\Omega = \frac{2}{T_s} \tan\left(\frac{\omega}{2}\right) \quad (8.6)$$

Therefore, when designing the specified filter bands, the critical edge frequencies must be pre-warped and then used for defining the z-domain center frequency by first transforming these into the normalized digital domain.

$$\omega_{\text{lower}} = 2^{-1/6} \cdot \frac{\Omega_0}{f_s} \quad \wedge \quad \omega_{\text{upper}} = 2^{1/6} \cdot \frac{\Omega_0}{f_s} \quad [\text{rad/sample}]$$

The lower and upper edge frequencies can now be pre-warped.

$$\Omega_{\text{lower-prewarp}} = \frac{2}{T_s} \cdot \tan\left(\frac{\omega_{\text{lower}}}{2}\right) \quad \wedge \quad \Omega_{\text{upper-prewarp}} = \frac{2}{T_s} \cdot \tan\left(\frac{\omega_{\text{upper}}}{2}\right) \quad [\text{rad/s}]$$

These pre-warped edge frequencies are then inserted in the expression for the center frequency.

$$\Omega_{0\text{-prewarp}} = \sqrt{\Omega_{\text{lower-prewarp}} \cdot \Omega_{\text{upper-prewarp}}}$$

And the pre-warped bandwidth is calculated.

$$\text{BW}_{\text{prewarp}} = \Omega_{\text{upper-prewarp}} - \Omega_{\text{lower-prewarp}}$$

Finally the pre-warped center frequency and bandwidth will then be applied to the bilinear transformation, showed in Equation 8.7.

$$H(S) = \frac{\text{BW}^3 S^3}{aS^6 + bS^5 + cS^4 + dS^3 + eS^2 + fS^1 + g} \bigg|_{S=\frac{2}{T_S} \cdot \frac{1-z^{-1}}{1+z^{-1}}} \quad (8.7)$$

where

- BW is pre-warped.
- Ω_0 is pre-warped.
- $a = 1$
- $b = 2\text{BW}$
- $c = (2\Omega_0^2 + \text{BW}^2) + \text{BW}^2 + \Omega_0^2$
- $d = \text{BW}\Omega_0^2 + \text{BW}(2\Omega_0^2 + \text{BW}^2) + \text{BW}\Omega_0^2$
- $e = \text{BW}^2\Omega_0^2 + \Omega_0^4 + (2\Omega_0^2 + \text{BW}^2)\Omega_0^2$
- $f = 2\text{BW}\Omega_0^4$
- $g = \Omega_0^6$

The approval of this design is done by designing a MATLAB code containing a for loop, which plots the desired filters, by using the specified algorithm. Figure 8.2, shows the resulting filters. The MATLAB code is found on the CD.

 /Source code/filter_design/analog_bandpass_40_1kHz.m

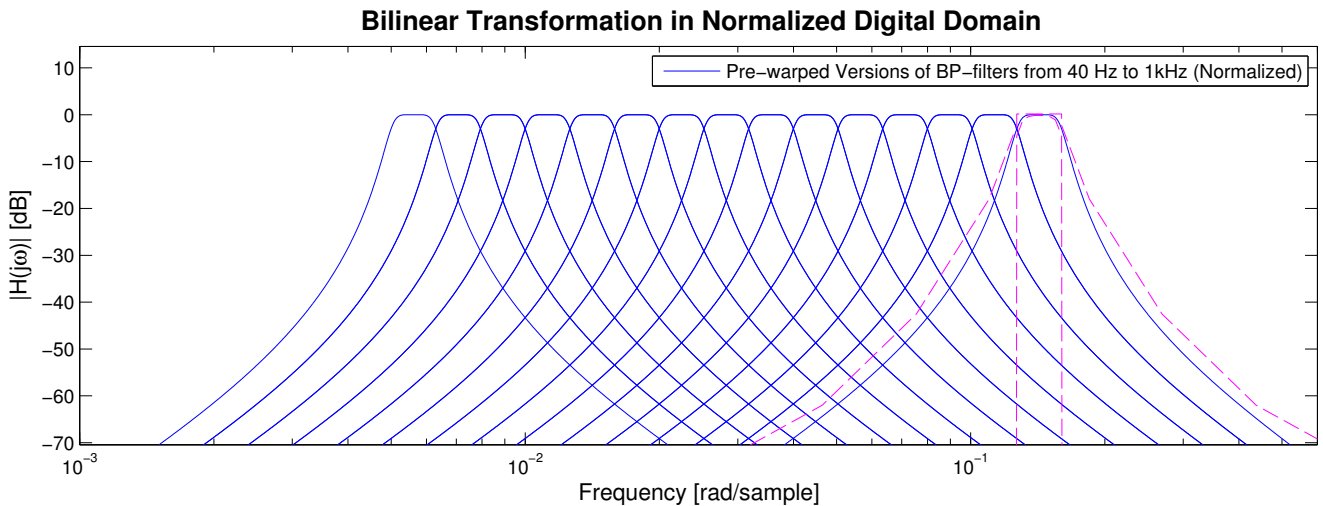


Figure 8.2: Bilinear design of the specified filters

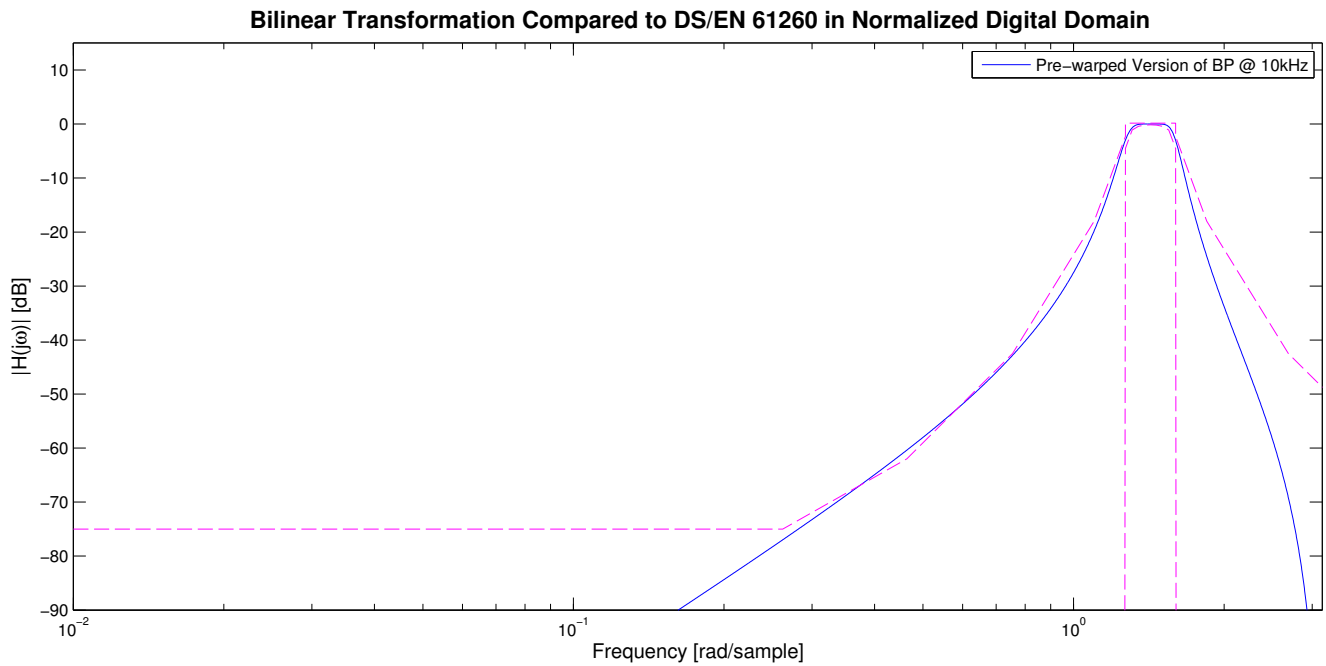


Figure 8.3: Plot in the digital normalized domain of the pre-warped, bilinear 10kHz band. The filter is not compliant with the standard DS/EN 61260, when it has been bilinear transformed.

8.3.1 Coefficients for the Bilinear Transformation

In order to translate the filters to C-code, the coefficients need to be defined. To find the coefficients in the z -domain, the transfer function $H(z)$ is rewritten to the desired form. By doing this, some unexpected non-linearities appeared in the frequency response of the filters. The filters with the greatest difference to the sample rate showed unexpected fluctuations. A plot of this is shown in Figure 8.4. Figure 8.4 shows the analog filter, which is stable with the poles in the left half plane. When carrying out the bilinear transformation on $H(s)$, the poles are mapped differently dependent on the sample rate used, which is shown in the second and third row of Figure 8.4. It should be noted that some poles are outside the unit circle at $f_s = 44,1$ kHz. The errors also appeared when using other math software than MATLAB and plotting the amplitude response, by running a sine sweep audio file through the difference equation of the filter. Therefore it is chosen to continue the filter design, by utilizing the `butter()` function in MATLAB.

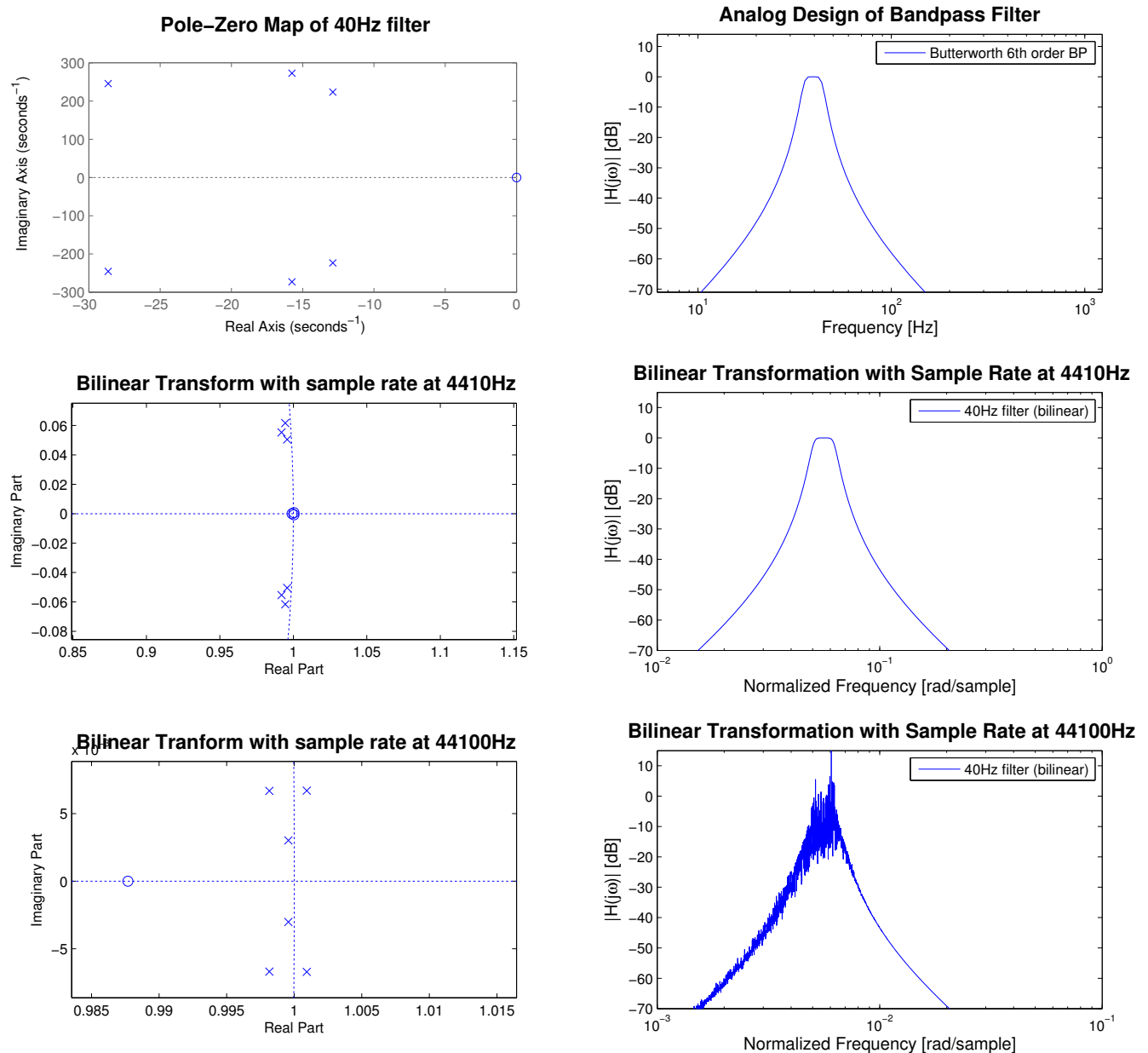


Figure 8.4: Plot showing poles and zeroes, next to plot of amplitude versus frequency, both in S-domain and Z-domain.

8.4 Using MATLAB to Obtain Filter Coefficients

Because of the problems described in the previous section, MATLAB is used to obtain the filter coefficients. MATLAB contains the function `butter()` which returns the zeros and poles for a Butterworth filter of a certain order. To obtain zeros and poles for a 6th order band pass filter, the script in Listing 8.1 is used.

```

1 Fc = 1000; % Varies for the different filters.
2 Fs = 44100;
3 omLower = Fc*2^(-1/6) / (Fs/2);
4 omUpper = Fc*2^(1/6) / (Fs/2);
5
6 [z,p,k] = butter(3,[omLower,omUpper],'bandpass');
```

Listing 8.1: MATLAB script for obtaining poles and zeros of the filters.

where

- F_c is the center frequency of the desired filter (the script needs to be run 15 times with different values for F_c , to obtain all filter coefficients).

- F_s is the sampling rate.
- omLower is the lower edge frequency, normalized to the Nyquist frequency ($F_s/2$).
- omUpper is the upper edge frequency, normalized to the Nyquist frequency.
- z , p , k are the zeros, poles, and the constant term of the desired filter.

Having the zeros and poles for all 15 filters, the each filters coefficients can be found. The relation is like the following, writing the filter as three second order parts:

$$H(z) = k \cdot \frac{(z - z_1)(z - z_2)}{(z - p_1)(z - p_2)} \cdot \frac{(z - z_3)(z - z_4)}{(z - p_3)(z - p_4)} \cdot \frac{(z - z_5)(z - z_6)}{(z - p_5)(z - p_6)}$$

$$= k \cdot \frac{b_{01} + b_{11}z^{-1} + b_{21}z^{-2}}{1 - a_{11}z^{-1} - a_{21}z^{-2}} \cdot \frac{b_{02} + b_{12}z^{-1} + b_{22}z^{-2}}{1 - a_{12}z^{-1} - a_{22}z^{-2}} \cdot \frac{b_{03} + b_{13}z^{-1} + b_{23}z^{-2}}{1 - a_{13}z^{-1} - a_{23}z^{-2}}$$

where

- b_{ij} is i th zero coefficient in the j th biquad (second order system).
- a_{ij} is i th pole coefficient in the j th biquad.

These coefficients are used to realize the difference equations describing each filter. The realization is described later in Section 8.5.

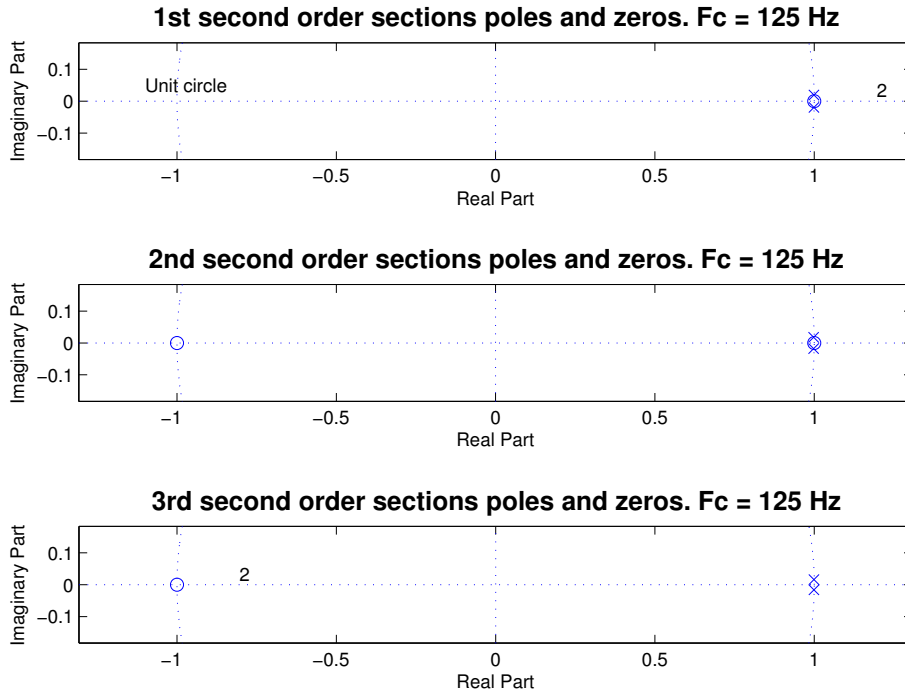


Figure 8.5: Pole zero plot of the filter with a center frequency of 125 Hz. It is seen, that the zeros are very close together.

Poles and zeros that are close together should be grouped into the same biquad. Also, different noise characteristics can be obtained by altering the order of the biquads.[19] This is not taken into account though, as the poles are very close together—especially at the low frequency filters, as seen on Figure 8.5. It is seen that the poles are in complex conjugated pairs. The biquads are constructed of a pair of complex conjugated poles and two zeros that are close to each other, i.e., either in -1 or 1 as seen on Figure 8.5.

In MATLAB the filter coefficients are calculated using the script in Listing 8.2.

```

1 h1num = poly([z(1) z(2)]);
2 h1den = poly([p(1) p(2)]);
3 h2num = poly([z(3) z(4)]);
4 h2den = poly([p(3) p(4)]);

```



```

5  h3num = poly([z(5) z(6)]);
6  h3den = poly([p(5) p(6)]);
7
8  k
9
10 b01 = h1num(1)
11 b11 = h1num(2)
12 b21 = h1num(3)
13 a11 = -h1den(2)
14 a21 = -h1den(3)
15
16 b02 = h2num(1)
17 b12 = h2num(2)
18 b22 = h2num(3)
19 a12 = -h2den(2)
20 a22 = -h2den(3)
21
22 b03 = h3num(1)
23 b13 = h3num(2)
24 b23 = h3num(3)
25 a13 = -h3den(2)
26 a23 = -h3den(3)

```

Listing 8.2: MATLAB script calculating the filter coefficients.

8.5 C Code Implementation—Cascaded Biquads

As suggested in the previous section, each 6th order filter will be constructed as three biquads. The filter coefficients for this construction were calculated, and do now need to be implemented.

The structure of the filter is shown in the flow graph in Figure 8.6. The implementation is done as

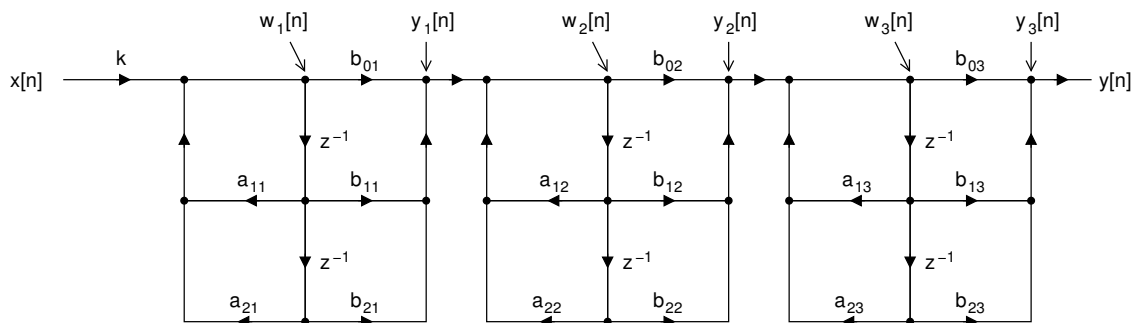


Figure 8.6: Cascaded biquads for a one 6th order filter. The figure is showing intermediate results $w_k[n]$ as well as intermediate outputs $y_k[n]$.

suggested by [19]. This way the following equations will be evaluated to obtain the output:

$$\begin{aligned}
 y_0[n] &= x[n] \\
 w_k[n] &= a_{1k}w_k[n-1] + a_{2k}w_k[n-2] + y_{k-1}[n], & k = 1, 2, 3 \\
 y_k[n] &= b_{0k}w_k[n] + b_{1k}w_k[n-1] + b_{2k}w_k[n-2], & k = 1, 2, 3 \\
 y[n] &= y_3[n]
 \end{aligned}$$

These equations are written explicitly in Appendix B.3. The code is implemented in C in the `butt6bp()` function seen in Listing 8.3.

```

1  /*
2   * Sample rate: 44100 Hz
3   * Filter coeffs: 39, 50, 63, 79, 99, 125, 157, 198, 250, 315, 397, 500, 630, 794, 1000 Hz
4   */
5  static const double k[15] = { ... } // These are shown in the table below.
6
7      static const double b01[15] = { ... };
8  static const double b11[15] = { ... };

```

```

9 static const double b21[15] = { ... };
10 static const double a11[15] = { ... };
11 static const double a21[15] = { ... };
12
13 static const double b02[15] = { ... };
14 static const double b12[15] = { ... };
15 static const double b22[15] = { ... };
16 static const double a12[15] = { ... };
17 static const double a22[15] = { ... };
18
19 static const double b03[15] = { ... };
20 static const double b13[15] = { ... };
21 static const double b23[15] = { ... };
22 static const double a13[15] = { ... };
23 static const double a23[15] = { ... };
24
25 /*
26  * Gain factors for filters. Initialized to 1, but changed during runtime.
27  */
28 static double filter_gains[15] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
29
30 /**
31  * 6th order Butterworth band pass filter.
32  * @param *input Pointer to input signal buffer.
33  * @param *output Pointer to output signal buffer.
34  * @param len Length of buffer.
35  * @param f Number of filter, e.g. f=14 is center freq. 1000 Hz.
36  * @param g Filter gain 0<=g<=1. 0=no input, 1=unity gain,.
37  */
38 void butt6bp(signed short *input, signed short *output, int f)
39 {
40     int i;
41
42     static double w1[NUM_OF_FILTERS] = {0};
43     static double w1_1[NUM_OF_FILTERS] = {0};
44     static double w1_2[NUM_OF_FILTERS] = {0};
45     static double w2[NUM_OF_FILTERS] = {0};
46     static double w2_1[NUM_OF_FILTERS] = {0};
47     static double w2_2[NUM_OF_FILTERS] = {0};
48     static double w3[NUM_OF_FILTERS] = {0};
49     static double w3_1[NUM_OF_FILTERS] = {0};
50     static double w3_2[NUM_OF_FILTERS] = {0};
51     static double y = 0;
52     static double x = 0;
53     static double y1 = 0;
54     static double y2 = 0;
55     static double y3 = 0;
56
57
58     for (i = 0; i < BUF_SIZE; i++)
59     {
60         x = (double)input[i] * k[f];
61
62         w1[f] = a11[f]*w1_1[f] + a21[f]*w1_2[f] + x;
63         y1 = b01[f]*w1[f] + b11[f]*w1_1[f] + b21[f]*w1_2[f];
64         w2[f] = a12[f]*w2_1[f] + a22[f]*w2_2[f] + y1;
65         y2 = b02[f]*w2[f] + b12[f]*w2_1[f] + b22[f]*w2_2[f];
66         w3[f] = a13[f]*w3_1[f] + a23[f]*w3_2[f] + y2;
67         y3 = b03[f]*w3[f] + b13[f]*w3_1[f] + b23[f]*w3_2[f];
68
69         y = y3;
70         output[i] = (signed short)y;
71
72         w3_2[f] = w3_1[f];
73         w3_1[f] = w3[f];
74         w2_2[f] = w2_1[f];
75         w2_1[f] = w2[f];
76         w1_2[f] = w1_1[f];
77         w1_1[f] = w1[f];
78     }
79 }

```

Listing 8.3: C implementation of all 6th order Butterworth filters. The center frequency is chosen by f.

The filter coefficients are not shown in the code, but can be found in Table 8.2.

Variable/ f_c	39 Hz	50 Hz	63 Hz	79 Hz	99 Hz
k	2,659 367 798 839 737E-10	5,601 922 571 251 911E-10	1,120 115 075 688 814E-9	2,207 462 238 523 890E-9	4,341 416 445 165 228E-9
b_{01}	1	1	1	1	1
b_{11}	-2	-2	-2	-2	-2
b_{21}	1	1	1	1	1
a_{11}	1,999 254 893 262 780	1,999 031 193 373 356	1,998 759 143 806 201	1,998 412 901 137 566	1,997 962 397 703 144
a_{21}	-9,992 926 112 858 296E-1	-9,990 931 825 011 605E-1	-9,988 575 458 503 474E-1	-9,985 676 089 307 506E-1	-9,982 053 083 258 046E-1
b_{02}	1	1	1	1	1
b_{12}	0	0	0	0	0
b_{22}	-1	-1	-1	-1	-1
a_{12}	1,998 683 276 491 562	1,998 301 043 512 366	1,997 843 165 472 574	1,997 270 485 968 592	1,996 540 469 688 058
a_{22}	-9,987 141 319 072 618E-1	-9,983 517 499 111 645E-1	-9,979 236 495 164 594E-1	-9,973 970 081 877 148E-1	-9,967 390 963 108 960E-1
b_{03}	1	1	1	1	1
b_{13}	2	2	2	2	2
b_{23}	1	1	1	1	1
a_{13}	1,999 395 853 999 915	1,999 216 384 139 402	1,998 999 140 685 960	1,998 724 118 188 818	1,998 368 479 893 290
a_{23}	-9,994 211 116 542 249E-1	-9,992 578 955 771 124E-1	-9,990 650 377 548 419E-1	-9,988 277 245 149 526E-1	-9,985 311 605 539 993E-1
Variable/ f_c	125 Hz	157 Hz	198 Hz	250 Hz	315 Hz
k	8,731 403 398 760 432E-9	1,728 206 087 410 670E-8	3,461 835 287 058 210E-8	6,956 463 822 613 412E-8	1,388 587 192 625 464E-7
b_{01}	1	1	1	1	1
b_{11}	-2	-2	-2	-2	-2
b_{21}	1	1	1	1	1
a_{11}	1,997 347 360 271 839	1,996 544 830 960 514	1,995 443 198 012 516	1,993 927 584 299 521	1,991 847 178 583 609
a_{21}	-9,977 345 186 383 979E-1	-9,971 553 989 286 375E-1	-9,964 139 111 863 375E-1	-9,954 743 177 319 568E-1	-9,943 011 450 069 916E-1
b_{02}	1	1	1	1	1
b_{12}	0	0	0	0	0
b_{22}	-1	-1	-1	-1	-1
a_{12}	1,995 567 940 406 467	1,994 334 549 015 705	1,992 695 615 644 863	1,990 522 409 584 286	1,987 657 618 375 656
a_{22}	-9,958 844 574 647 879E-1	-9,948 335 959 626 691E-1	-9,934 887 914 962 190E-1	-9,917 857 830 550 013E-1	-9,896 610 927 328 785E-1
b_{03}	1	1	1	1	1
b_{13}	2	2	2	2	2
b_{23}	1	1	1	1	1
a_{13}	1,997 886 458 021 087	1,997 262 659 200 254	1,996 414 203 353 910	1,995 258 673 142 130	1,993 689 494 811 716
a_{23}	-9,981 457 555 329 134E-1	-9,976 716 082 524 942E-1	-9,970 644 223 042 499E-1	-9,962 948 360 575 664E-1	-9,953 336 324 915 507E-1
Variable/ f_c	397 Hz	500 Hz	630 Hz	794 Hz	1000 Hz
k	2,772 330 298 352 829E-7	5,519 738 778 855 591E-7	1,099 473 480 985 984E-6	2,189 284 544 056 890E-6	4,344 426 201 580 417E-6
b_{01}	1	1	1	1	1
b_{11}	-2	-2	-2	-2	-2
b_{21}	1	1	1	1	1
a_{11}	1,988 928 732 726 706	1,984 799 688 788 573	1,978 855 077 139 464	1,970 195 247 752 909	1,957 497 456 588 057
a_{21}	-9,928 232 607 286 578E-1	-9,909 702 966 601 559E-1	-9,886 371 212 065 588E-1	-9,857 027 277 765 992E-1	-9,820 314 903 483 434E-1
b_{02}	1	1	1	1	1
b_{12}	0	0	0	0	0
b_{22}	-1	-1	-1	-1	-1
a_{12}	1,983 809 392 459 594	1,978 607 175 224 339	1,971 459 097 009 606	1,961 522 172 820 266	1,947 602 365 475 845
a_{22}	-9,869 871 374 885 393E-1	-9,836 384 874 869 053E-1	-9,794 279 789 666 762E-1	-9,741 414 201 918 076E-1	-9,675 403 293 533 541E-1
b_{03}	1	1	1	1	1
b_{13}	2	2	2	2	2
b_{23}	1	1	1	1	1
a_{13}	1,991 512 493 364 511	1,988 466 571 507 734	1,984 128 668 706 655	1,977 873 752 705 908	1,968 786 741 748 784
a_{23}	-9,941 222 481 332 931E-1	-9,926 024 975 216 743E-1	-9,906 872 369 088 432E-1	-9,882 754 417 837 817E-1	-9,852 525 379 431 085E-1

Table 8.2: Filter coefficients for a 6th order Butterworth band pass filter.

8.6 Test

The tests for this section were specified in Section 5.2. The test was carried out as described in the work sheet, Appendix A.2. In this section the results of the test will be shown.

The resulting frequency responses are plotted in Figure 8.8 as individual measurements and in Figure 8.8 as a total frequency response there all the filters are added together.

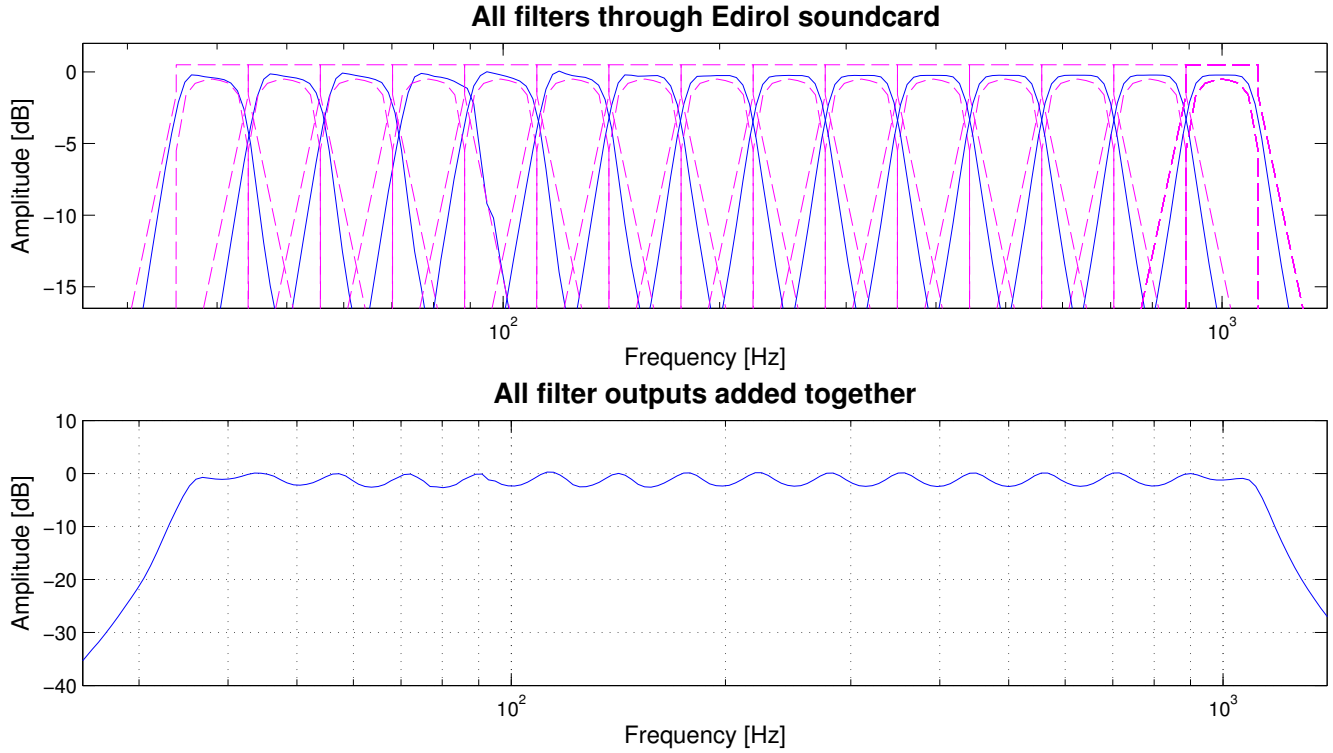


Figure 8.7: Top: Frequency response of the individual filters together with the Class 2 requirements. All filters except for the 79 Hz filters live up to the requirements of a Class 2 filter. Bottom: All filters added together.

It is seen, that all filters, except for the 79 Hz filter, fulfills the requirements of a Class 2 filter. Tables 8.3 and 8.4 show the edge and center frequencies, calculated as described in Section 5.2. Here it also shows that all filters—except for the 79 Hz filter—lies within 2 % of the desired edge and center frequencies.

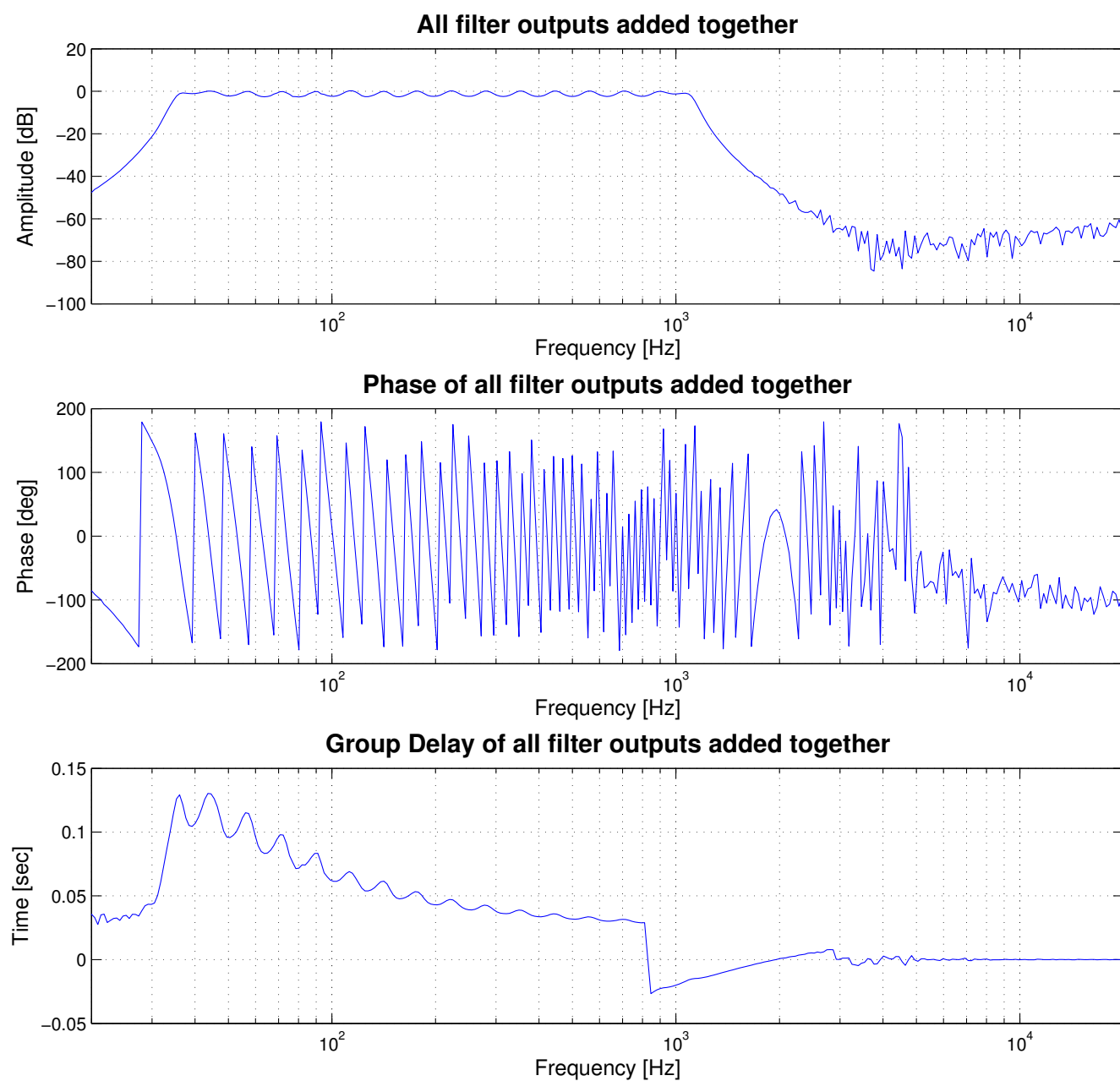


Figure 8.8: Frequency response of all filters added together—amplitude, phase, and group delay.

Filter	Des. f_{low} [Hz]	Act. f_{low} [Hz]	Des. f_c [Hz]	Act. f_c [Hz]	Des. f_{up} [Hz]	Act. f_{up} [Hz]
39	35,08	35,02	39,37	39,14	44,19	43,74
50	44,19	44,89	49,61	50,20	55,68	56,15
63	55,68	56,55	62,50	63,27	70,15	70,79
79	70,15	72,54	78,75	81,44	88,39	91,41
99	88,39	88,90	99,21	99,51	111,36	111,39
125	111,36	112,42	125,00	125,51	140,31	140,14
157	140,31	140,63	157,49	157,22	176,78	175,76
198	176,78	177,11	198,43	198,10	222,72	221,58
250	222,72	223,50	250,00	250,02	280,62	279,70
315	280,62	281,55	314,98	315,00	353,55	352,42
397	353,55	354,82	396,85	397,00	445,45	444,18
500	445,45	446,89	500,00	500,03	561,23	559,49
630	561,23	563,07	629,96	630,05	707,11	704,99
794	707,11	709,60	793,70	794,06	890,90	888,58
1000	890,90	893,68	1000,00	1000,12	1122,46	1119,25

Table 8.3: Desired and actual lower edge, upper edge, and center frequencies for all filters. “Des.” for Desired, “Act.” for Actual.

Filter	Des. f_{low} [Hz]	Act. f_{low} [%]	Des. f_c [Hz]	Act. f_c [%]	Des. f_{up} [Hz]	Act. f_{up} [%]
39	35,08	−0,17	39,37	−0,58	44,19	−1,02
50	44,19	1,58	49,61	1,19	55,68	0,84
63	55,68	1,56	62,50	1,23	70,15	0,91
79	70,15	3,41	78,75	3,42	88,39	3,42
99	88,39	0,58	99,21	0,3	111,36	0,03
125	111,36	0,95	125,00	0,41	140,31	−0,12
157	140,31	0,23	157,49	−0,17	176,78	−0,58
198	176,78	0,19	198,43	−0,17	222,72	−0,51
250	222,72	0,35	250,00	0,01	280,62	−0,33
315	280,62	0,33	314,98	0,01	353,55	−0,32
397	353,55	0,36	396,85	0,04	445,45	−0,29
500	445,45	0,32	500,00	0,01	561,23	−0,31
630	561,23	0,33	629,96	0,01	707,11	−0,3
794	707,11	0,35	793,70	0,05	890,90	−0,26
1000	890,90	0,31	1000,00	0,01	1122,46	−0,29

Table 8.4: Desired and how much the actual measurement differs from the desired in percent. “Des.” for Desired, “Act.” for Actual. The 79 Hz filter is outside the desired range of $\pm 2\%$.

8.7 Implementation of Treble Filter

To cover the treble area, a static filter is needed which is not suppose to be controlled from the main program. Firstly it is decided to use the 6th order biquad filter, C-implementation algorithm. Therefore it is chosen to design a 6th order Butterworth high pass filter with cutoff frequency at 1225 Hz.

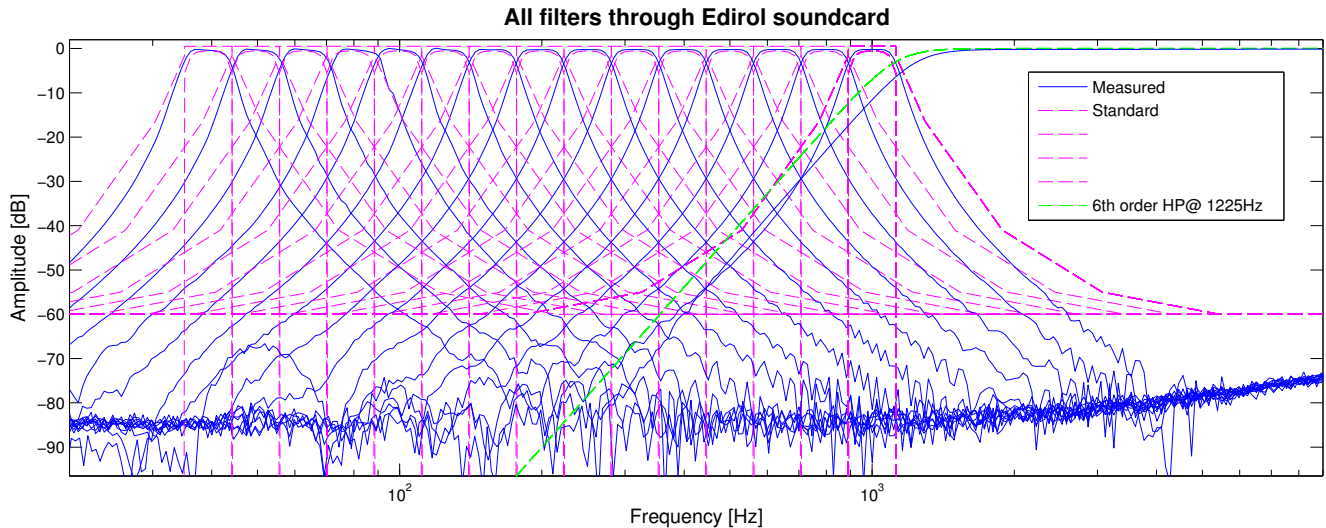


Figure 8.9: Implementation of the 6th order HP-filter at 1235 Hz. All measurements are shown in blue. In green is a calculated 6th order HP-filter at 1225 Hz, which is further designed to 1235 Hz.

8.7.1 6th Order High Pass Filter

An implementation of a 6th order HP-filter with a cut off frequency at 1225 Hz shows that an improved ripple is obtained, compared the 3rd order band pass filter in Section A.3.1. Therefore this design is chosen as a satisfying solution, when not using 30 bands, or another algorithm with a higher order than 6. In this design it is then decided to move the cut off frequency upwards in steps of 10 Hz, execute the C-program, and then measure the ripple again, to see if the ripple will improve in the measurements.

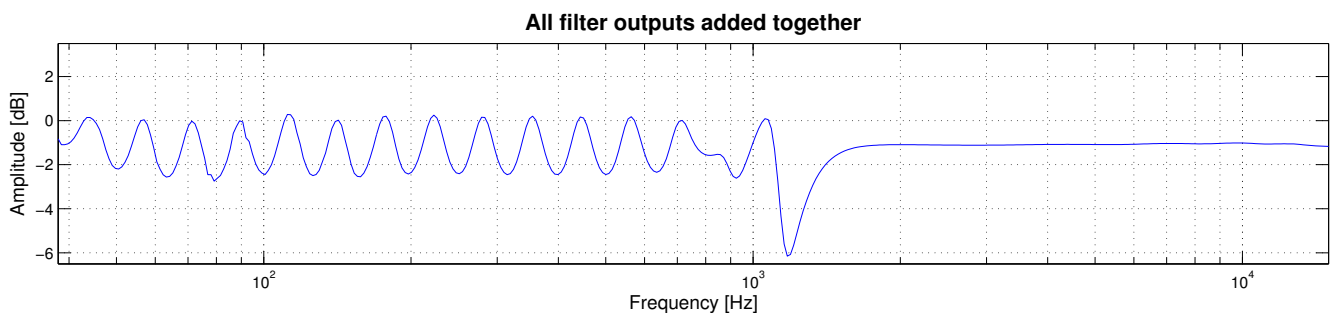


Figure 8.10: Ripple with HP-filter at 1235 Hz and a hp-filter gain factor of 0.9 and 1 kHz gain factor of 0.8.

Because it is preferred to have a smaller ripple especially upwards as a peak, it is decided to implement the high pass, with a cut off frequency at 1235 Hz, shown in Figure A.22b. This filter has only 2 dB peak and is an improvement. The next approach is to lower both the gain factor of the high pass, so that it fits the mean value of the pass band from 40 Hz to 1 kHz and in the same approach, lower the 1 kHz BP-filter gain to further improve the filter frequency response ripple, and especially to keep the filter from clipping, preventing it from peaking at 0 dB. The results are shown in Figure A.23c in Section A.3.3

It is concluded to design the filter with initial settings that limits the attenuation range. The HP-filter is implemented with a maximal gain factor of 0.9. The 1 kHz band has a maximal gain factor of 0.8. These limits are implemented directly in the file `butt6bp.c`. It is furthermore decided from the investigation of

the attenuation range, that it is reasonable to put a lower limit on the gain factor of all filters, in order to keep a predictable attenuation. This limit is 0.2 and is specified through the measurements in the measurement report which can be found in Section A.3.4.

A filter library has now been designed which contains a filter function that controls the behavior of each of the 16 filters—15 band pass filters and 1 high pass filter. A function, where all the filters are summed together, has also been designed and the gain for each of the filters can be controlled by the main program. The main program will be implemented in the Base Station, which is communicating with the Mobile Station, through a cable or through a wireless link. In the following chapter the development of the wireless communication scheme will be described.

9 Wireless

In this chapter the wireless communication scheme will be discussed and developed. The general idea is to build a module that can be used in both the Base Station and the Mobile Station.

For the Base Station, the module needs to be able to receive data from the Mobile Station and output it as an analog signal with a signal level around 2,8 V. The main function of the Mobile Station is to receive data from a microphone and transmit the audio data wirelessly to the base-station. The Mobile Station consists of a microphone, an amplifier, an analog to digital converter, battery circuit, and, of course, the wireless Bluetooth radio interface. It has been decided to use a RN52 Bluetooth Module, which includes all the above besides a battery circuit. The Bluetooth module also includes a DSP, a MCU, and a microphone input which also handles amplification and analog to digital conversion. All this can be seen of the block diagram of the module on Figure 9.1.

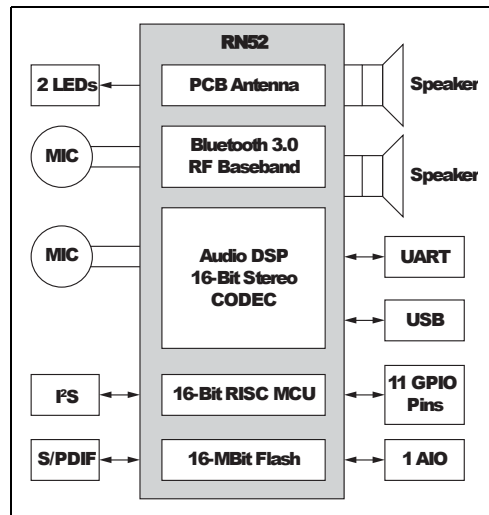


Figure 9.1: Block-diagram of RN-52 [22]

9.1 Bluetooth Module RN52

Since the RN-52 Bluetooth module has been chosen according to the specific requirements set for this module, some of the key features for this module will be listed below to get an overview.

- Frequency band: 2,4–2,48 GHz
- Maximum data rate: 3 Mbps
- Interface: UART, GPIO, AIO, USB, SPI, speaker and microphone
- Operation range: 10 m
- Supply voltage: 1,8–3,6 V
- Standby current: < 0,5 mA

The module uses Bluetooth version 3.0 but is still fully compatible with Bluetooth version 2.1+EDR, 1.2 and 1.1[22]. It also supports various Bluetooth profiles including HSP, HFP, A2DP, AVRCP, SPP, and

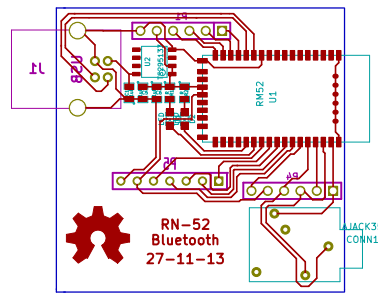


Figure 9.3: Initial PCB for transceiver.

9.3 Control of the RN52

Controlling the Bluetooth module is simple, it consists of a simple UART ASCII command interface, where the module can be configured. There are 3 basic types of commands: SET, GET and ACTION.

The SET-commands stores settings to flash memory, GET-commands retrieves information from the flash memory and displays it and lastly the ACTION-commands performs actions such as controlling volume etc. The basic setup for the Mobile Station is to use the A2DP to stream high quality audio from the Mobile Station to the Base Station. Therefore, some settings needs to be changed in the flash memory of the RN-52.

To enable the UART command-line, GPIO9 on the Bluetooth module needs to be grounded, which as shown on the schematic has been done. Using an FTDI, to establish a UART connection with a PC. To set the discovery profile the SET-command `SD,<8bit hex value>` is used. The datasheet provides a table with different bit-mask values for setting the Bluetooth profile to be used, this table has been copied and can be seen in Table 9.1. The command which needs to be send is therefore `SD,04`.

Bit Position	Value	Profile
0	01	iAP
1	02	SPP
2	04	A2DP
3	08	HFP

Table 9.1: Table showing bitmasks for different Profiles

This is the essential setup, however a few other settings were made, such as changing the name and pin. This is done with the following commands, `SP,<string>` for the Pin and `SN,<string>` to set the name.

9.4 Tests

A test of the frequency response for the Bluetooth module, was performed and showed good results in the lower and mid frequency range, but when a sine of 20 kHz is send through the signal does get distorted. This is however not a problem since the filters only regulate the lower frequencies. The measurement report is in Appendix A.5

9.4.1 Measuring of the Output Impedance

Output impedance has been measured as described in Section 26. The voltage without load has been measured to $V_o = 0,350\text{ V}$. Load has been chosen to be $8,2\Omega$. Voltage with load is measured to be $V_{RL} = 0,225\text{ V}$.

Current running through the load can be found from the voltage drop across the load.

$$I = \frac{V_{RL}}{RL} = 27,439\text{ mA} \quad (9.1)$$

The output impedance can be found as:

$$Z_o = \frac{V_o - V_{RL}}{I} = 4,5556 \Omega \quad (9.2)$$

A functional Bluetooth module has been designed and a PCB has been made for it. It can be directly connected with the USB sound card and be supplied by a USB cable, acting as a 5 V power supply. This subsystem was the last block to be designed in order to complete the full system. In the following chapter the subsystems will be implemented one by one, and then lastly, the complete system will be implemented.

10 Implementation

The implementation chapter documents the implementation of all the technical parts. This includes audio interface, filter library, and the wireless communication.

The implementation is done in three steps as listed below.

1. Main program implemented with audio interface
2. Step 1 + filter library
3. Step 2 + wireless

10.1 Main Program Implementation With ALSA

The overall code of the main program, implemented with the ALSA interface can be seen on Listing 10.1. The program still follows the flowchart shown in Figure 6.2 and Figure 6.3. The `controller()` thread can be seen on line 135 on Listing 10.1 and the `player_filter()` thread can be seen on line 123.



Figure 10.1: Implementation step 1

The first implementations challenge was to split a stereo audio buffer (recorded with `record` function on line 139) to two mono buffers. One of those mono buffers is used for microphone input and the other one used for reference input. They called `mic_input` and `ref_input` respectively, and are shared variables defined on the lines 20 and 21. The function `split_stereo()` can be seen on the line 107 and it runs a for-loop, which puts even buffer elements in the `left` buffer and odd buffer elements in the `right` buffer.

Another function, that has been made under the implementation phase is `calculate_coeff()` function (line 73), which utilizes the `calculate_RMS()` function (line 37) and calculates the gain factor for the filter (line 55). Function `calculate_coeff()` calculates RMS value of the two buffers and subtracts one buffer from the other one to find the difference in RMS value. The function also ensure that the gain factor will never get higher than 1.

The function to calculate the RMS value `calculate_RMS()` shown on line 37 and it basically utilizes the math formula

$$\text{RMS} = \sqrt{\frac{\sum_{n=0}^{N-1} x_n^2}{N}} \quad (10.1)$$

The size of the buffer has been chosen to be higher than 2048. Also it has been found that bigger the buffer the longer the system delay. On the other hand a bigger buffer size can solve such problems as clicking (small noise on the top of the signal).

The `main()` function on line 148 consists of the ALSA setup procedure described in the Section 7 for the capture and for the playback. Then two threads are running in the infinite for loop until the user suspends the program.

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5 #include <math.h>
6 #include "mypcm.h"
7
```

```

8  #define PCM "plughw:1,0"
9  #define BUF_SIZE 2048
10 #define NUM_OF_FILTERS 13
11 #define STERIO 2
12 #define MONO 1
13 #define RATE 44100
14
15 //-----Global variables-----
16
17 sem_t SHARED_VARS;
18 float coefficients[NUM_OF_FILTERS];
19 signed short stereo_input[BUF_SIZE];
20 signed short mic_input[BUF_SIZE/2];
21 signed short ref_input[BUF_SIZE/2];
22
23 snd_pcm_t *capture_handle;
24 snd_pcm_t *playback_handle;
25 snd_pcm_hw_params_t *capture_params;
26 snd_pcm_hw_params_t *playback_params;
27 signed short output_buf[BUF_SIZE/2];
28
29
30 //-----Functions-----
31
32 /**
33  * Function to calculate RMS value.
34  * @param buf pointer to data_buffer.
35  * @return float containing RMS value.
36  */
37 float calculate_RMS (signed short *buf,int size)
38 {
39     float squared_sum;
40     float data_RMS;
41     int i;
42
43     for ( i = 0 ; i < size ; i++ )
44         squared_sum += buf[i] * buf[i];
45     data_RMS = squared_sum / size;
46     data_RMS = sqrt(data_RMS);
47     return data_RMS;
48 }
49
50 /**
51  * Function to apply lp-filter to a buffer.
52  * @param *input_buf input to the filter
53  * @param *output_buf output from the filter
54  */
55 void lp_filter (signed short *input_buf, signed short *output_buf,float gain,int size) {
56     int i;
57     float a = 0.86709;
58     float b = 0.14261;
59     float old_output = 0;
60     for (i = 0; i < size; ++i)
61     {
62         output_buf[i] = gain*(a * old_output + b * input_buf[i]);
63         old_output =(signed short)output_buf[i];
64     }
65 }
66
67 /**
68  * Calculates the gain factor from the difference of RMS value of two buffers
69  * @param *ref_buf pointer to the reference buffer
70  * @param *mic_buf pointer to the microphone buffer
71  * @return gain_factor calculated gain factor
72  */
73 float calculate_coeff(signed short *ref_buf,signed short *mic_buf)
74 {
75     float ref_rms;
76     float mic_rms;
77     float gain_factor;
78     float temp;
79
80     ref_rms = calculate_RMS (ref_buf, BUF_SIZE/2);

```

```

81     mic_rms = calculate_RMS (mic_buf, BUF_SIZE/2);
82     if ((mic_rms - ref_rms) == 0)
83     {
84         gain_factor = 1;
85     }
86     else if ((mic_rms - ref_rms) < 0)
87     {
88         gain_factor = 1;
89     }
90     else
91     {
92         temp = 1 / (mic_rms - ref_rms);
93         if (temp > 1)
94             gain_factor = 1;
95         else
96             gain_factor = temp;
97     }
98     return gain_factor;
99 }
100
101 /**
102  * Split stereo buffer into two mono buffers
103  * @param stereo_buf pointer to the stereo buffer
104  * @param left pointer to left mono buffer
105  * @param right pointer to right mono buffer
106  */
107 void split_stereo (signed short* stereo_buf, signed short* left, signed short* right)
108 {
109     int i;
110     for (i = 0; i < BUF_SIZE/2; i++)
111     {
112         left[i] = stereo_buf[2*i];
113         right[i] = stereo_buf[2*i+1];
114     }
115 }
116
117 //-----Threads-----
118 /**
119  * Thread to playback audio and apply the output filter, by use of semaphore.
120  * @param input ID of thread used in test.
121  * @return void
122  */
123 void player_filter (void* input)
124 {
125     lp_filter(ref_input, output_buf, coefficients[0], BUF_SIZE/2);
126     prepair_interface(playback_handle);
127     play(playback_handle, output_buf, BUF_SIZE/2);
128 }
129
130 /**
131  * Thread to compare the mic- and ref-signals, and write result in semaphore.
132  * @param input ID of thread used in test.
133  * @return void
134  */
135 void controller (void * input)
136 {
137     signed short filtered_mic_input[BUF_SIZE/2];
138     signed short filtered_ref_input[BUF_SIZE/2];
139     record(capture_handle, stereo_input, BUF_SIZE/2);
140     split_stereo(stereo_input, mic_input, ref_input);
141     lp_filter(mic_input, filtered_mic_input, 1, BUF_SIZE/2);
142     lp_filter(ref_input, filtered_ref_input, 1, BUF_SIZE/2);
143     coefficients[0] = calculate_coeff(filtered_ref_input, filtered_mic_input);
144 }
145
146 //-----Main-----
147 int main (int argc, char *argv[])
148 {
149     int id_number[2];
150     pthread_t t_id[2];
151     id_number[0] = 1;
152     id_number[1] = 2;

```



```

154 //-----ALSA setup-----
155
156 open_pcm(&capture_handle,PCM,SND_PCM_STREAM_CAPTURE, 0);
157 snd_pcm_hw_params_malloc (&capture_params);
158 snd_pcm_hw_params_any (capture_handle, capture_params);
159 set_params(capture_handle,capture_params,STERIO,RATE);
160 write_params(capture_handle,capture_params);
161 prepair_interface(capture_handle);
162 snd_pcm_hw_params_free (capture_params);
163
164
165 open_pcm(&playback_handle,PCM,SND_PCM_STREAM_PLAYBACK,0);
166 snd_pcm_hw_params_malloc (&playback_params);
167 snd_pcm_hw_params_any (playback_handle, playback_params);
168 set_params(playback_handle,playback_params,MONO,RATE);
169 write_params(playback_handle,playback_params);
170 prepair_interface(playback_handle);
171 snd_pcm_hw_params_free (playback_params);
172
173
174
175
176 //-----THREADS-----
177
178 sem_init(&SHARED_VARS, 0, 1);
179 for (; ;)
180 {
181     pthread_create(&t_id[0], NULL, (void *) &player_filter, (void *) &id_number[0]);
182     pthread_create(&t_id[1], NULL, (void *) &controller, (void *) &id_number[1]);
183
184     pthread_join(t_id[0], NULL);
185     pthread_join(t_id[1], NULL);
186 }
187 return 0;
188 }
189

```

Listing 10.1: Main program implemented with ALSA interface and a simple filter

10.1.1 Test

All the testing with the ALSA and main program implementation is done with the simple digital Butterworth low pass filter designed in Section 7.3 and on a USB sound card. To test the reference and microphone input one signal generator is used and the amplitude is controlled on the USB sound card. The general test setup can be seen on Figure 10.2.

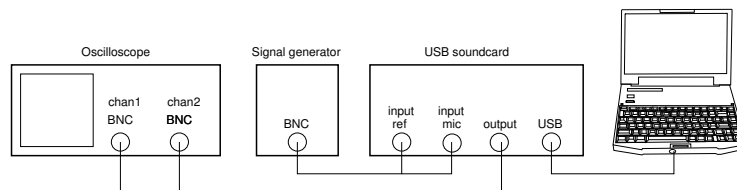


Figure 10.2: Test setup of main program implementation with ALSA

The following equipment are used in the test:

10.1.2 Equipment Used

Instrument	AAU-number.	Manufacture, type etc.
Oscilloscope	64572	Agilent Technologies DSO6034A
Signal generator	08684	N/A
USB sound card	2157-93	Roland

The test is divided into two steps where the code is tested with and without the filter function. This

is done to make sure that the capture playback part is working with the main program.

Test Without Filter

The first test is done without a filter to check that the ALSA code is playing the correct output. To check the output a known signal of 1 kHz is played as input and the input and output are compared on an oscilloscope. The input and output comparison can be seen on Figure 10.3.

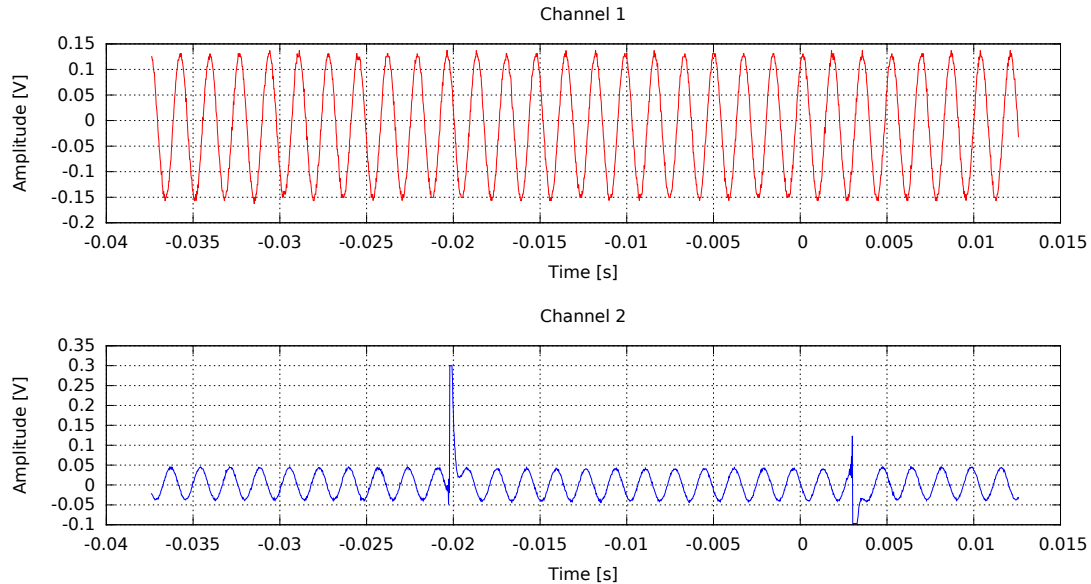


Figure 10.3: Comparison of input and output without a filter.

As shown on Figure 10.3 the output and input signals are not the same. The output and input signals are at the same frequency, but something is causing the output signal to crack, which cause a clicking noise if you listen to the signal. As mentioned in the beginning of this section the buffer size has something to do with the clicking noise and the delay. It seems that the clicking noise is showing each time the buffer ends, however increasing the buffer size didn't stop the clicking. If the buffer size is increased the delay between the clicking noise increases as well, but so is the delay from the input to the output, which means that increasing the buffer size isn't the right solution.

Test With Filter

The second test is done with the same test setup as the first test, but is done by including the filter function as explained earlier. At this time the clicking noise is not corrected, but this test is done only to see if the filter and gain part is working after implementing the main program with ALSA.

The test showed that the filter and gain factor are working correctly and in the same way than before the main program implementation. The earlier test results from the ALSA program, can be seen in Section 7.4

Conclusion

From the comparison Figure 10.3 of the input and output it can be concluded that the program is not working properly and still needs some debugging. In the following 10.1.3 the program has been updated and the clicking error has been resolved.

10.1.3 Final Main Program

To solve the clicking noise the threads has been removed to decrease the overall delay in the program. Furthermore the output buffer was changing each time the function was called and to resolve this problem the filter function has been duplicated three times, one time for each function call. This solution solved the clicking noise, the working code can be seen on Listening 10.2

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5 #include <math.h>
6 #include "mypcm.h"
7
8 #define PCM "plughw:2,0"
9 #define BUF_SIZE 2048
10 #define NUM_OF_FILTERS 16
11 #define STERIO 2
12 #define MONO 1
13 #define RATE 44100
14
15
16 //-----Functions-----
17
18 /**
19  * Function to calculate RMS value.
20  * @param buf pointer to data_buffer.
21  * @return float containing RMS value.
22  */
23 float calculate_RMS (signed short *buf,int size)
24 {
25     float squared_sum;
26     float data_RMS;
27     int i;
28     for ( i = 0 ; i < size ; i++ )
29         squared_sum += buf[i] * buf[i];
30     data_RMS = squared_sum / size;
31     data_RMS = sqrt(data_RMS);
32     return data_RMS;
33 }
34
35
36 /**
37  * Function to apply lp-filter to a buffer.
38  * @param *input_buf input to the filter
39  * @param *output_buf output from the filter
40  */
41 void lp_filter (signed short *input_buf, signed short *output_buf,float
42                gain,int size)
43 {
44     int i;
45     float a = 0.86709,b = 0.14261;
46     static signed short output_buf_old=0;
47
48     for (i=0; i<size; i++) /* Diff. equation: y[n]=bx[n]+ay[n-1] */
49     {
50         output_buf[i]=a*output_buf_old + b*input_buf[i];
51         output_buf_old=output_buf[i];
52     }
53
54     for (i=0; i<size; i++) output_buf[i]=(signed short)(gain*output_buf[i]);
55     /* Gain applied at the end--not in the proc.
56        loop. If in doubt, look at the H(z). */
57 }
58
59
60 /**
61  * Function to apply lp-filter to a buffer.
62  * @param *input_buf input to the filter
63  * @param *output_buf output from the filter
64  */
65 void lp_filter1 (signed short *input_buf, signed short *output_buf,float
66                 gain,int size)
67 {
68     int i;
69     float a = 0.86709,b = 0.14261;
70     static signed short output_buf_old=0;
71
72     for (i=0; i<size; i++) /* Diff. equation: y[n]=bx[n]+ay[n-1] */
73     {

```

```

74         output_buf[i]=a*output_buf_old + b*input_buf[i];
75         output_buf_old=output_buf[i];
76     }
77
78     for (i=0; i<size; i++) output_buf[i]=(signed short)(gain*output_buf[i]);
79     /* Gain applied at the end--not in the proc.
80        loop. If in doubt, look at the H(z). */
81 }
82
83
84 /**
85  * Function to apply lp-filter to a buffer.
86  * @param *input_buf input to the filter
87  * @param *output_buf output from the filter
88  */
89 void lp_filter2 (signed short *input_buf, signed short *output_buf, float
90                 gain, int size)
91 {
92     int i;
93     float a = 0.86709, b = 0.14261;
94     static signed short output_buf_old=0;
95
96     for (i=0; i<size; i++) /* Diff. equation: y[n]=bx[n]+ay[n-1] */
97     {
98         output_buf[i]=a*output_buf_old + b*input_buf[i];
99         output_buf_old=output_buf[i];
100     }
101
102     for (i=0; i<size; i++) output_buf[i]=(signed short)(gain*output_buf[i]);
103     /* Gain applied at the end--not in the proc.
104        loop. If in doubt, look at the H(z). */
105 }
106
107 /**
108  * Calculates the gain factor from the difference of RMS value of two buffers
109  * @param *ref_buf pointer to the reference buffer
110  * @param *mic_buf pointer to the microphone buffer
111  * @return gain_factor calculated gain factor
112  */
113 float calculate_coeff(signed short *ref_buf, signed short *mic_buf)
114 {
115     float ref_rms;
116     float mic_rms;
117     float gain_factor;
118     float temp;
119
120     ref_rms = calculate_RMS (ref_buf, BUF_SIZE/2);
121     mic_rms = calculate_RMS (mic_buf, BUF_SIZE/2);
122     if ((mic_rms-ref_rms)==0)
123     {
124         gain_factor=1;
125     }
126
127     else if ((mic_rms-ref_rms)<0)
128     {
129         gain_factor=1;
130     }
131     else
132     {
133         temp=1/(mic_rms - ref_rms);
134         if (temp > 1)
135             gain_factor=1;
136         else
137             gain_factor=temp;
138     }
139     return gain_factor;
140 }
141 }
142
143 /**
144  * Split stereo buffer into two mono buffers
145  * @param stereo_buf pointer to the stereo buffer
146  * @param left pointer to left mono buffer

```

```

147  * @param right pointer to right mono buffer
148  */
149  void split_stereo (signed short* stereo_buf, signed short* left, signed short* right)
150  {
151      int i;
152      for (i = 0; i < BUF_SIZE/2; i++)
153      {
154          left[i]=stereo_buf[2*i];
155          right[i]=stereo_buf[2*i+1];
156      }
157  }
158
159
160  //-----Main-----
161
162  int main (int argc, char *argv[])
163  {
164      int i;
165      signed short filtered_mic_input[BUF_SIZE/2];
166      signed short filtered_ref_input[BUF_SIZE/2];
167      float coefficients[NUM_OF_FILTERS];
168      signed short stereo_input[BUF_SIZE];
169      signed short mic_input[BUF_SIZE/2];
170      signed short ref_input[BUF_SIZE/2];
171
172
173      snd_pcm_t *capture_handle;
174      snd_pcm_t *playback_handle;
175      snd_pcm_hw_params_t *capture_params;
176      snd_pcm_hw_params_t *playback_params;
177      signed short output_buf[BUF_SIZE/2];
178
179      //-----ALSA setup-----
180
181      open_pcm(&capture_handle, PCM, SND_PCM_STREAM_CAPTURE, 0);
182      snd_pcm_hw_params_malloc (&capture_params);
183      snd_pcm_hw_params_any (capture_handle, capture_params);
184      set_params(capture_handle, capture_params, STERIO, RATE);
185      write_params(capture_handle, capture_params);
186      prepair_interface(capture_handle);
187      snd_pcm_hw_params_free (capture_params);
188
189      open_pcm(&playback_handle, PCM, SND_PCM_STREAM_PLAYBACK, 0);
190      snd_pcm_hw_params_malloc (&playback_params);
191      snd_pcm_hw_params_any (playback_handle, playback_params);
192      set_params(playback_handle, playback_params, MONO, RATE);
193      write_params(playback_handle, playback_params);
194      prepair_interface(playback_handle);
195      snd_pcm_hw_params_free (playback_params);
196
197      for (; ;)
198      {
199          record(capture_handle, stereo_input, BUF_SIZE/2);
200          split_stereo(stereo_input, mic_input, ref_input);
201          lp_filter1(mic_input, filtered_mic_input, 1, BUF_SIZE/2);
202          lp_filter2(ref_input, filtered_ref_input, 1, BUF_SIZE/2);
203          coefficients[0]= calculate_coeff(filtered_ref_input, filtered_mic_input);
204          lp_filter(ref_input, output_buf, coefficients[0], BUF_SIZE/2);
205          play(playback_handle, output_buf, BUF_SIZE/2);
206      }
207      return 0;
208  }

```

Listing 10.2: Final Main Program with low pass filter

10.2 Implementation of the ALSA With the Filter Library

The final code of the main program, implemented with a filter library and ALSA PCM interface, is shown on Listing 10.3.

A new function has been written `find_gains()`, which filters reference input and microphone input

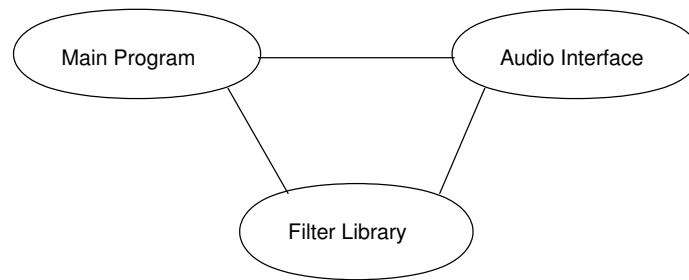


Figure 10.4: Implementation step 2

with a chosen filter with a gain factor of 1. Then it calculates the filter gain in the chosen frequency range. It is done N number of times until the gains for all the filters are calculated. The filter number 16 must remain at attenuation of 0 dB or gain of 1. This is why it has been chosen to set the number of filters to 15 in the function `find_gains()`. The value of the `filter_gains[15]` is set to 1 and can't be changed.

The minor adjustments in the filter library have been made to implement it with a main program. First the `butt6bp()` shown on the Listing 8.3 function has been duplicated 3 times. The duplicated functions `butt6bp1()` and `butt6bp2()` are used in the main program in the function `find_gains()`. The function `butt6bp2()` is still used in the `apply_all_filters()` function.

```

1  open_pcm(&capture_handle,PCM,SND_PCM_STREAM_CAPTURE, 0);
2  snd_pcm_hw_params_malloc (&capture_params);
3  snd_pcm_hw_params_any (capture_handle, capture_params);
4  set_params(capture_handle,capture_params,STERIO,RATE);
5  write_params(capture_handle,capture_params);
6  prepair_interface(capture_handle);
7  snd_pcm_hw_params_free (capture_params);
8
9
10
11 open_pcm(&playback_handle,PCM,SND_PCM_STREAM_PLAYBACK,0);
12 snd_pcm_hw_params_malloc (&playback_params);
13 snd_pcm_hw_params_any (playback_handle, playback_params);
14 set_params(playback_handle,playback_params,MONO,RATE);
15 write_params(playback_handle,playback_params);
16 prepair_interface(playback_handle);
17 snd_pcm_hw_params_free (playback_params);
18
19 for (; ;)
20 {
21     record(capture_handle, stereo_input, BUF_SIZE/2);
22     split_stereo(stereo_input, mic_input, ref_input);
23     find_gains(mic_input, ref_input, filter_gains, NUM_OF_FILTERS-1);
24     apply_all_filters(ref_input, output_buf, BUF_SIZE/2, filter_gains);
25     prepair_interface(playback_handle);
26     play(playback_handle, output_buf, BUF_SIZE/2);
27 }
28 return 0;
29 }
```

Listing 10.3: Main program with ALSA interface and filter library

10.2.1 Full System Test

At this point all elements of the system—except the wireless communication—are implemented. Assuming the wireless communication will not distort the microphone signal, the system functionality can now be tested. Two things will be tested:

- Frequency equalization. The systems ability to make the real life signal sound like the signal reference signal.
- Response time. How long time it takes from the distortion source changes until the filters compensates.

In Figure 10.5 and 10.6 the reference signal is a pink noise source and a sine sweep is input as a noise source (See Figure 10.8 to see where the noise source fits in). The figures show that when the noise signal goes between 40 Hz and 1000 Hz the signal going to the loudspeakers is compensated. Furthermore, it is seen that the system is better at compensating when the signal is slowly changing that when it changes quickly.

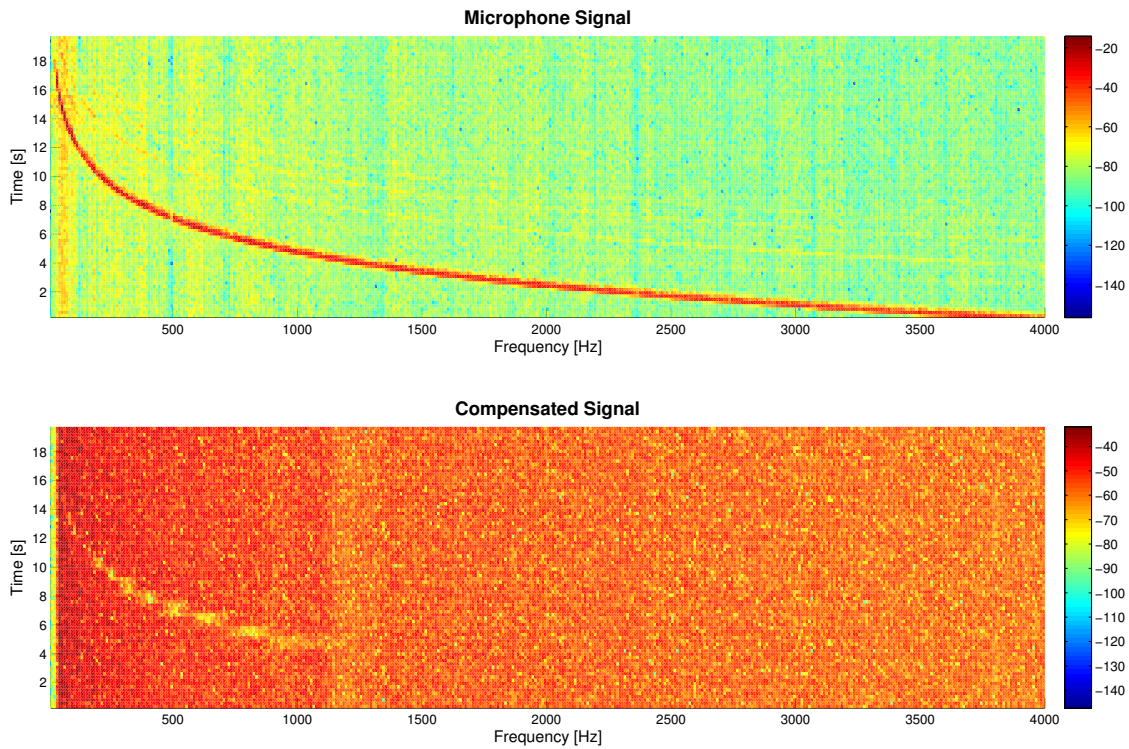


Figure 10.5: 20 second sine sweep noise on top of a pink noise reference. It is seen that the system compensates by lowering the amplitude at the frequencies of the sine sweep, within 40 Hz and 1000 Hz.

The reaction time can be determined from Figure 10.7. A tone is shifted from 440 Hz to 619 Hz on the reference. The reaction time is determined as the time from the reference changes until the microphone signal changes. With this definition of reaction time, the figure shows that

$$t_{\text{reaction}} < 100 \text{ ms} \quad (10.2)$$

10.2.2 Test of implementation of ALSA and Filters

In order to test attenuation of the system, after the implementation of the filter a DAW is as the main interface to the system used.

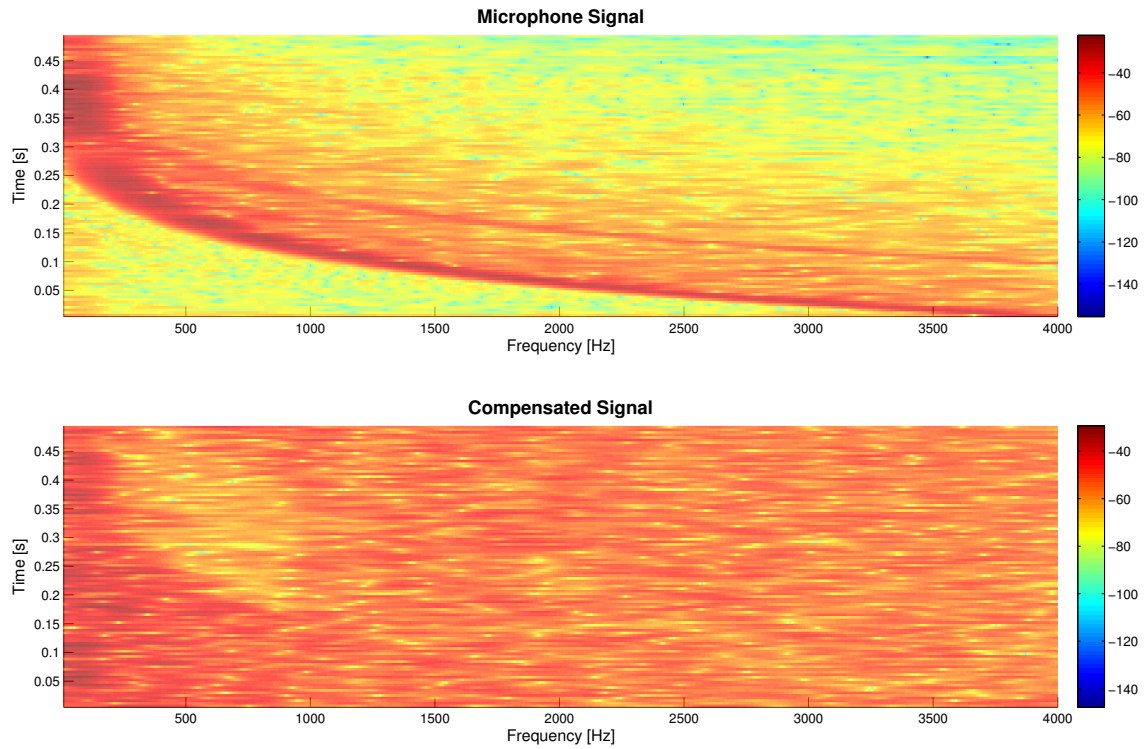


Figure 10.6: 0.5 second sine sweep noise on top of a pink noise reference. A hint of correction can be seen but it does not react as well on the fast sweep as on the slow sweep.

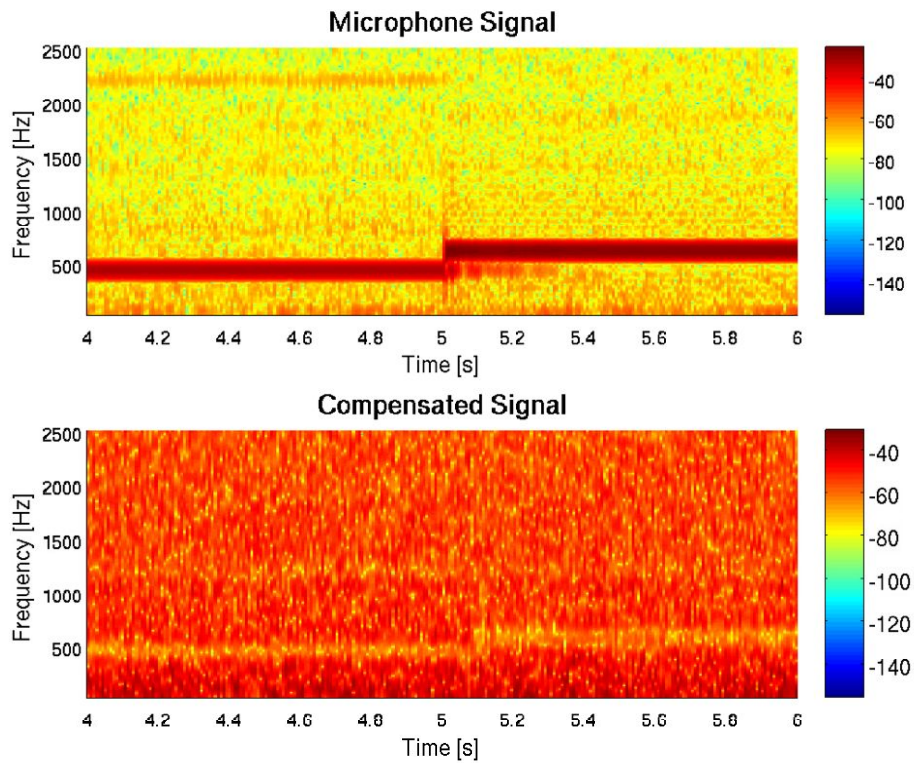


Figure 10.7: The reference is a pink noise signal. On top of this is a sine playing 440Hz and 619Hz. In the bottom it is seen, that the system is trying to compensate for the noise that is applied to the microphone. It is seen, that the reacts in less than 100 ms

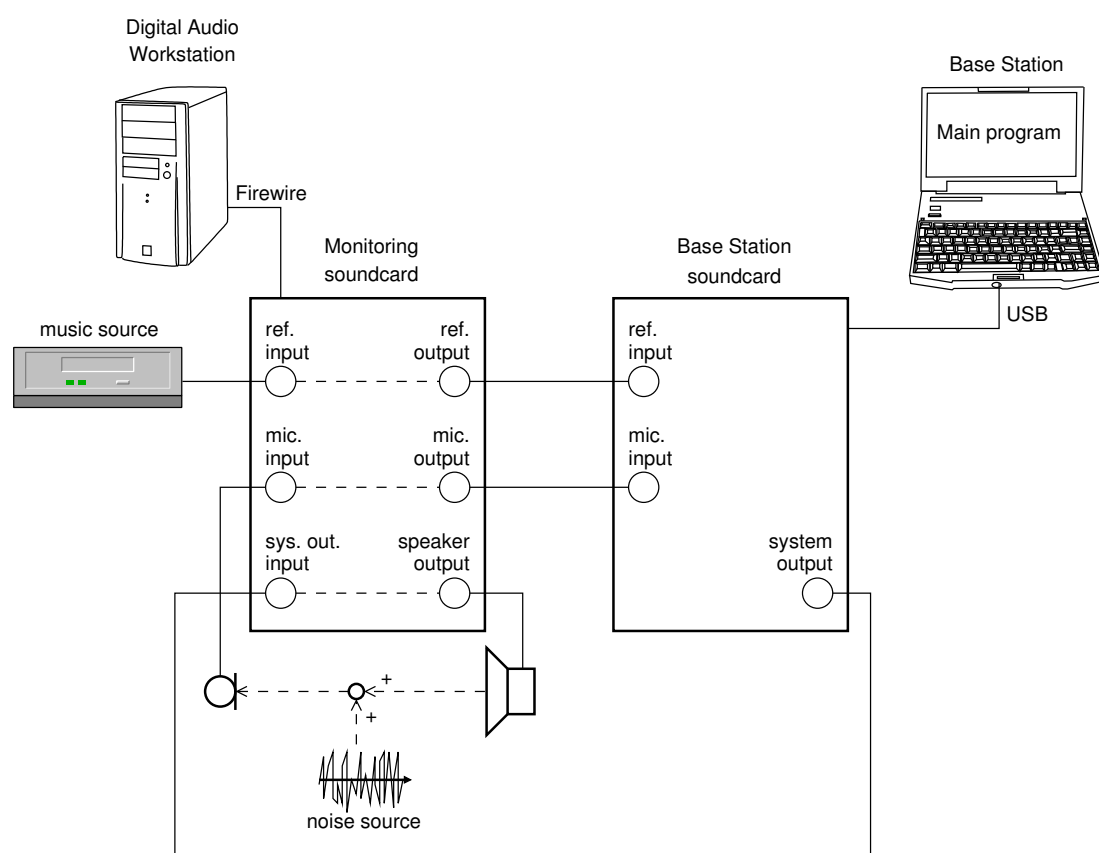


Figure 10.8: Test Setup

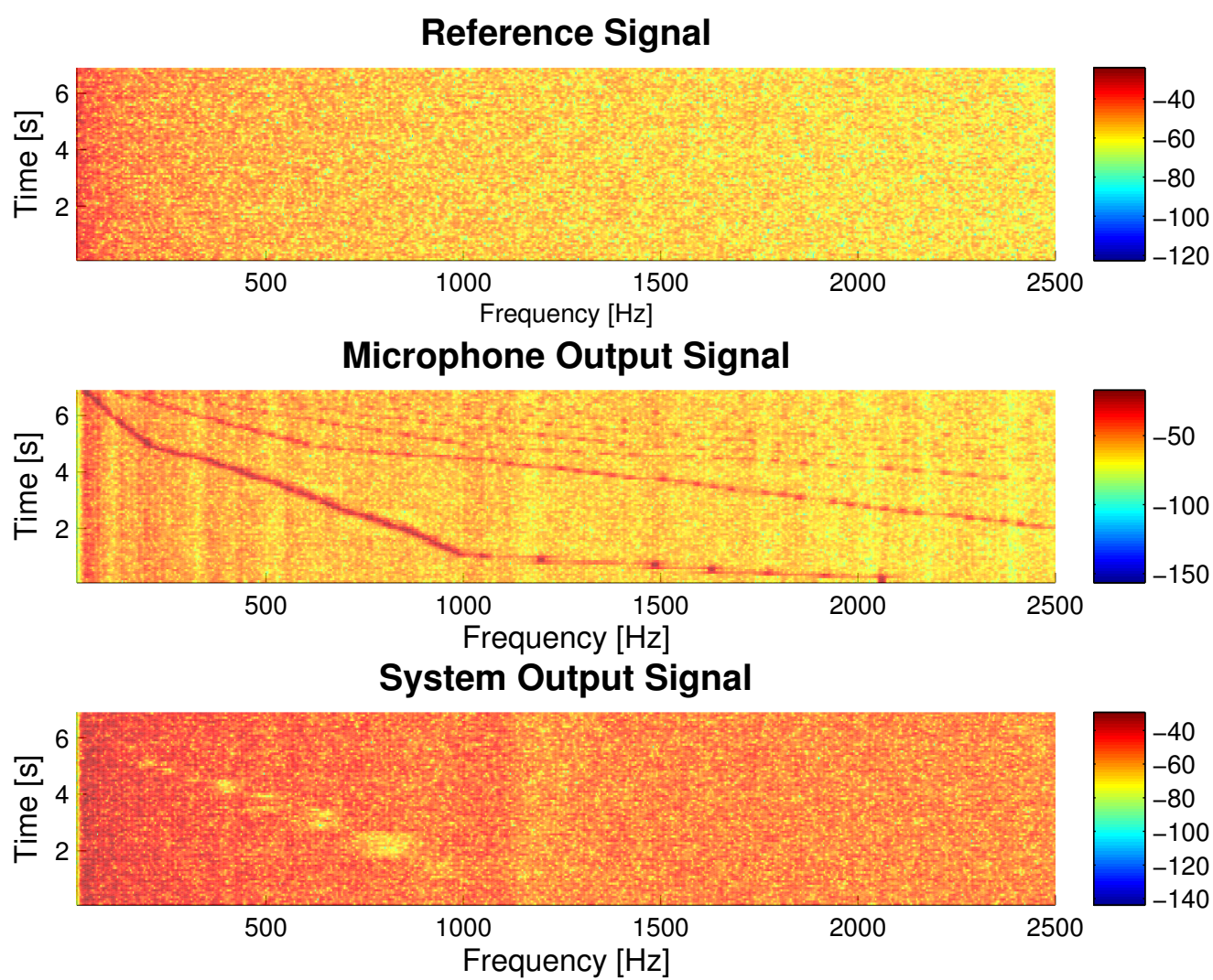


Figure 10.9: Result from test.

10.3 Main Program With ALSA PCM Interface and Bluetooth

This section will describe how the Mobile Station Bluetooth module (described in Chapter 9) and main program (described in Section 10.2) are implemented together.

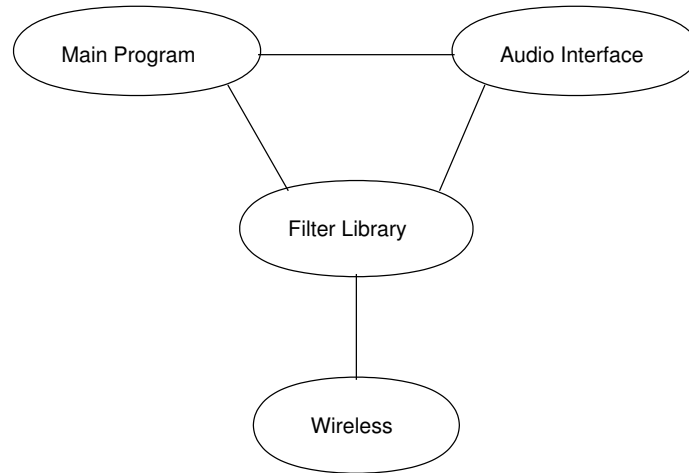


Figure 10.10: Implementation step 3

It is chosen to use a laptop as a Mobile Station. The microphone is connected to the laptop through a microphone input and a Bluetooth sender is connected to the USB port. The built-in microphone and a Bluetooth module can be used too. The Mobile Station laptop running a sound forward, which captures sound from microphone and outputs captured sound through the Bluetooth module.

The main program runs on an additional laptop. The sound card is connected to USB and input 2 connected to the jack input on the laptop. Music played from the laptop is used as reference input (input 2 on sound card). The input 1 on the sound card is used as the microphone input. The microphone input is acquired by receiving it at the Bluetooth receiver and then it is amplified with an amplifier, designed in Section 10.4, to match the signal of the reference input. The active speaker is connected to the output of the sound card and plays the reference input, altered with help of the filters in main program.

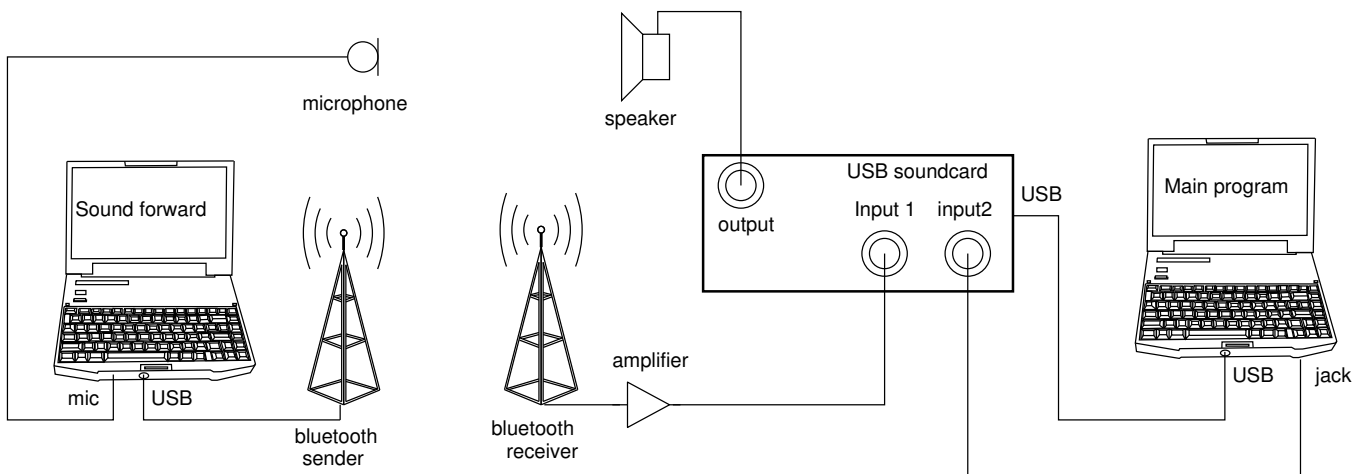


Figure 10.11: Test setup of the final system with a Bluetooth

10.4 Signal Amplifier for Bluetooth module

During the implementation it was found that the output levels were too low, as show in the measurement report. For the Bluetooth module the signal level was around 350 mV_{pp} . Therefore a very simple common-emitter amplifier was made to get the level up to $2,8 \text{ V}$ peak-to-peak, which the input signal amplified 8

times.. The schematic can be seen on Figure 10.12.

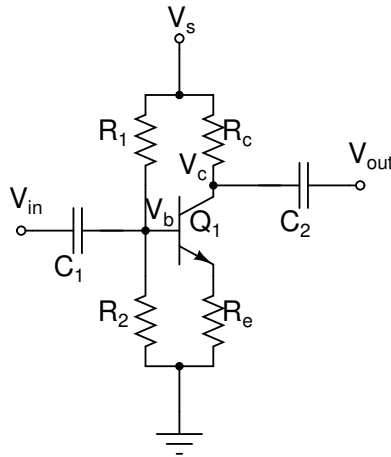


Figure 10.12: Signal amplifier

The calculations for this circuit can be read in Appendix B.1.

The implementation of the technical parts has been made and tested in three steps as mentioned in the beginning of the implementation, Chapter 10.1. The main program is now functional with the audio interface and the filter library implemented on a Linux PC.

III

Conclusion

11 Acceptance Test

✓ = passed, × = not passed, ⊗ = not tested.

Base Station		
Requirement 1	Line input for a regular amplifier, complying to the standard in Table 2.1	⊗
Requirement 2	Line output for a regular amplifier, complying to the standard in Table 2.1	⊗
Filter		
Requirement 3	Sampling rate of audio input	✓
Requirement 4	Bandwidth of regulation	✓
Requirement 5	Frequency range	✓
Requirement 6	Band pass filters	(✓)
Requirement 7	Band pass filters' midband frequencies	(✓)
Requirement 8	Edge frequencies proportional to center frequencies	(✓)
Requirement 9	Pass band ripple	×
Controller		
Requirement 10	Sample rate	✓
Requirement 11	Signal attenuation	⊗
Requirement 12	Signal boost	×
Requirement 13	Response time	✓
Mobile Station		
Requirement 14	Physical size	×
Requirement 15	Weight	×
Wireless Transceiver		
Requirement 16	Frequency band	✓
Requirement 17	Bit rate	✓
Requirement 18	Transceiver range	✓
Requirement 19	Standby power consumption	⊗
Microphone System (based on Gefell M 930+Edirol UA-25EX used in test)		
Requirement 20	Sensitivity value at 120 dB _{SPL}	(✓)
Requirement 21	Linear output signal	×
Requirement 22	SNR	✓
Requirement 23	THD	⊗
Requirement 24	Operation voltage	✓
Requirement 25	Current consumption	⊗
Requirement 26	Output impedance	⊗
Requirement 27	Input impedance	⊗

Table 11.1: Accepttest

12 Conclusion

The starting point for this report is based on the assumption that it is possible to equalize reverberations and standing waves in a room, by applying frequency equalization with electronic filters. This assumption led to a problem analysis with measurements of room acoustics and theory behind sound pressure, microphones, and digital filters compliant with DS/EN 61260. Furthermore, the computer platform and wireless communication schemes were investigated.

From the problem analysis a problem statement was formulated, which is as follows:

How is a self adjusting loudspeaker system designed, which is able to counteract the effect of the frequency characteristics of reflections, by equalizing the signal to a speaker, so that the sound in the air is similar to the audio source—even when the listener is moving around inside the room?

To fulfill this question a list of requirements has been set for the product, and for each requirement a test case has been written. The results are summarized here.

A system diagram has been developed seen in Figure 5.3 with the purpose to clarify subsystem interfaces and provide a clear overview of the designed system and how to translate it in to software, which can be implemented on a Linux platform. The system can be considered as two major elements; the controller and the filter.

The Controller part is considered as the main program that handles the audio interface and processes the comparisons between room measurement and the audio reference.

The Filter part processes a spectrum analysis by use of 15 bands, implemented with a dynamic range of 30 dB respectively. The filters are designed as 6th order band pass filters, compliant with the DS/EN 61260 class 2 standard. The filter bank was developed, calculated, and implemented on a Linux system in the C programming language.

A `capture_playback` program has been developed. The program captures a buffer from ALSA PCM interface, then applies the filters on the buffer and the output is played through the ALSA PCM interface. Furthermore, an associating library has been developed, which utilizes ALSA PCM functions. The program has been tested on a laptop and an external USB sound card Roland UA-22.

The wireless Bluetooth module described in Section 9, is working. However, it was necessary to build a voltage amplifier for the output signal to the Base Station. This should be finalized to a single PCB. Several tests have been conducted to make sure it is working properly. Since only one RN-52 were available only the Base Station side was developed.

The implementation of the subsystems has been done in a systematic fashion, where each block is implemented one at a time. During the implementation some problems appeared in the ALSA code and Bluetooth module. These were, however, easily fixed and documented. The whole system did not work on the Raspberry Pi, as it used up all it's CPU resources. A complete acceptance test has been made, in order to systematically clarify which of the requirements are compliant. Here it was found that the prototype meets the most relevant requirements in the specifications.

In cooperation between the Controller and the Filter, regulation of gain coefficients is verified by doing measurements in a room. The measurements show that the system is counteracting any sound that is different from the audio source. The spectrum analysis and the regulation of gain factors may, in close cooperation with each other, ensure that the sound quality is improved in home audio systems.

All in all a prototype has been developed and includes:

- A Base Station including a laptop and a USB sound card.
- A Mobile Station including a laptop, a USB sound card, a microphone.
- A wireless Bluetooth link from the Mobile Station to the Base Station.

IV

Appendices

A Measurement Reports

A.1 Measurements of RPI Sound Output

A.1.1 Purpose

The purpose of the measurements, is to determine how well the Raspberry Pi is able to reproduce certain sine waves at different frequencies, compared to a Line 6 USB sound card. The measurements will also determine the output voltage at different frequencies for both the Raspberry Pi and the Line 6 sound card.

A.1.2 Test objects

- Raspberry Pi
- Line 6 USB sound card

A.1.3 Test Setup

The following figures show the test setup

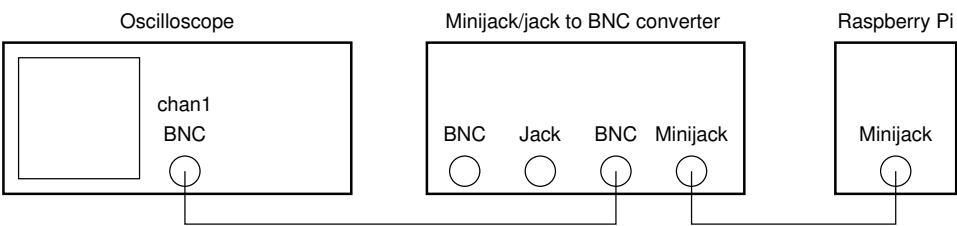


Figure A.1: Test setup with Raspberry Pi output

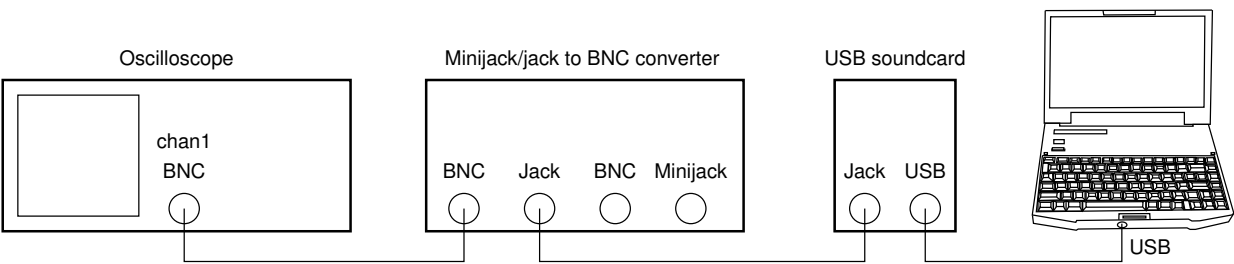


Figure A.2: Test setup with Line 6 USB sound card output

A.1.4 Equipment Used

Instrument	AAU-number.	Manufacture, type etc.
Oscilloscope	64572	Agilent Technologies DSO6034A
Raspberry PI	N/A	E2613RS2V13B1.0
Tone Port UX1 sound card	N/A	Line 6

A.1.5 Test Procedure

To test the frequency response for the two devices, 24 different lossless wave files is generated, which each contains a sinusoid at a different frequency. The wave files were made in Audacity and the generated tones are shown in the following Table A.5.5.

Type	Frequency
Sinusoid	20 Hz
Sinusoid	50 Hz
Sinusoid	100 Hz
Sinusoid	500 Hz
Sinusoid	1 kHz
Sinusoid	2 kHz
Sinusoid	3 kHz
Sinusoid	4 kHz
Sinusoid	5 kHz
Sinusoid	6 kHz
Sinusoid	7 kHz
Sinusoid	8 kHz
Sinusoid	9 kHz
Sinusoid	10 kHz
Sinusoid	11 kHz
Sinusoid	12 kHz
Sinusoid	13 kHz
Sinusoid	14 kHz
Sinusoid	15 kHz
Sinusoid	16 kHz
Sinusoid	17 kHz
Sinusoid	18 kHz
Sinusoid	19 kHz
Sinusoid	20 kHz

The frequency and peak-to-peak voltage is measured for each frequency in Table A.1.8, on an oscilloscope. Furthermore, for each measurement the output from the oscilloscope will be saved to a csv-file for graphing the waveform.

A.1.6 Raspberry Pi Specific

On the Raspberry Pi, the sound levels are set to 0 dB with no gain to avoid clipping. The Raspberry Pi uses a 3,5 mm TRS connector for the sound output, and a 3,5 mm TRS connector to BNC was built to interface it with the oscilloscope, however all measurements was done in mono.

A.1.7 Line 6 USB Sound Card Specific

The Line 6 USB sound card, was tested with a PC, where the sound levels are set to 0 dB and no gain. A 6,35 mm TRS connector to BNC was built to interface with the oscilloscope, however all measurements was done in mono.

A.1.8 Results

The raw results of the Raspberry Pi compared to the line 6 USB sound card, see the Table A.1.8 table.

Input frequency	V _{pp} (RPI)	Measured frequency (RPI)	V _{pp} (USB)	Measured frequency (USB)
20 Hz	1,31 V	20 Hz	7,06 V	20 Hz
50 Hz	1,3 V	50 Hz	7,06 V	50 Hz
100 Hz	1,33 V	101 Hz	6,88 V	100 Hz
500 Hz	1,39 V	500 Hz	6,75 V	500 Hz
1000 Hz	1,41 V	1000 Hz	6,75 V	1000 Hz
2000 Hz	1,38 V	1980 Hz	6,81 V	2000 Hz
3000 Hz	1,42 V	2999 Hz	6,81 V	3000 Hz
4000 Hz	1,39 V	3970 Hz	6,75 V	4000 Hz
5000 Hz	1,47 V	4930 Hz	6,75 V	5000 Hz
6000 Hz	1,25 V	5999 Hz	6,75 V	6000 Hz
7000 Hz	1,27 V	7040 Hz	6,81 V	7000 Hz
8000 Hz	1,38 V	8130 Hz	6,81 V	8000 Hz
9000 Hz	1,25 V	8997 Hz	6,75 V	9000 Hz
10 000 Hz	1,23 V	10 200 Hz	6,88 V	10 000 Hz
11 000 Hz	1,25 V	11 050 Hz	6,88 V	11 000 Hz
12 000 Hz	1,26 V	12 050 Hz	6,81 V	12 000 Hz
13 000 Hz	1,26 V	13 158 Hz	6,94 V	13 000 Hz
14 000 Hz	1,26 V	14 050 Hz	6,88 V	14 000 Hz
15 000 Hz	1,25 V	14 780 Hz	6,88 V	15 000 Hz
16 000 Hz	1,35 V	16 129 Hz	6,75 V	16 000 Hz
17 000 Hz	1,27 V	17 519 Hz	6,88 V	17 000 Hz
18 000 Hz	1,29 V	18 182 Hz	6,94 V	18 000 Hz
19 000 Hz	1,39 V	19 231 Hz	6,81 V	19 000 Hz
20 000 Hz	1,25 V	20 000 Hz	6,75 V	20 000 Hz

A.1.9 Comparison of Output Waveforms

In the following the output waveform from the Raspberry Pi and the USB sound card are compared. The comparison is made at fourth different frequencies to get an overview of the differences. Furthermore the measurements are normalized and compared with the reference signal as well. The measurement waveforms have been made for every frequencies represented in the Table A.1.8 above and can be found on the appendix CD.

 /Graphs and Figures/rpi_frequency_output/data

A.1.10 Conclusion

From the raw measuring results in Table A.1.8 it is shown, that the RPI sound card and the USB sound card individually are very accurate both in the frequency and in the V_{pp} output at every frequency. But when taking a look at the graph comparisons especially from 4000 kHz and at higher frequencies, it can be seen that the sine waveform is getting more and more sawtooth like compared to the USB sound card. The waveform from the RPI sound card is getting more and more ugly and sawtooth like the higher the frequency, which means that the RPI sound card is not suitable for this project.

A.1.11 Uncertainty of Measurements

In this case the small calculation and reading uncertainties of the raw measuring results is not important to the conclusion, which were made by looking at the waveform comparisons. The raw measuring results were made to see if there were any big changes in the frequency and V_{pp} values for each sound card and therefore the small uncertainties in the calculation and reading is not important to get the overall picture.

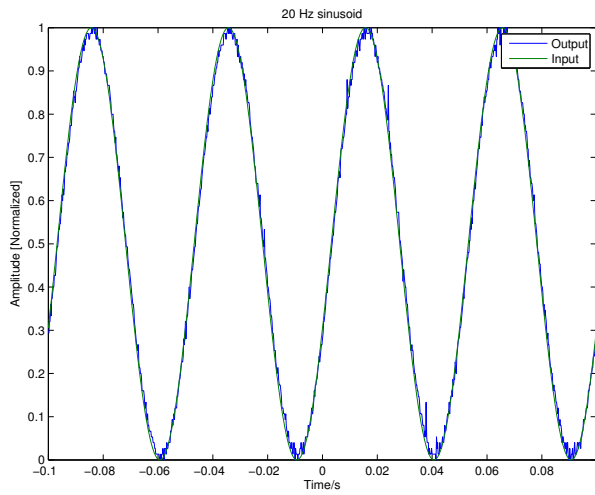


Figure A.3: The waveform at 20 Hz—Raspberry Pi

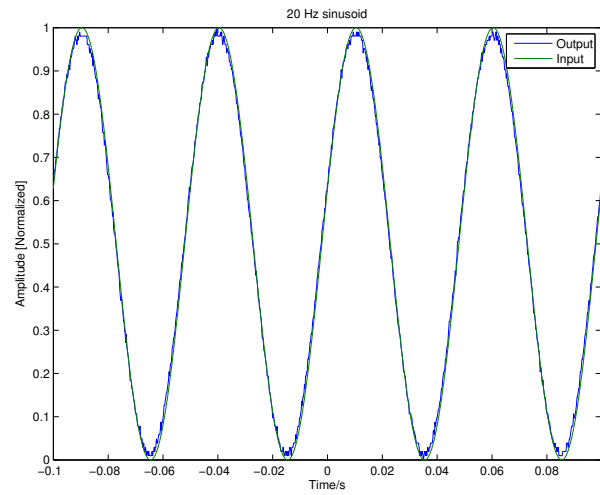


Figure A.4: The waveform at 20 Hz—USB sound card

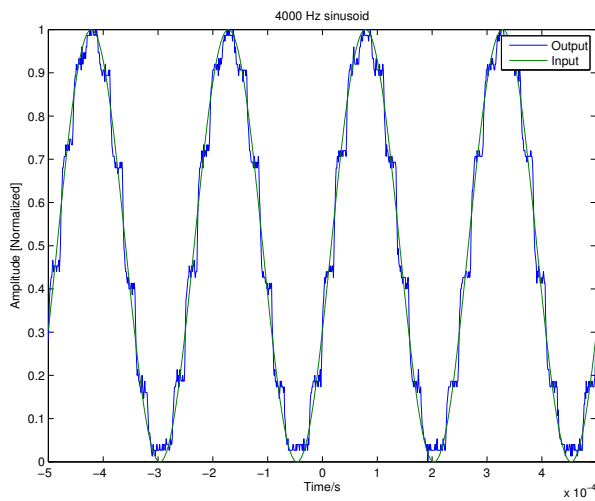


Figure A.5: The waveform at 4000 Hz—Raspberry Pi

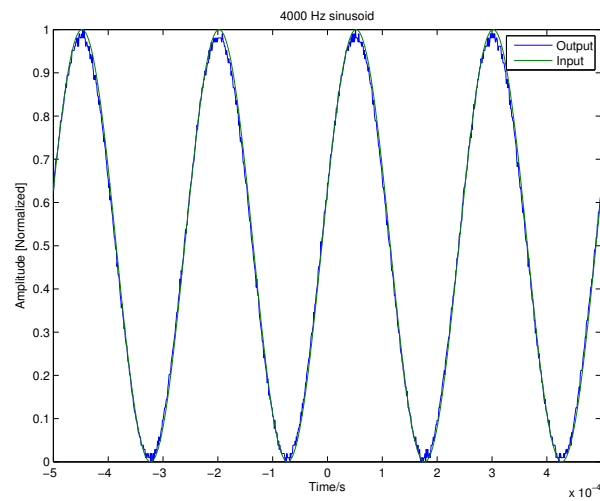


Figure A.6: The waveform at 4000 Hz—USB sound card

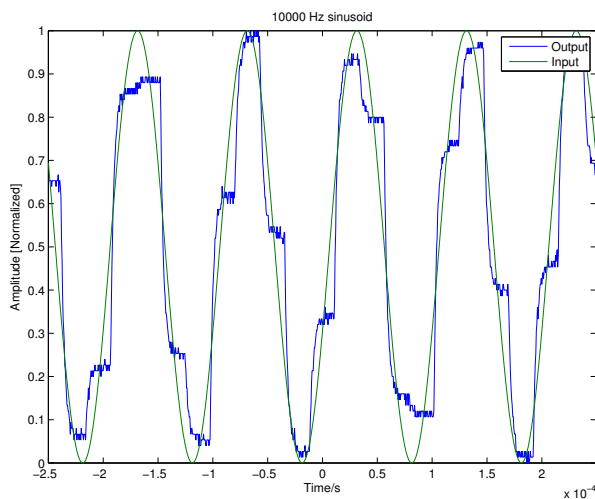


Figure A.7: The waveform at 10000 Hz—Raspberry Pi

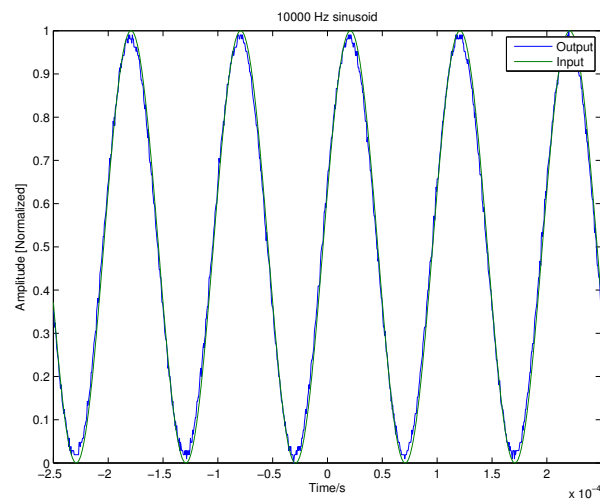


Figure A.8: The waveform at 10000 Hz—USB sound card

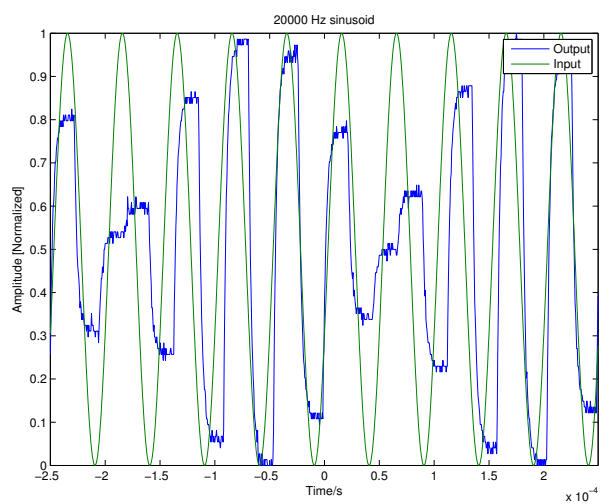


Figure A.9: The waveform at 20000 Hz—Raspberry Pi

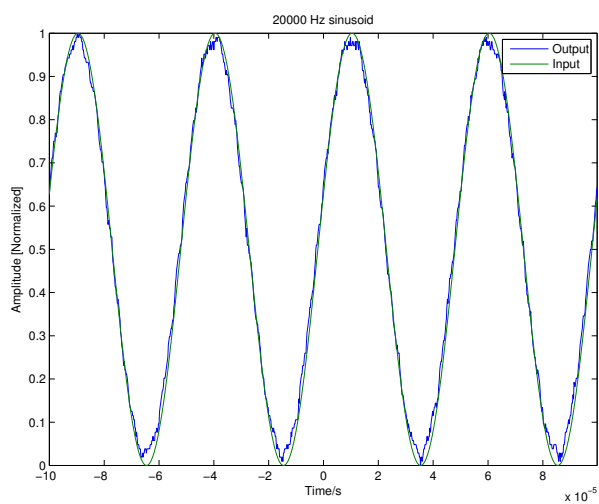


Figure A.10: The waveform at 20000 Hz—USB sound card

A.2 Measurements of 6th Order Butterworth Band Pass Filters' Frequency Response

A.2.1 Purpose

The purpose of this measurement is to determine, if the `butt6bp()` filters fulfills the requirements.

A.2.2 Test Objects

- `main.c`, `butt6bp.c`, `butt6bp.h`, `mypcm.h`.

 `/filter_design/test_alsa/`

A.2.3 Test Setup

The test setup is split into a calibration setup and a measuring setup. These are seen in the Figures A.11 and A.12.

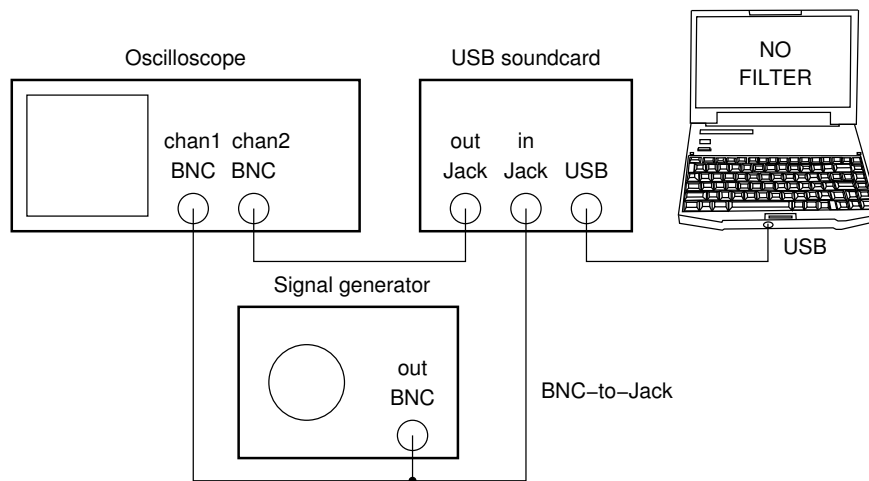


Figure A.11: Calibration setup for making sure the input and output has the same amplitude if no filter is applied.

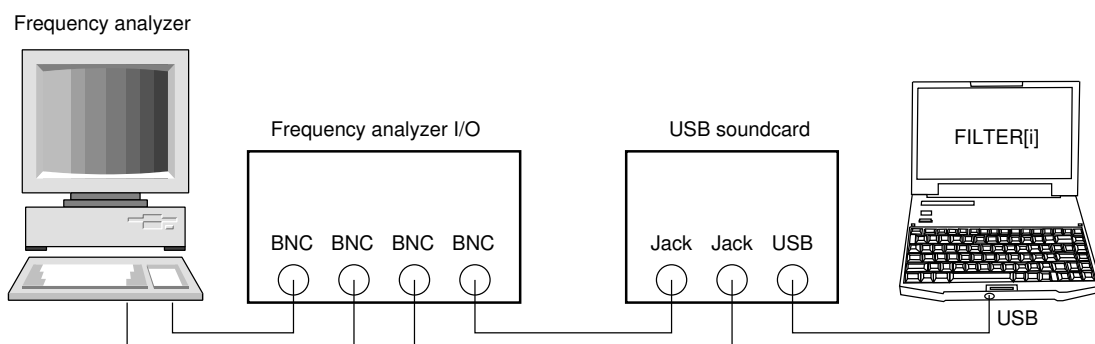


Figure A.12: Setup for doing frequency analysis of the filters. The measurement is done 16 times, to get through all filters, one at a time and at last to get the frequency response if all filters are set to unity gain.

A.2.4 Equipment Used

Instrument	AAU-number.	Manufacture, type etc.
Oscilloscope	33875	Agilent 54621A
Sound card	78377	Edirol USB AudioCapture UA-25EX
Frequency Analyzer		DAQmx Swept Sine

A.2.5 Test Procedure

Calibration

1. The filter program is set to pass the input signal directly to the output.
2. The signal generator is set to an output of 500 mV amplitude, 1000 Hz. This is done by observing the oscilloscope.
3. The sound card's gain control is turned up as much as possible without the peak LED lighting up.
4. The sound card's output control is set so the output is 500 mV amplitude. This is also done by observing the oscilloscope.

Frequency Analysis Measurement

1. The frequency analyzer is set to an output of 500 mV amplitude, with a sweep from 10–44 100 Hz, recording 400 data points.
2. The sweep is performed on a tested filter with unity gain, and zero gain on all other filters. The output amplitudes characteristics are plotted.
3. The ideal Class 2, 1/3 octave filters are plotted on top.
4. 2 and 3 are done for each filter from 0–14.
5. All filters are applied with unity gain. The sweep is performed and the output amplitude characteristic is plotted.

A.2.6 Results

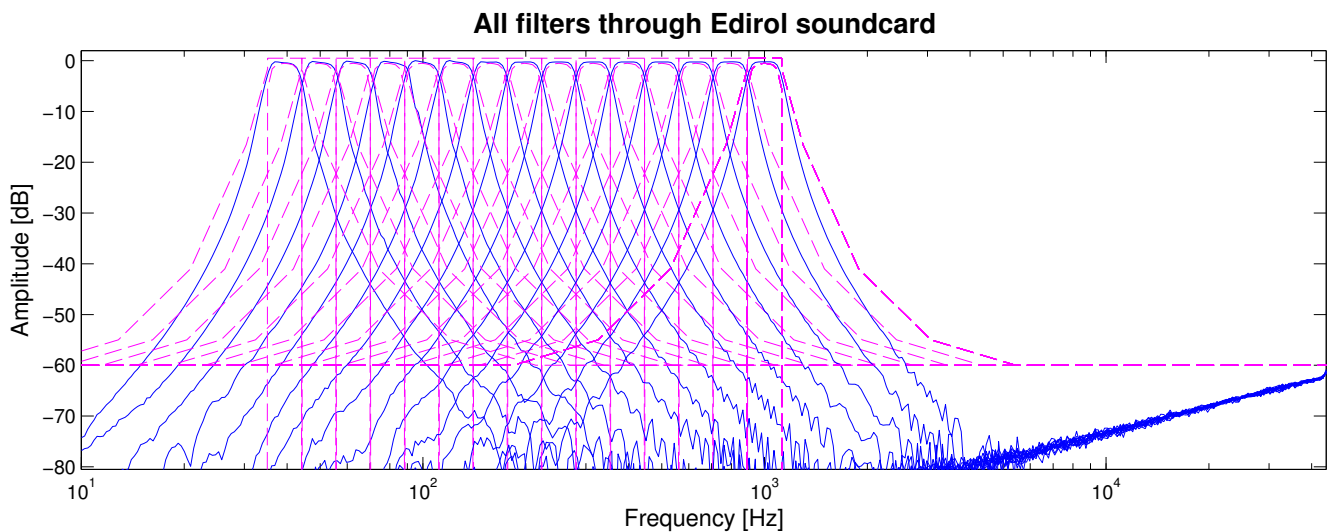


Figure A.13: All filters measured and compared to DS/EN 61260

A.2.7 Conclusion

The measurements in Figure A.14 shows that all filters, except one, is compliant with the DS/EN 61260. The filter with center frequency at 79 Hz, is somehow outside the required limits. It can be concluded that the filter design withholds Requirement 6 and with a little effort the error should be fixable by adjusting the coefficients of the specific filter.

A.2.8 Uncertainty of Measurements

- The sound card indicated a peak in the peak LED, with a sine wave of 0,5 V and with the compressor/limiter switched off. The peak LED showed no emission with a sine input of 50 mV. Anyhow the amplitude response with 50 mV is similar to Figure A.14, as can be seen on Figure A.16.

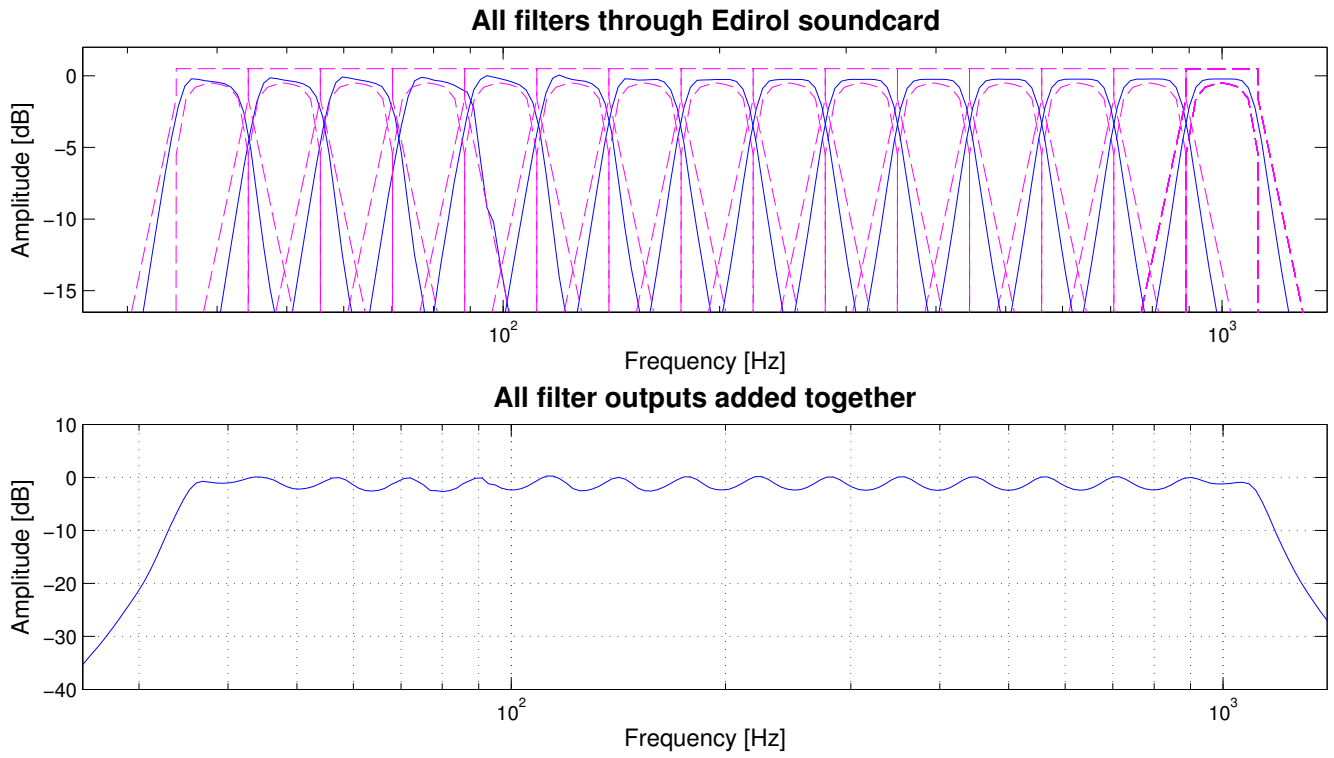


Figure A.14: Measurement results of the designed filters in the band from 40 Hz to 1 kHz. Each of the filters in the upper plot is a separate measurement. The bottom plot is measured through an addition of all bp-filters.

- The cables that was used during this measurement was not of best quality and might have introduced a small amount of noise. This has however, not been measured.

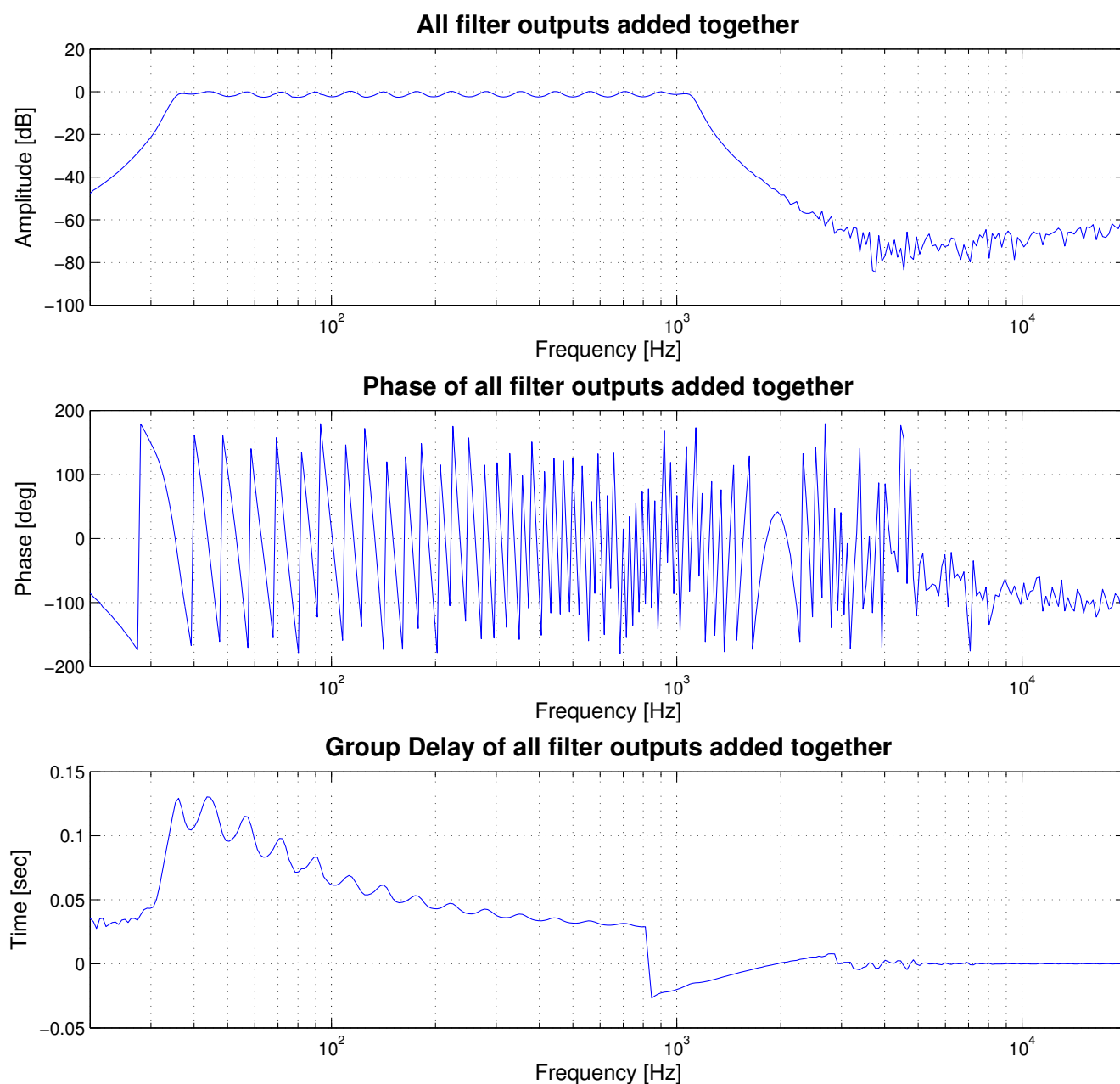


Figure A.15: Measurement results of all the filters applied. Amplitude in the top. Phase in the middle. Calculated Group Delay in the bottom.

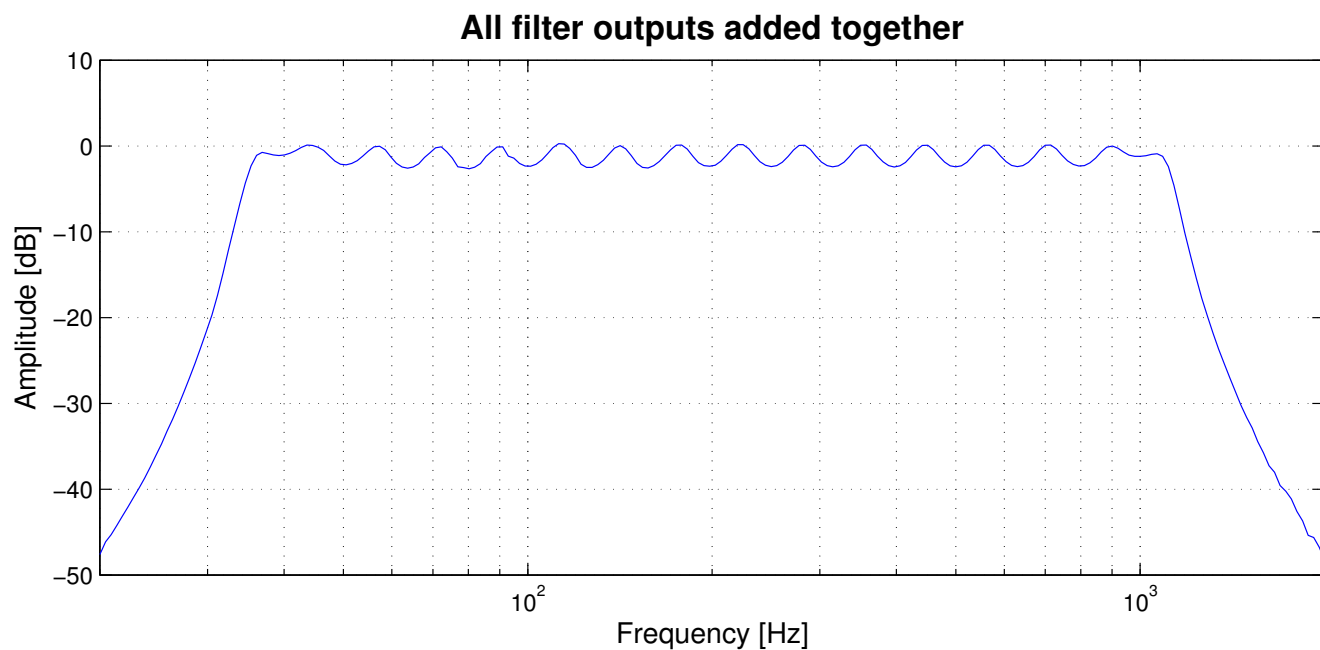


Figure A.16: All filters with input of 50 mV

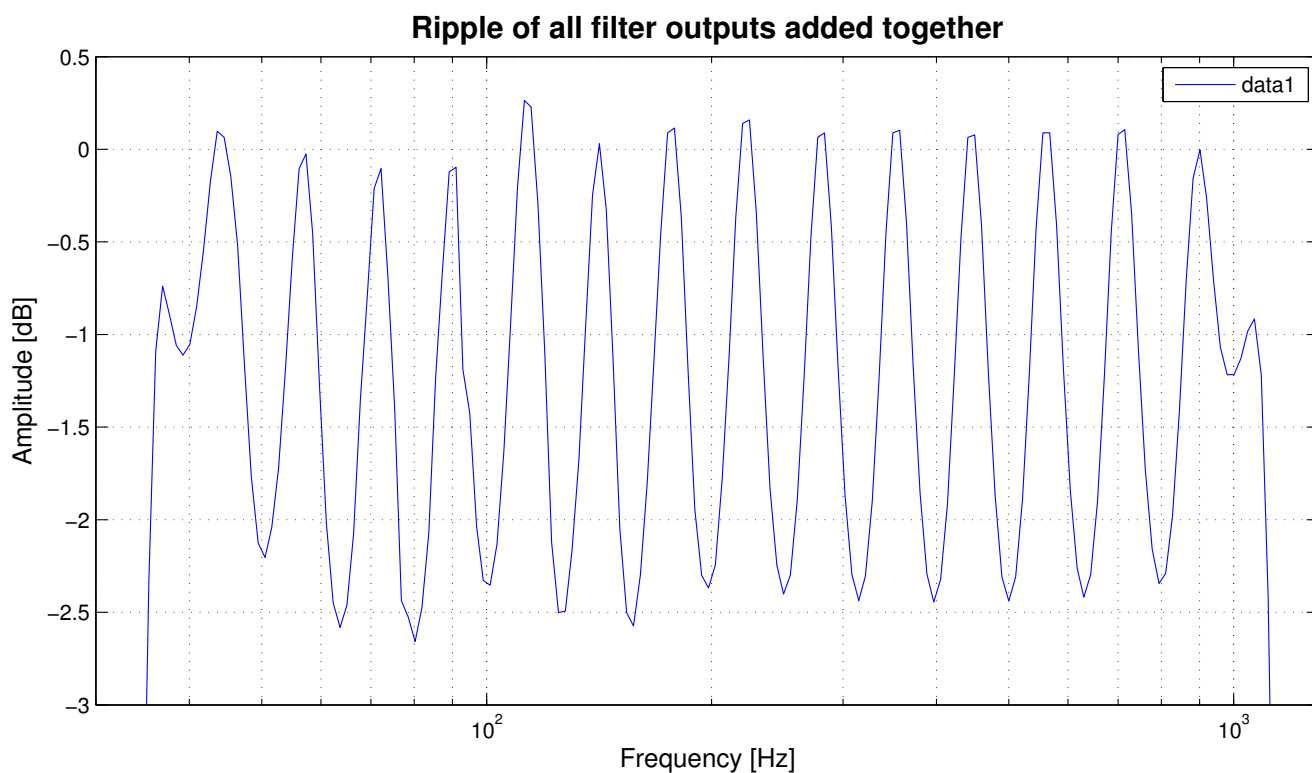


Figure A.17: Ripple of all bp-filter outputs added together.

A.3 Implementation of Treble Filter

To cover the treble area, a static filter is needed which is not suppose to be controlled from the main program. Firstly it is decided to reuse the cascaded biquad C-implementation algorithm. Therefore an obvious 3rd order Butterworth band pass filter is developed with corner frequencies at 1225 Hz and 20 kHz.

A.3.1 3rd Order Band Pass Filter

The design followed the procedure described in Section 8.4. The only difference was the pairing of poles and zeros was different as shown in Figure A.19. The result is shown amongst others in Figure A.18. All the measurements in this section has been carried out by using the same procedure as described in Sec. A.2.

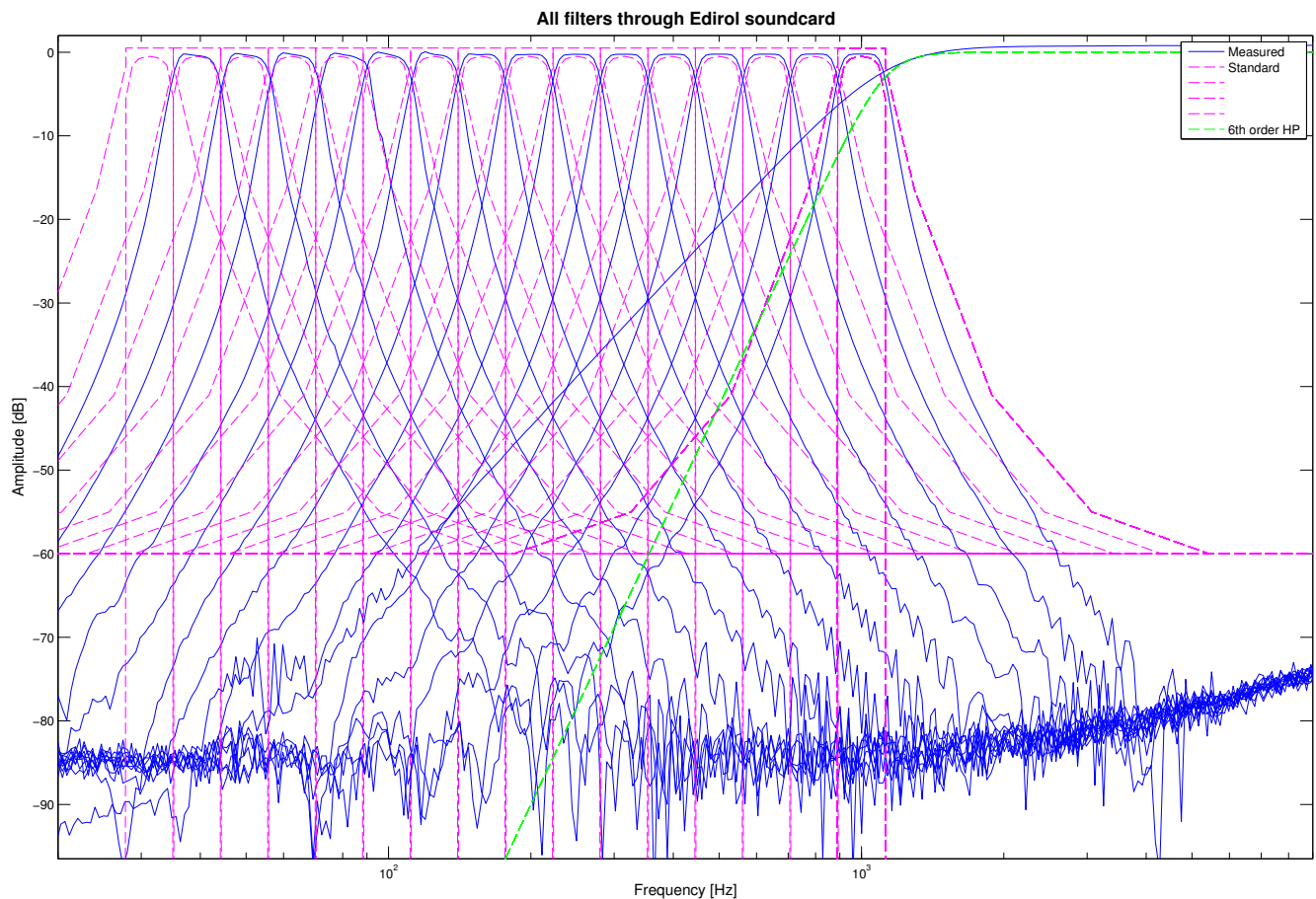


Figure A.18: Implementation of 3rd order BP-filter from 1225 kHz to 20 kHz. All measurements are shown in blue. In green is a calculated 6th order HP-filter.

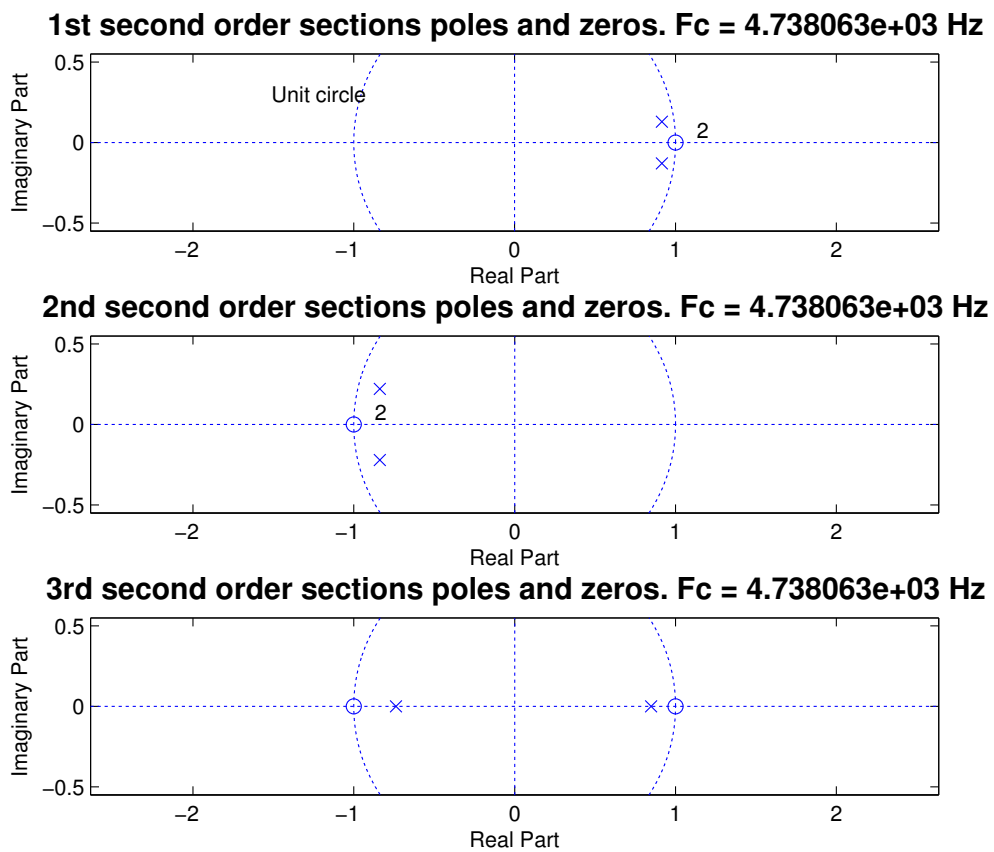


Figure A.19: Pairing of poles and zeros for the treble filter which was designed as a band pass filter.

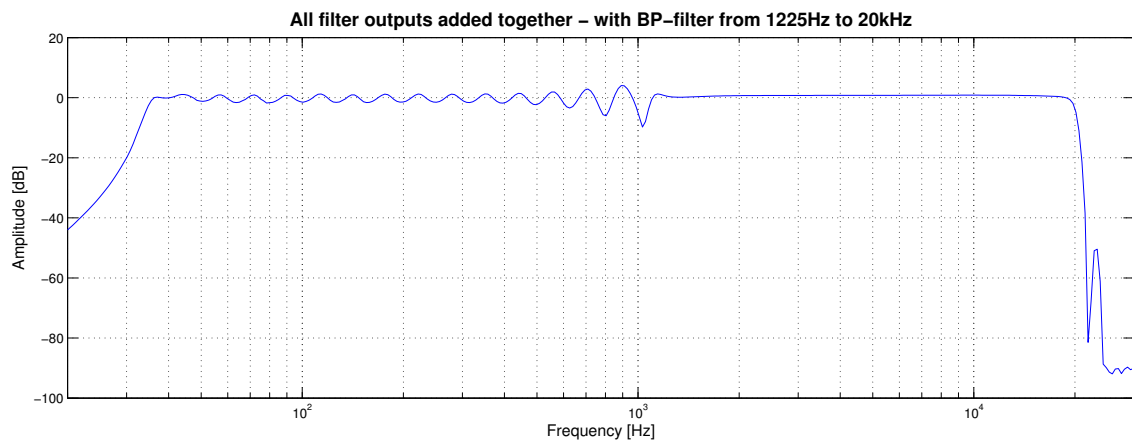


Figure A.20: Implementation of a high 3rd order band pass filter.

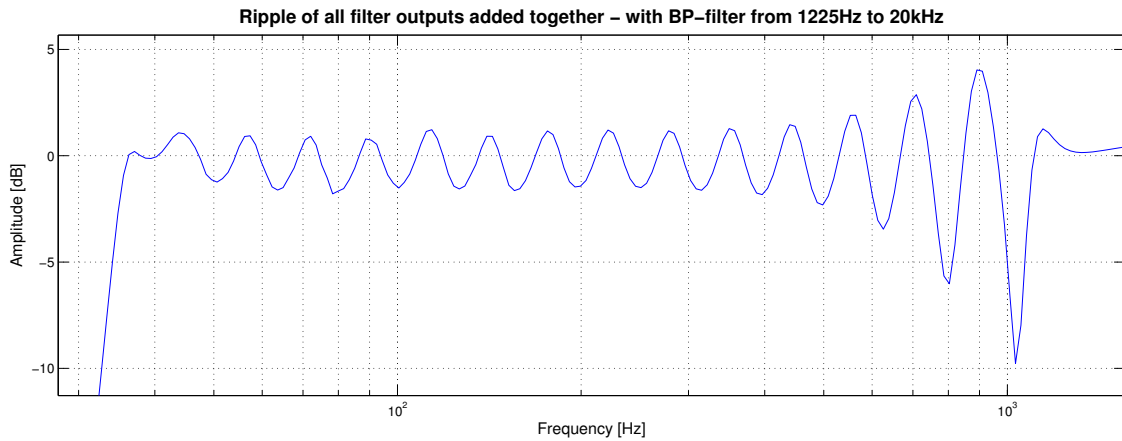


Figure A.21: Ripple in the implementation of a 3rd order band pass filter from 1225 Hz to 20 kHz is app. 14 dB.

From measuring the frequency response of the implementation of a 3rd order band pass filter, it is concluded that a better ripple is needed. From Figure A.20 it is noted that the sound card has a built in anti aliasing filter, therefore the next approach is to use all the 6 delays in the C-algorithm for implementing a 6th order HP-filter from 1225 Hz.

A.3.2 6th Order High Pass Filter

An implementation of a 6th order HP-filter with a cut off frequency at 1225 Hz shows that an improved ripple is obtained, compared the 3rd order band pass filter in Section A.3.1. Therefore this design is chosen as a satisfying solution, when not using 30 bands, or another algorithm with a higher order than 6. In this design it is then decided to move the cut off frequency upwards in steps of 10 Hz, execute the C-program, and then measure the ripple again, to see if the ripple will improve in the measurements.

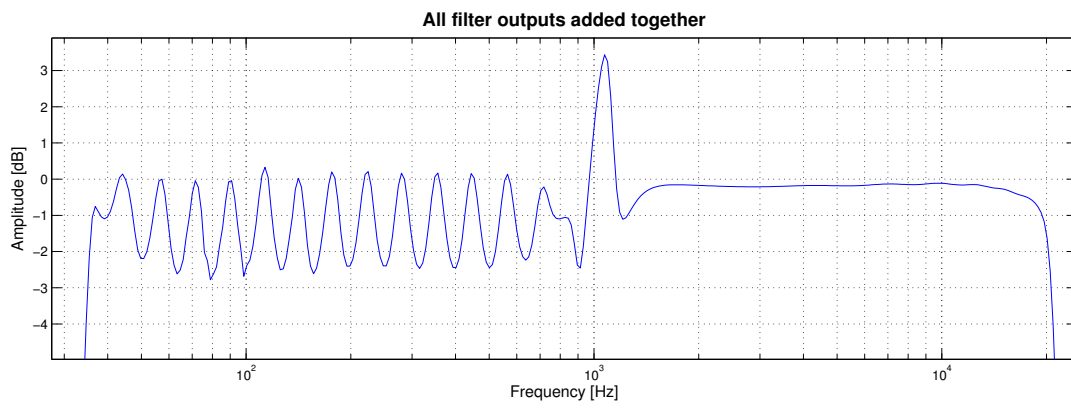
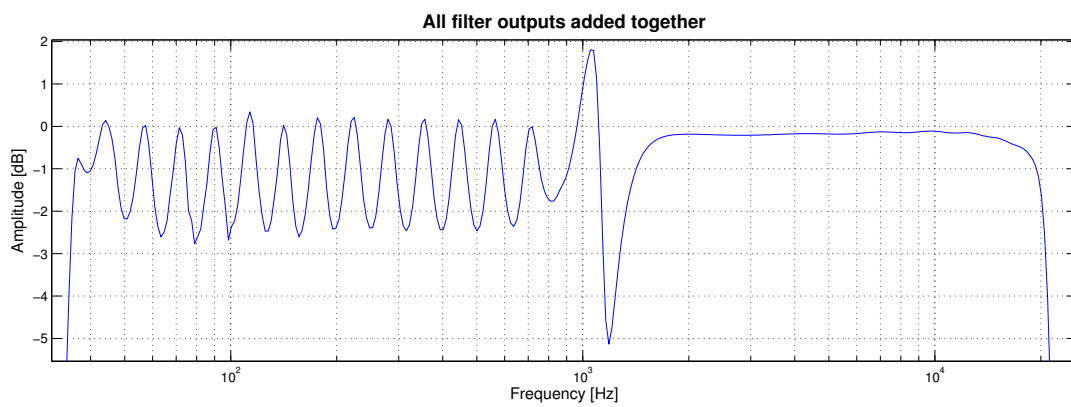
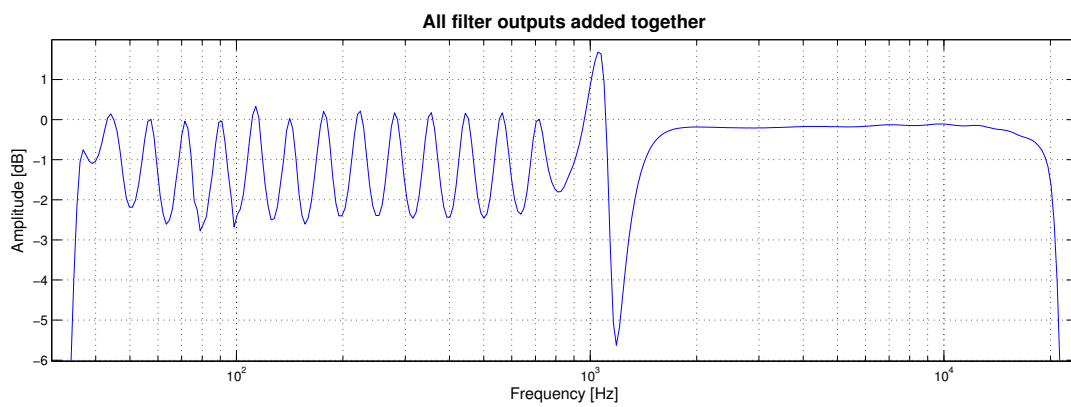
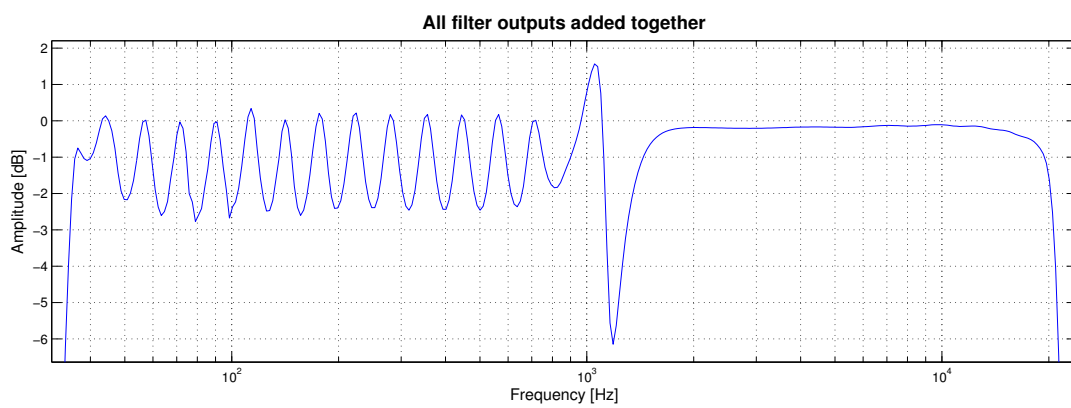


Figure A.22: Ripple of HP-filter with cutoff at different frequencies.

Because it is preferred to have a smaller ripple especially upwards as a peak, it is decided to implement the high pass, with a cut off frequency at 1235 Hz, shown in Figure A.22b. This filter has only 2 dB peak and is an improvement. The next approach is to lower both the gain factor of the high pass, so that it fits the mean value of the pass band from 40 Hz to 1 kHz and in the same approach, lower the 1 kHz BP-filter gain to further improve the filter frequency response ripple, and especially to keep the filter from clipping, preventing it from peaking at 0 dB. The results are shown in Figure A.23c in Section A.3.3

A.3.3 Improving the 1235 High Pass Ripple

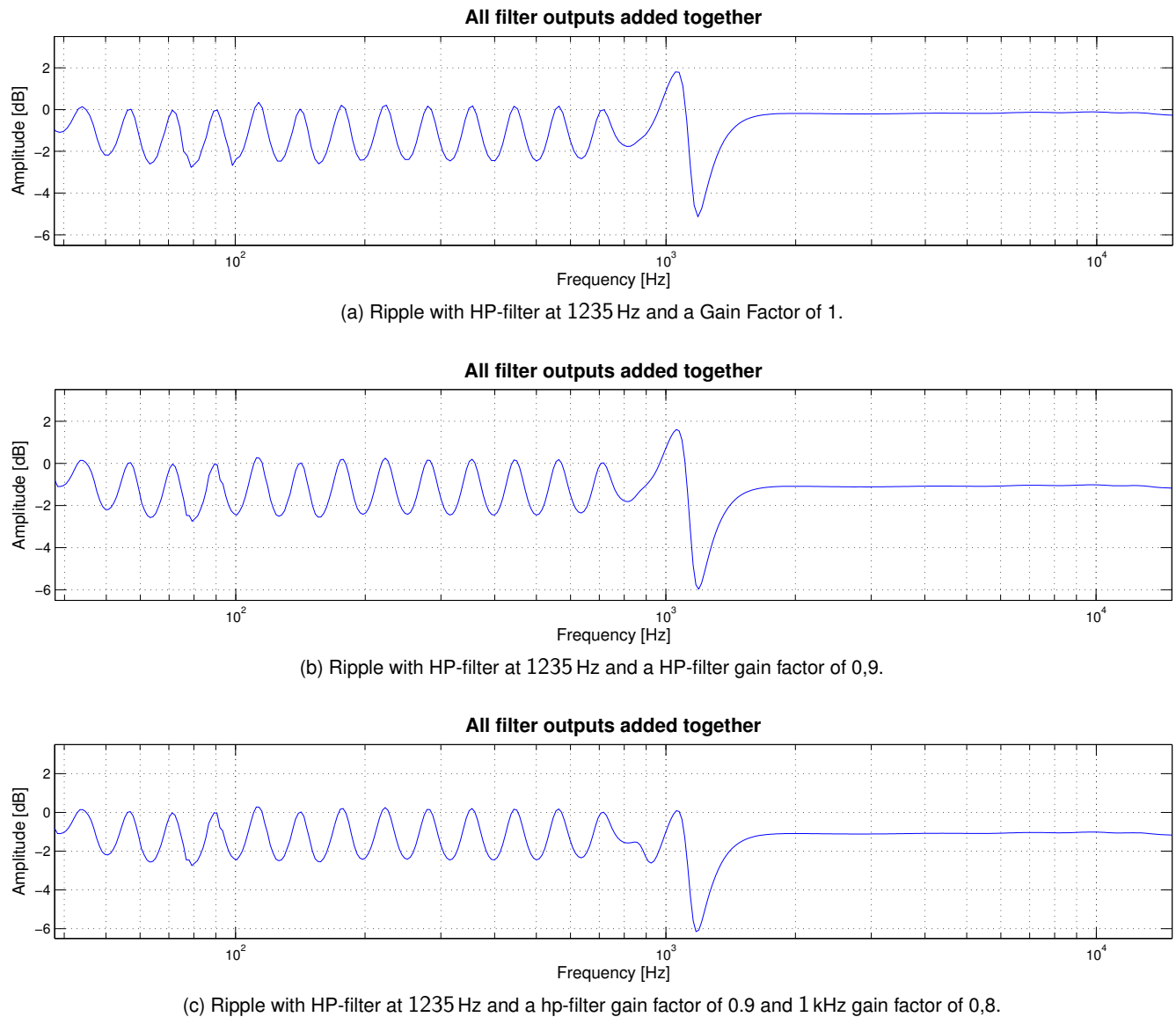


Figure A.23: Reduction of ripple by adjusting gain factors for the upper BP-filter and the HP-filter.

It is now concluded to design the initial settings for the filter with the hp-filter with a cut off frequency 1235 Hz and a gain factor of max 0,9 and the 1 kHz band is allowed to max have a gain factor of 0,8. Now the range of attenuation will be further investigated by measuring this concluded filter setup.

A.3.4 Attenuation Range

It must now be investigated how much the designed filter is able to attenuate at a given frequency. Therefore some test will be made by setting filters at different gain factors and measuring the frequency response on the output.

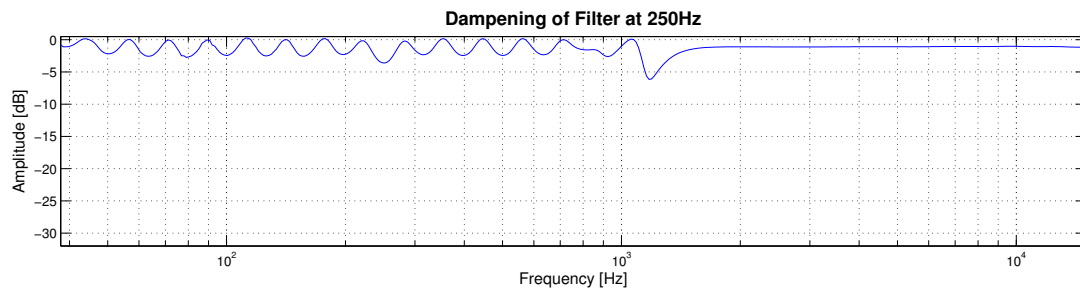
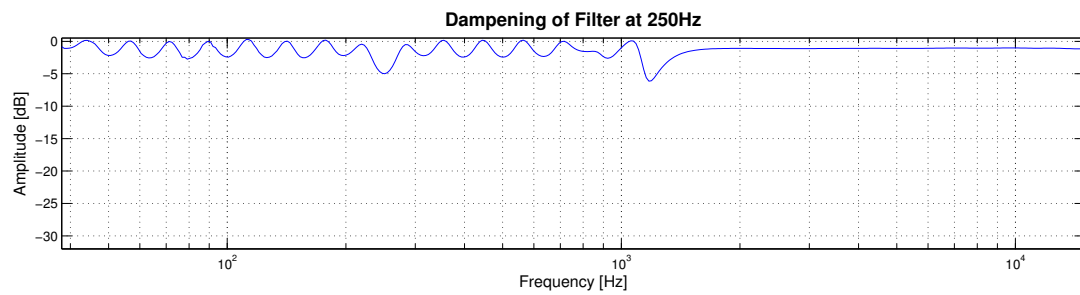
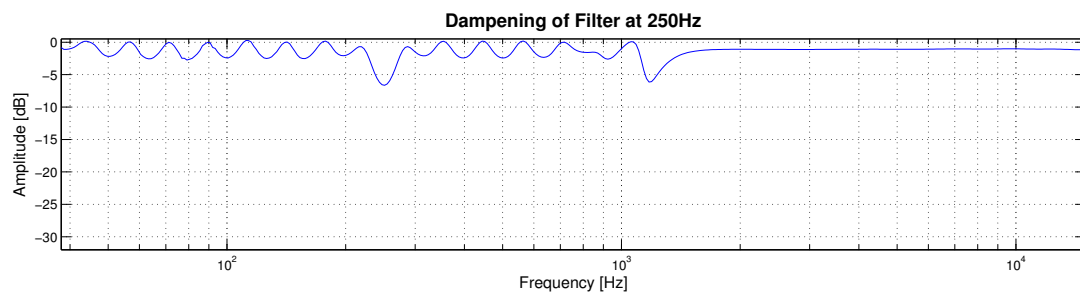
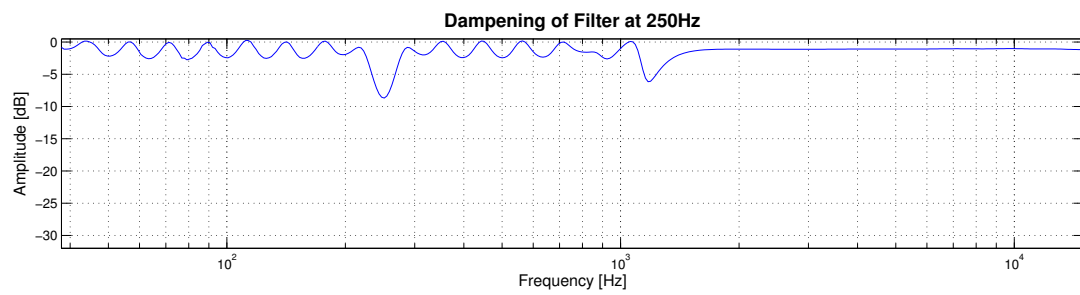
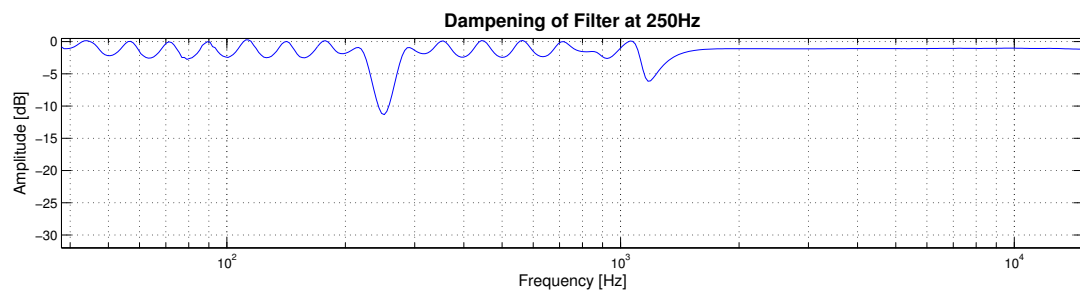


Figure A.24: The 250 Hz filters attenuation at different gain factors.

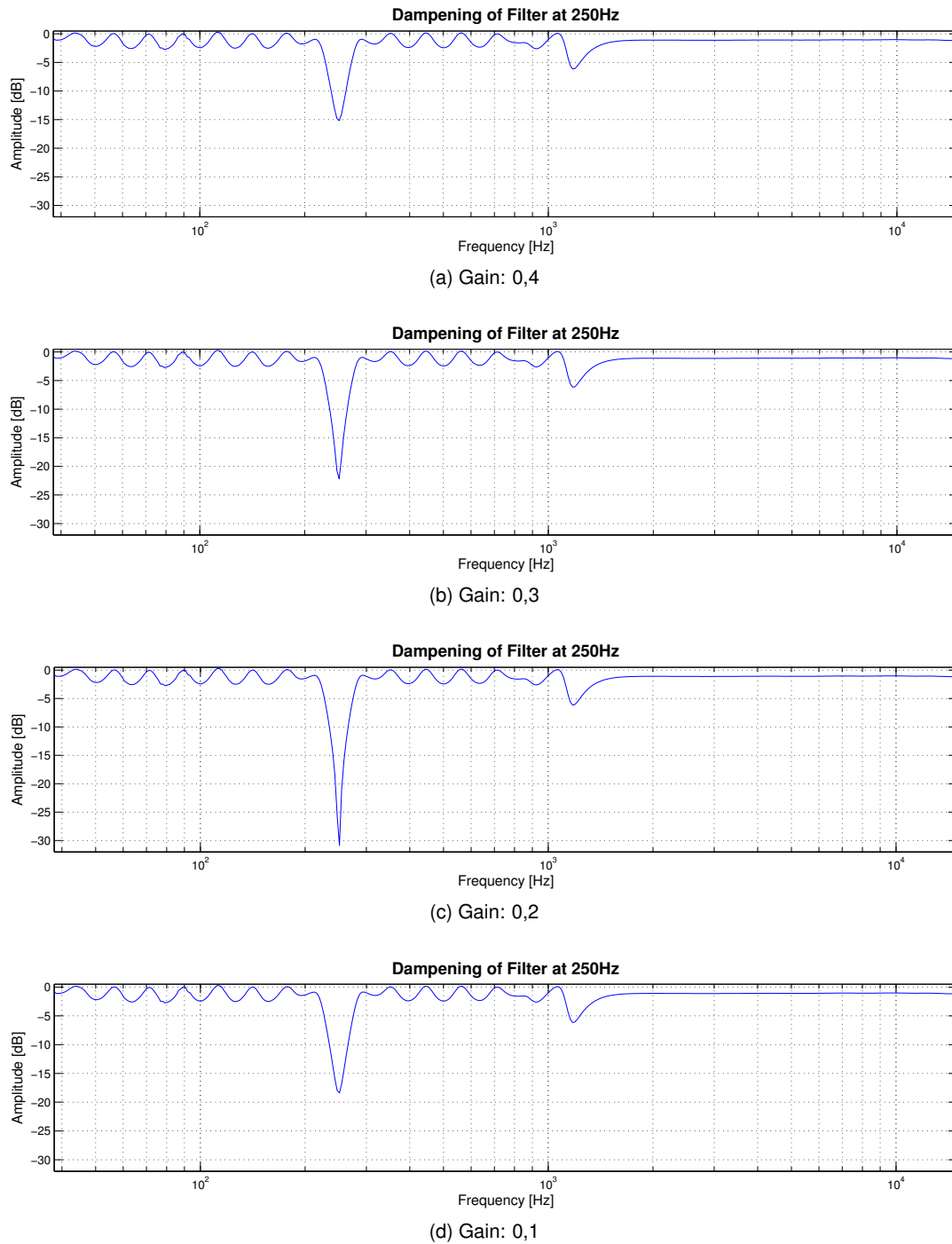


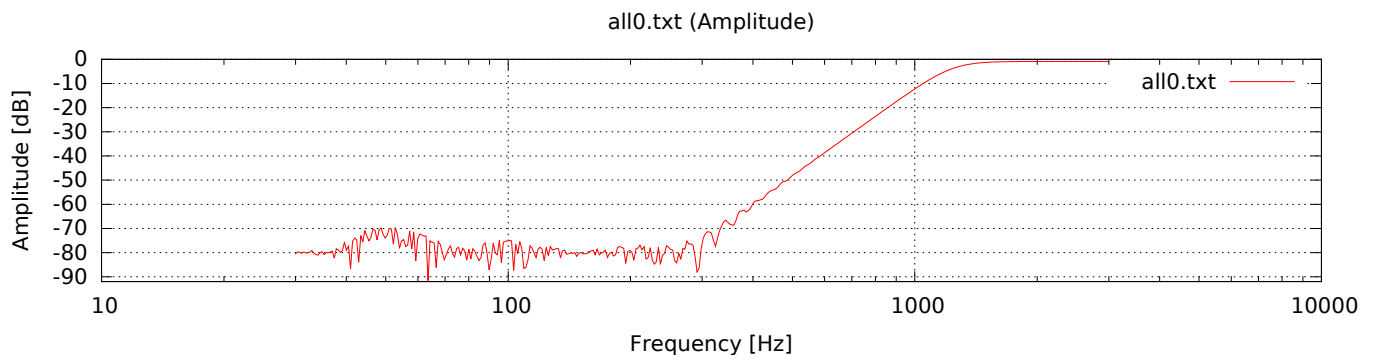
Figure A.25: (Continued) The 250 Hz filters attenuation at different gain factors.

By doing this research it is seen that somewhere between a gain factor of 0,2 and 0,1 the calculations flip over so that the attenuation becomes negative. More measurements have been carried out to determine how much attenuation can be obtained when

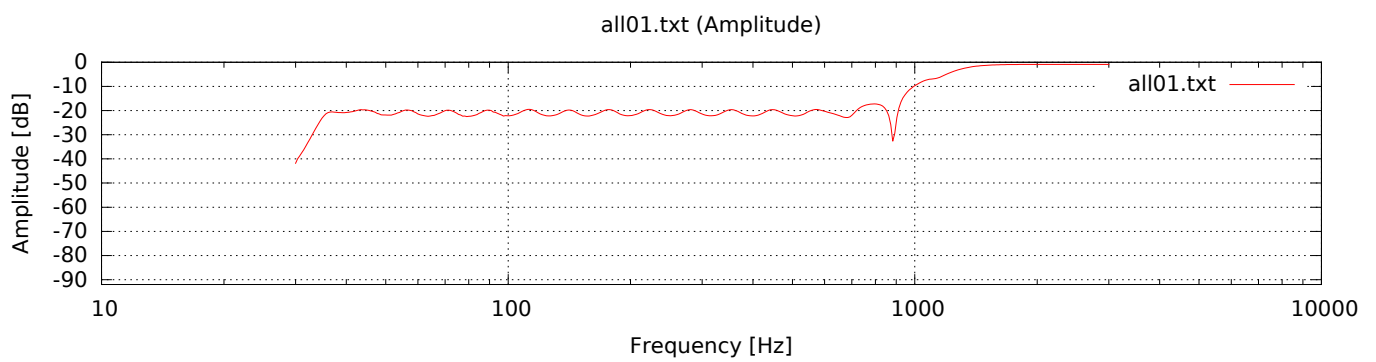
- All band pass gains are 0–0,3.
- Every second gain is 1 and the others are 0–0,3.
- Every second gain is 0–0,3 and the others are 1 (reversed).
- Two middle gains are 0–0,3, the others are 1.
- Four middle gains are 0–0,3, the others are 1.

- The two outermost gains are 0–0,3, the others are 1.

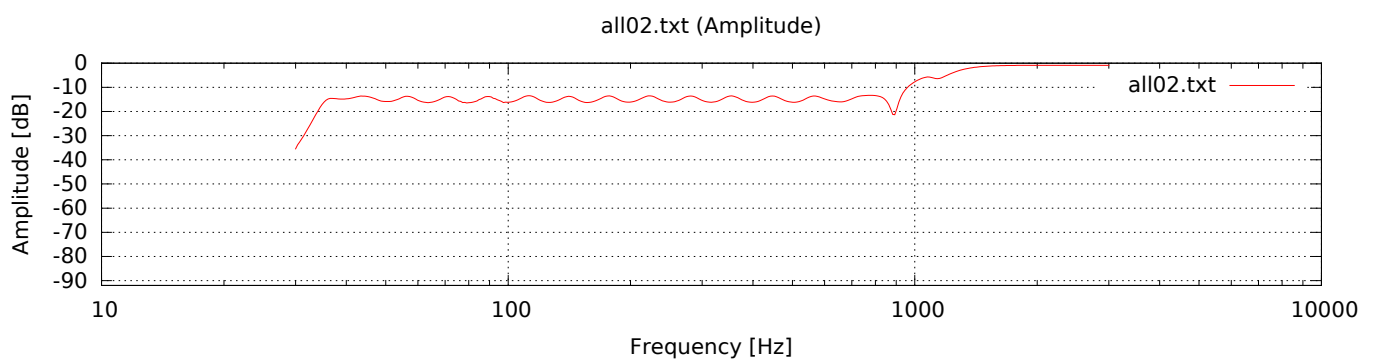
These measurements are shown in the Figures: A.26, A.27, A.28, A.29, A.30, and A.31. From these it can be seen that gains at below 0,2 generally gives *less* attenuation than at 0,2, which is unexpected.



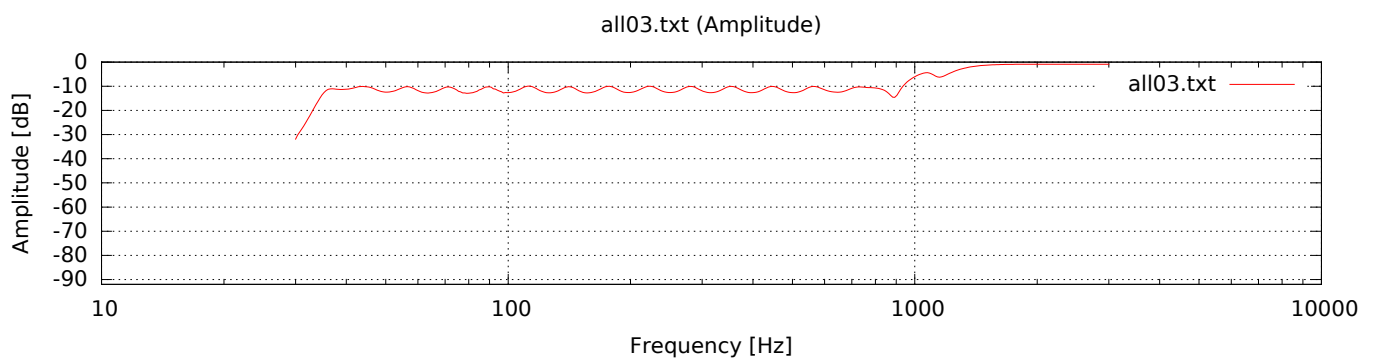
(a) Gain: 0



(b) Gain: 0,1

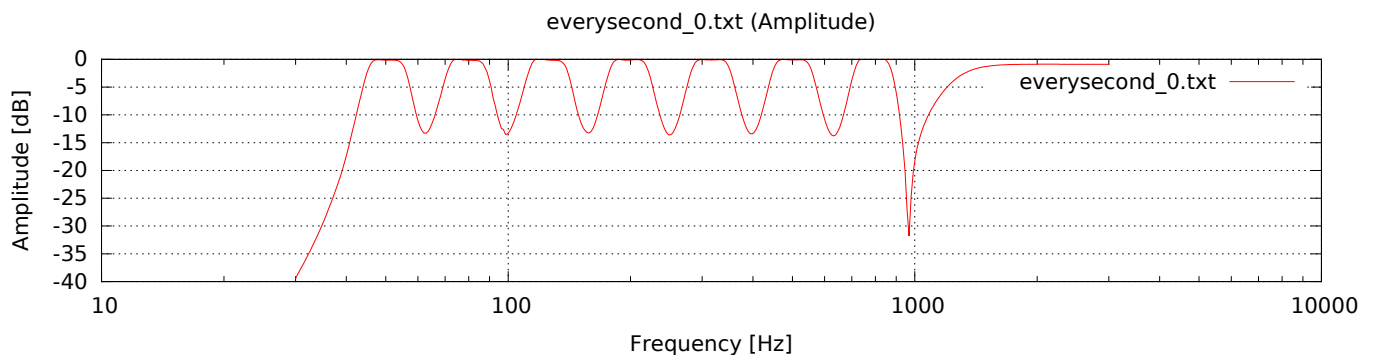


(c) Gain: 0,2

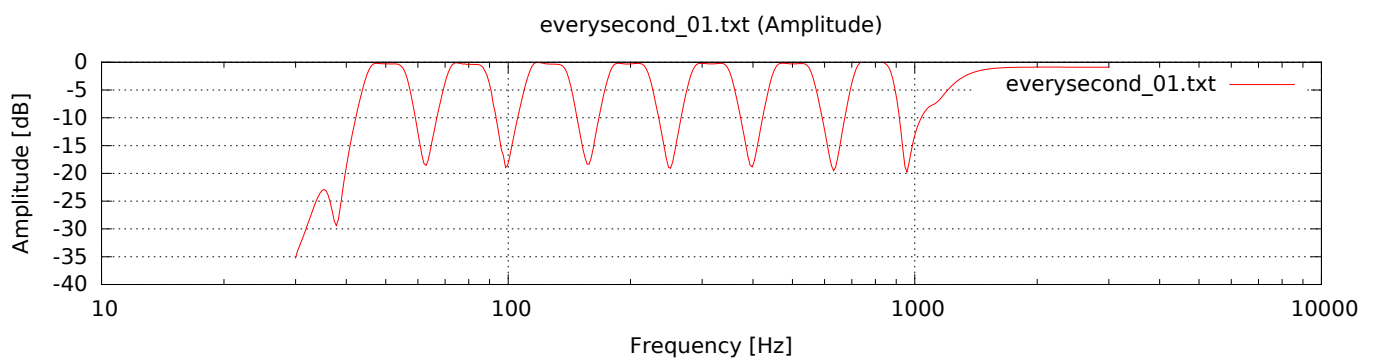


(d) Gain: 0,3

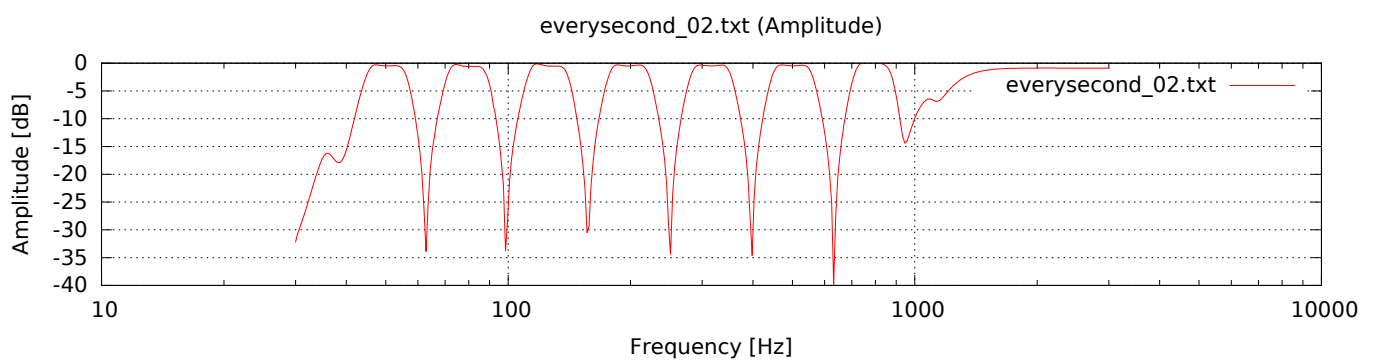
Figure A.26: All filter gains are varied.



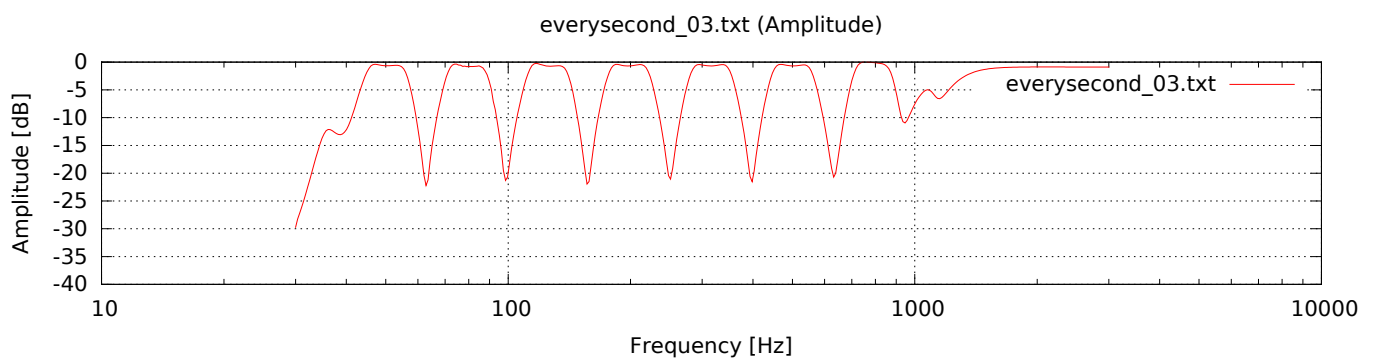
(a) Gain: 0



(b) Gain: 0,1

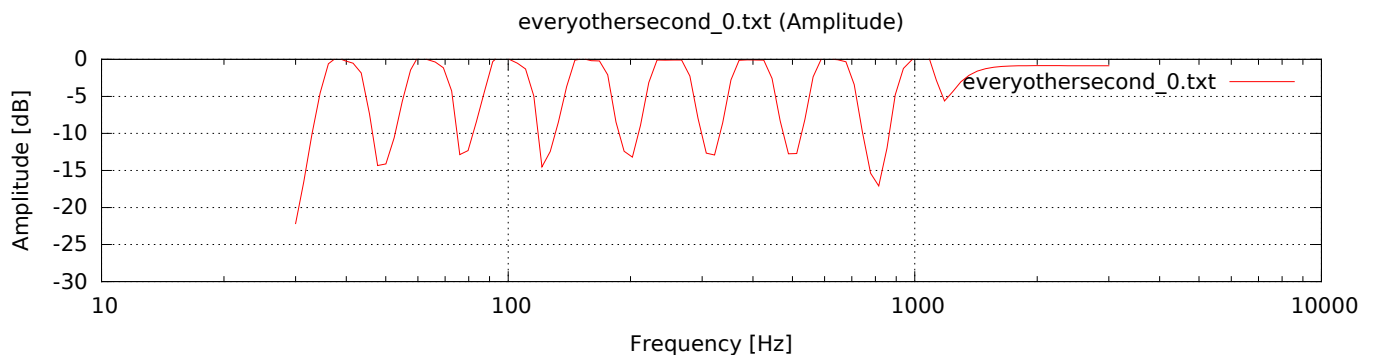


(c) Gain: 0,2

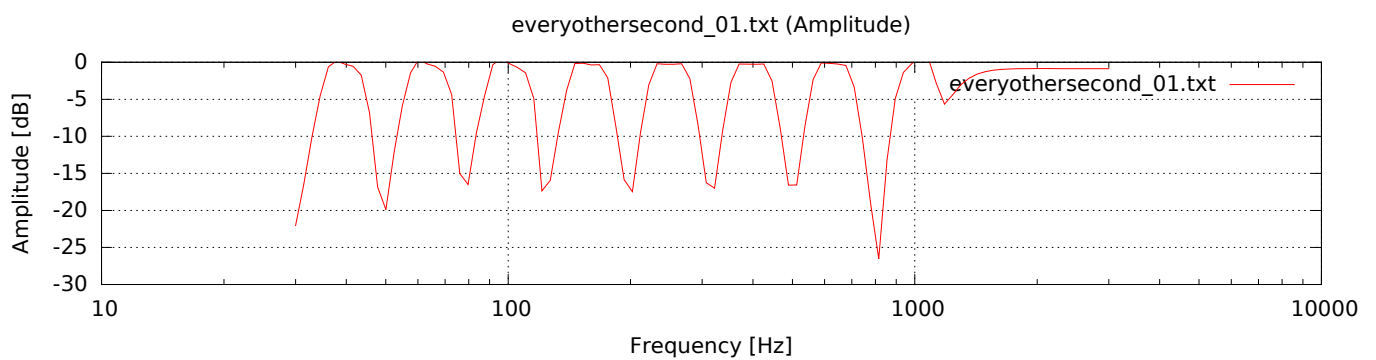


(d) Gain: 0,3

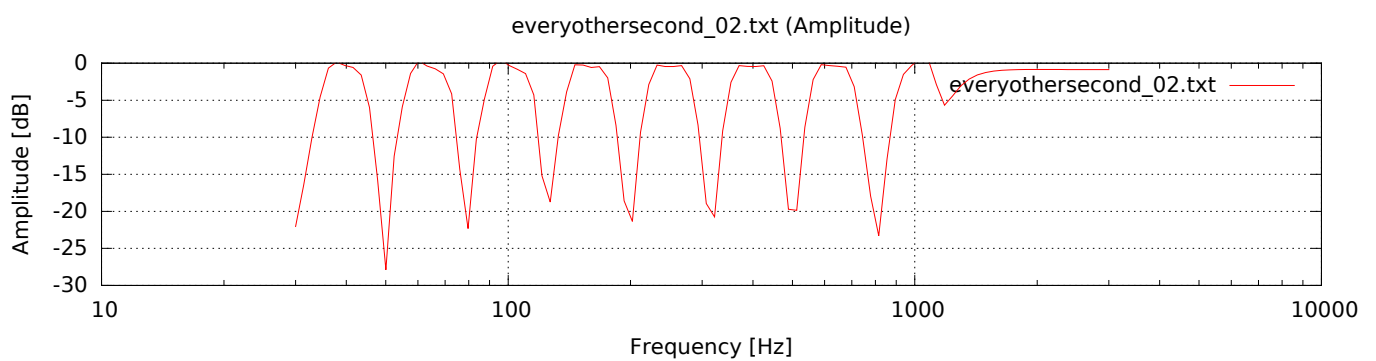
Figure A.27: Every second filter gains are varied.



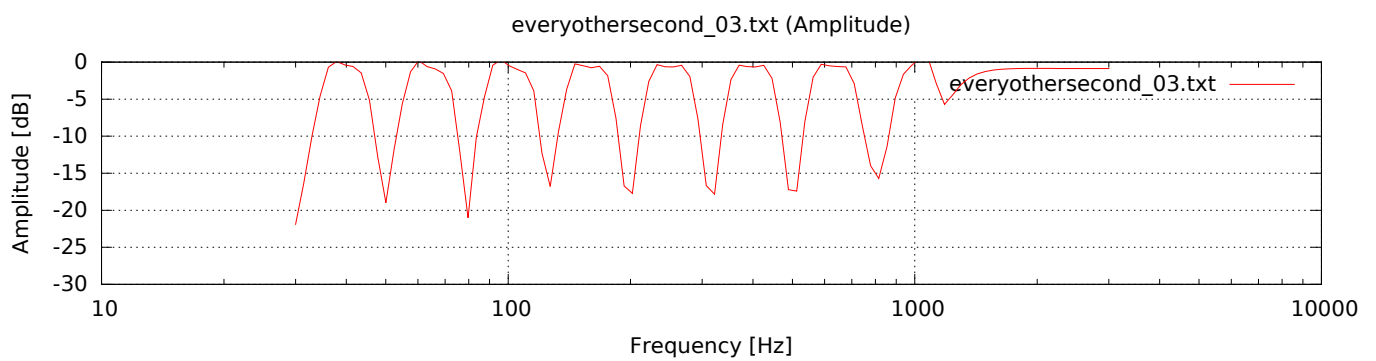
(a) Gain: 0



(b) Gain: 0,1

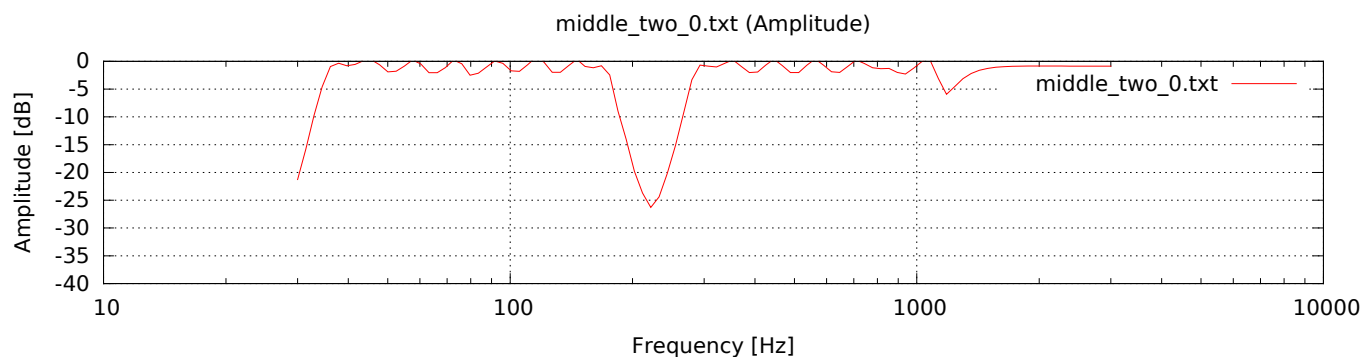


(c) Gain: 0,2

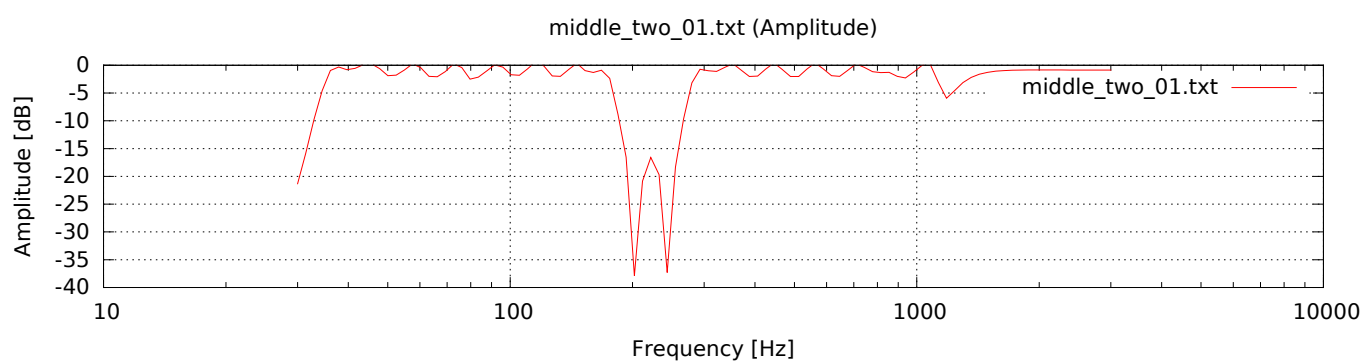


(d) Gain: 0,3

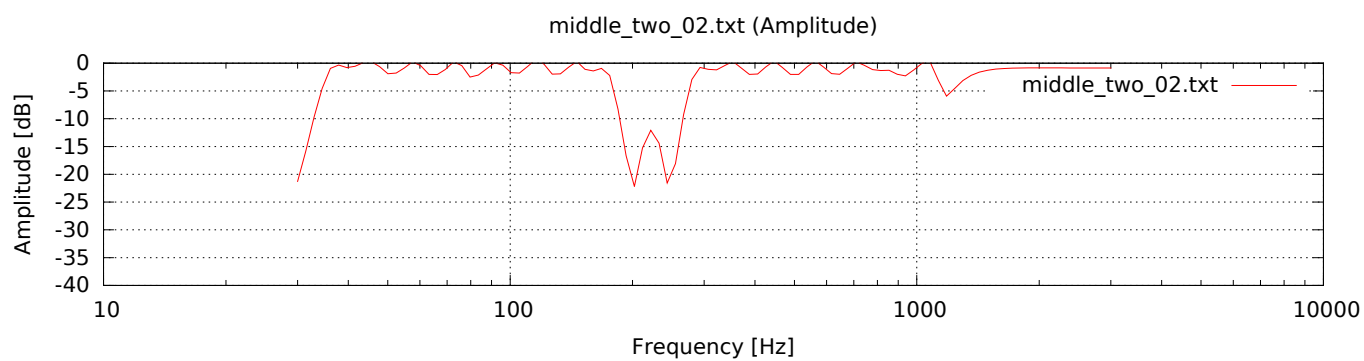
Figure A.28: Every other second filter gains are varied.



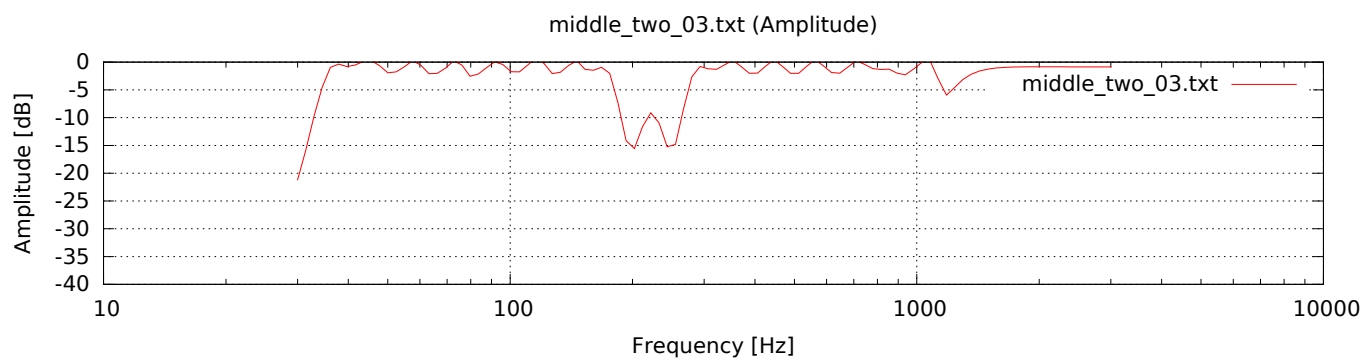
(a) Gain: 0



(b) Gain: 0,1



(c) Gain: 0,2



(d) Gain: 0,3

Figure A.29: Two middle filter gains are varied.

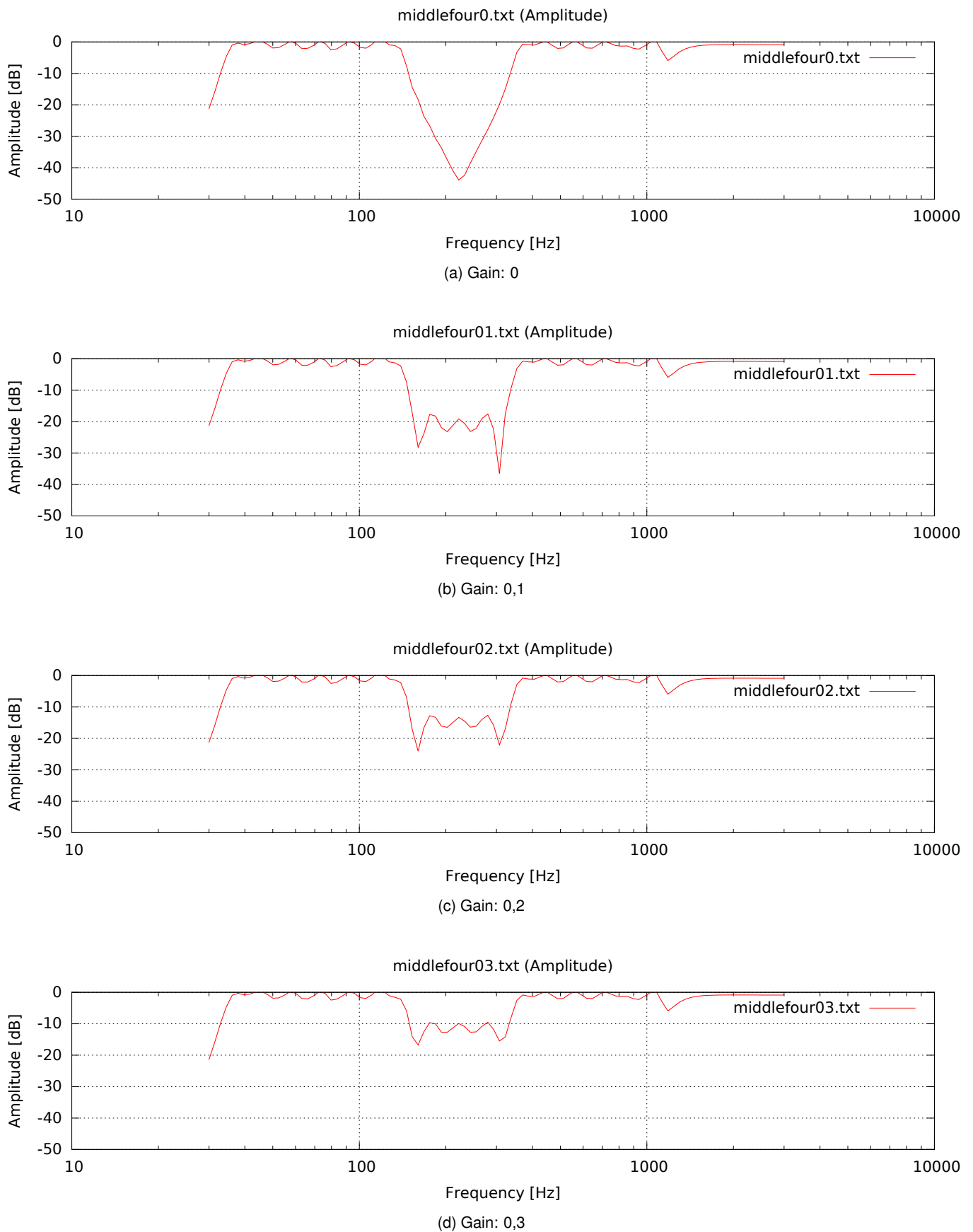


Figure A.30: Four middle filter gains are varied.

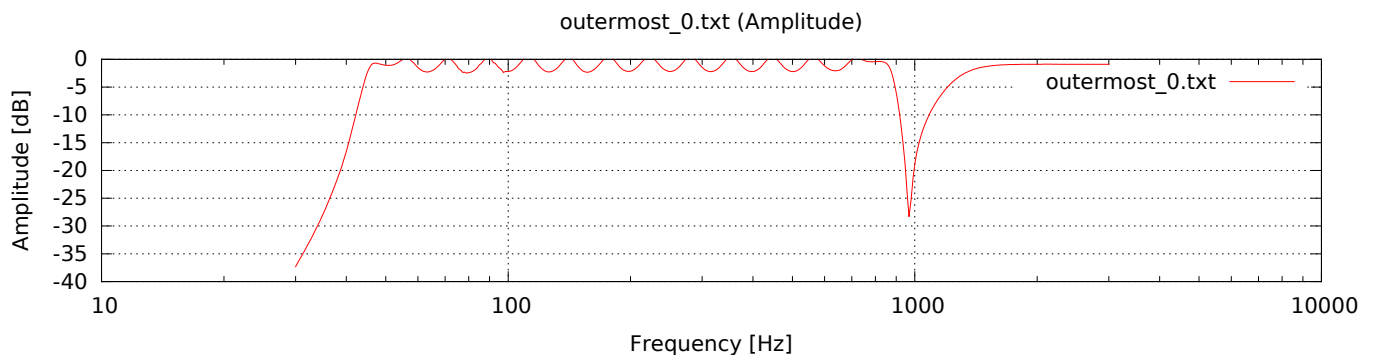
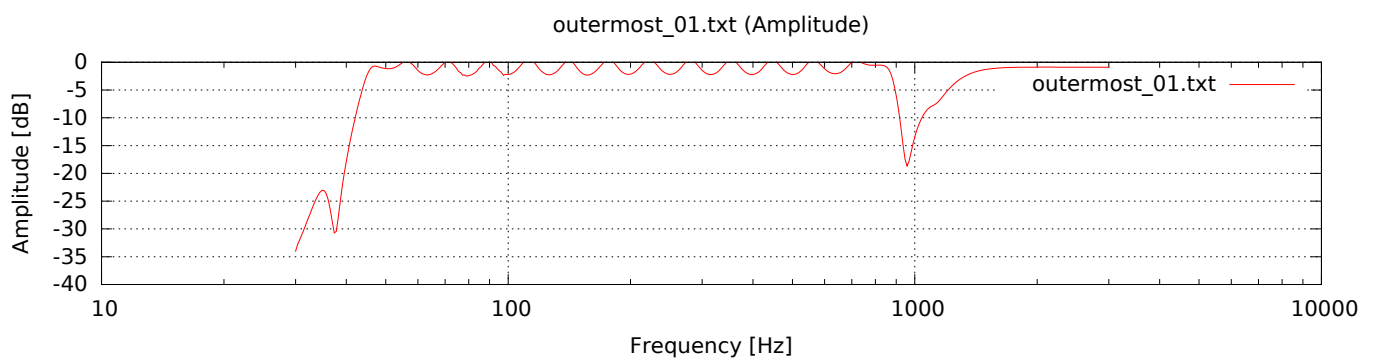
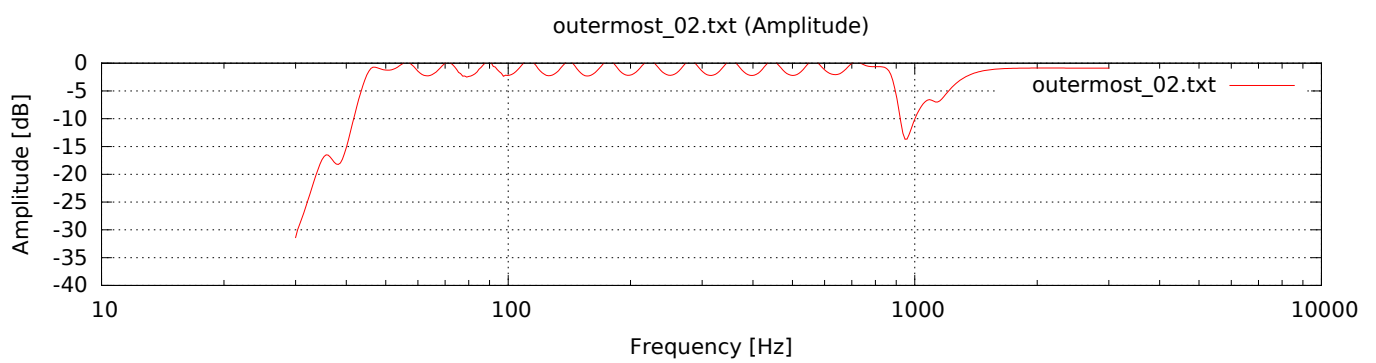
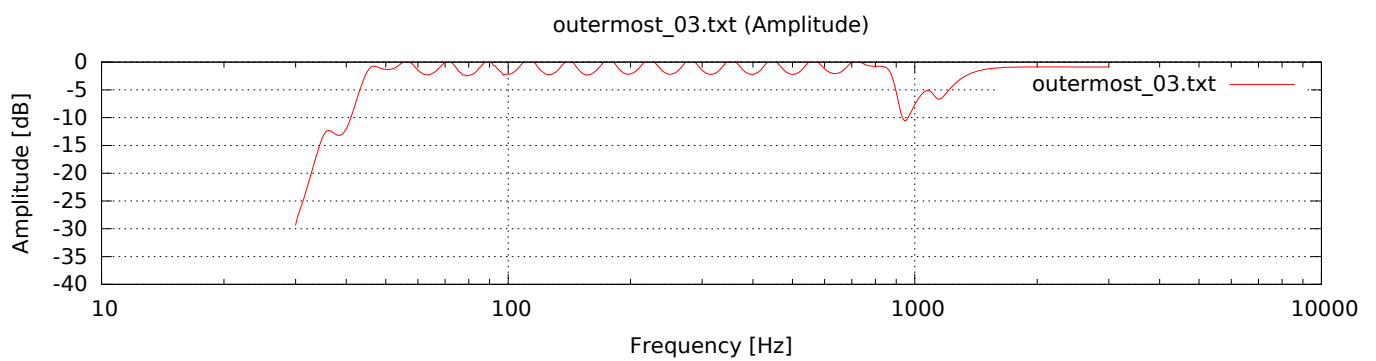


Figure A.31: The two outermost filter gains are varied.

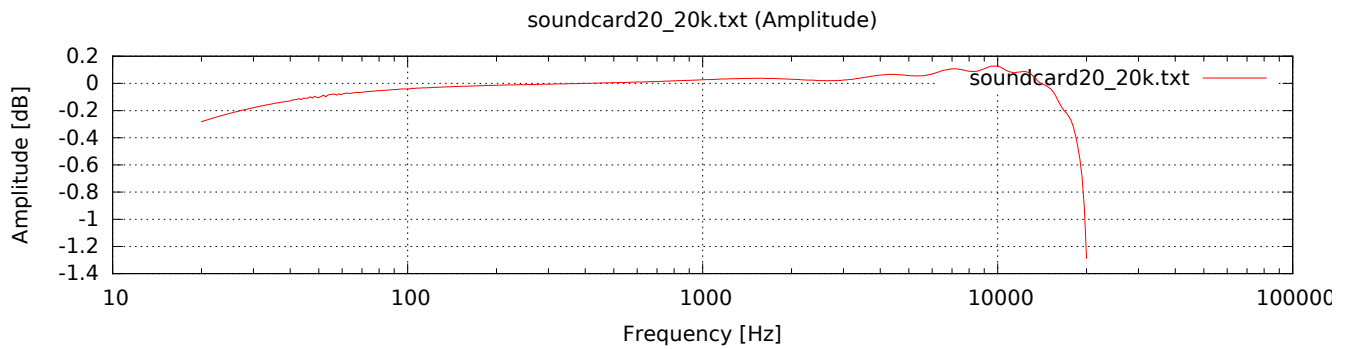


Figure A.32: Frequency Response of the UA-25EX sound card that was used during these tests.

A.4 Conclusion

The high pass has a bit of a change in cutoff frequency to minimize ripple around the transition from band pass filters to high pass filters. The new cutoff frequency is

- $f_{HP} = 1235 \text{ Hz}$

Also, the gain factor at 1235 Hz is set to not go above

- $\max G_{1235 \text{ Hz}} = 0,9$

and the maximum gain at 1000 Hz must not be above

- $\max G_{1000 \text{ Hz}} = 0,8$

When looking at the attenuation it is seen that the minimum gain, that should be used for the band pass filters, is

- $\min G = 0,2$

as this gives more attenuation than a gain of 0.

A.5 Measurements of Bluetooth Sound Output

A.5.1 Purpose

The purpose of the measurements, is to determine how well the Bluetooth module, is able to reproduce certain sinusoid-waves at different frequencies. The measurements will also determine the output voltage at different frequencies.

A.5.2 Test objects

- Bluetooth module RN-52

A.5.3 Test Setup

The following figures show the test setup

A.5.4 Equipment Used

Instrument	AAU-number.	Manufacture, type etc.
Oscilloscope	64572	Agilent Technologies DSO6034A
Bluetooth Module	N/A	Roving RN-52

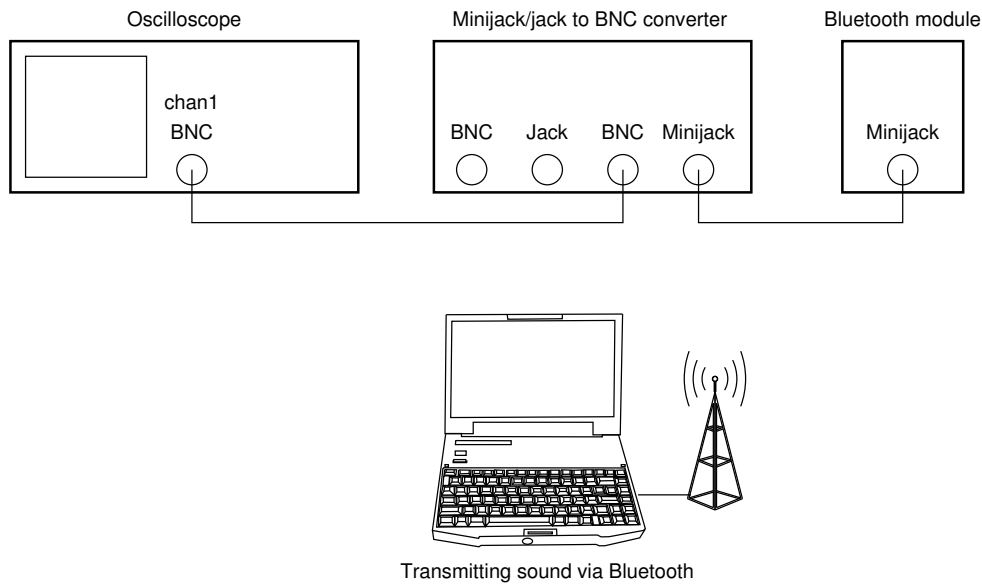


Figure A.33: Test setup with RN-52 module

A.5.5 Test Procedure

To test the frequency response, we first generated 24 different lossless wave files, which contains sinusoids at different frequencies. The wave files were made in Audacity and the different generated tones are shown in the following Table A.5.5.

Type	Frequency
Sinusoid	20 Hz
Sinusoid	50 Hz
Sinusoid	100 Hz
Sinusoid	500 Hz
Sinusoid	1 kHz
Sinusoid	2 kHz
Sinusoid	3 kHz
Sinusoid	4 kHz
Sinusoid	5 kHz
Sinusoid	6 kHz
Sinusoid	7 kHz
Sinusoid	8 kHz
Sinusoid	9 kHz
Sinusoid	10 kHz
Sinusoid	11 kHz
Sinusoid	12 kHz
Sinusoid	13 kHz
Sinusoid	14 kHz
Sinusoid	15 kHz
Sinusoid	16 kHz
Sinusoid	17 kHz
Sinusoid	18 kHz
Sinusoid	19 kHz
Sinusoid	20 kHz

The frequency and peak-to-peak voltage will be measured for each frequency in the Table A.5.7 below, on an oscilloscope. Furthermore, for each measurement the output from the oscilloscope will be saved to a csv-file for graphing of the waveform. The sound levels are set to 0 dB with no gain to avoid clipping. The Bluetooth module uses a 3,5 mm TRS connector for the sound output, and a 3,5 mm TRS connector to BNC was built to interface it with the oscilloscope, however all measurements was done in mono.

A.5.6 Results

A.5.7 Measurements

The raw results is in the Table A.5.7 below.

Input frequency	Measured frequency	V_{pp}
20 Hz	20,0 Hz	341 mV
50 Hz	50,0 Hz	344 mV
100 Hz	100,1 Hz	344 mV
500 Hz	500,0 Hz	344 mV
1 Hz	1,0 kHz	347 mV
2 kHz	2,0 kHz	341 mV
3 kHz	2,99 kHz	344 mV
4 kHz	4,0 kHz	344 mV
5 kHz	5,0 kHz	338 mV
6 kHz	6,01 kHz	344 mV
7 kHz	6,99 kHz	338 mV
8 kHz	8,0 kHz	334 mV
9 kHz	9,01 kHz	334 mV
10 kHz	10,0 kHz	331 mV
11 kHz	11,01 kHz	328 mV
12 kHz	11,99 kHz	325 mV
13 kHz	13,02 kHz	322 mV
14 kHz	14,01 kHz	319 mV
15 kHz	14,97 kHz	319 mV
16 kHz	15,97 kHz	316 mV
17 kHz	17,07 kHz	316 mV
18 kHz	18,12 kHz	313 mV
19 kHz	18,90 kHz	303 mV
20 kHz	19,69 kHz	275 mV

A.5.8 Output Waveforms

The measurement waveforms has been made for every frequencies represented in the Table A.5.7 above and can be found on the appendix CD.

 /Graphs and Figures/bt-module

A.5.9 Conclusion

From the raw measuring results in Table A.5.7 it is accurate both in the frequency and in the V_{pp} output at every frequency. But its seen that the waveforms are not perfect especially in the high frequencies. It does not seem to be stable in the frequency and its time-shifted. However it is not a major problem, and it should be sutable for the project.

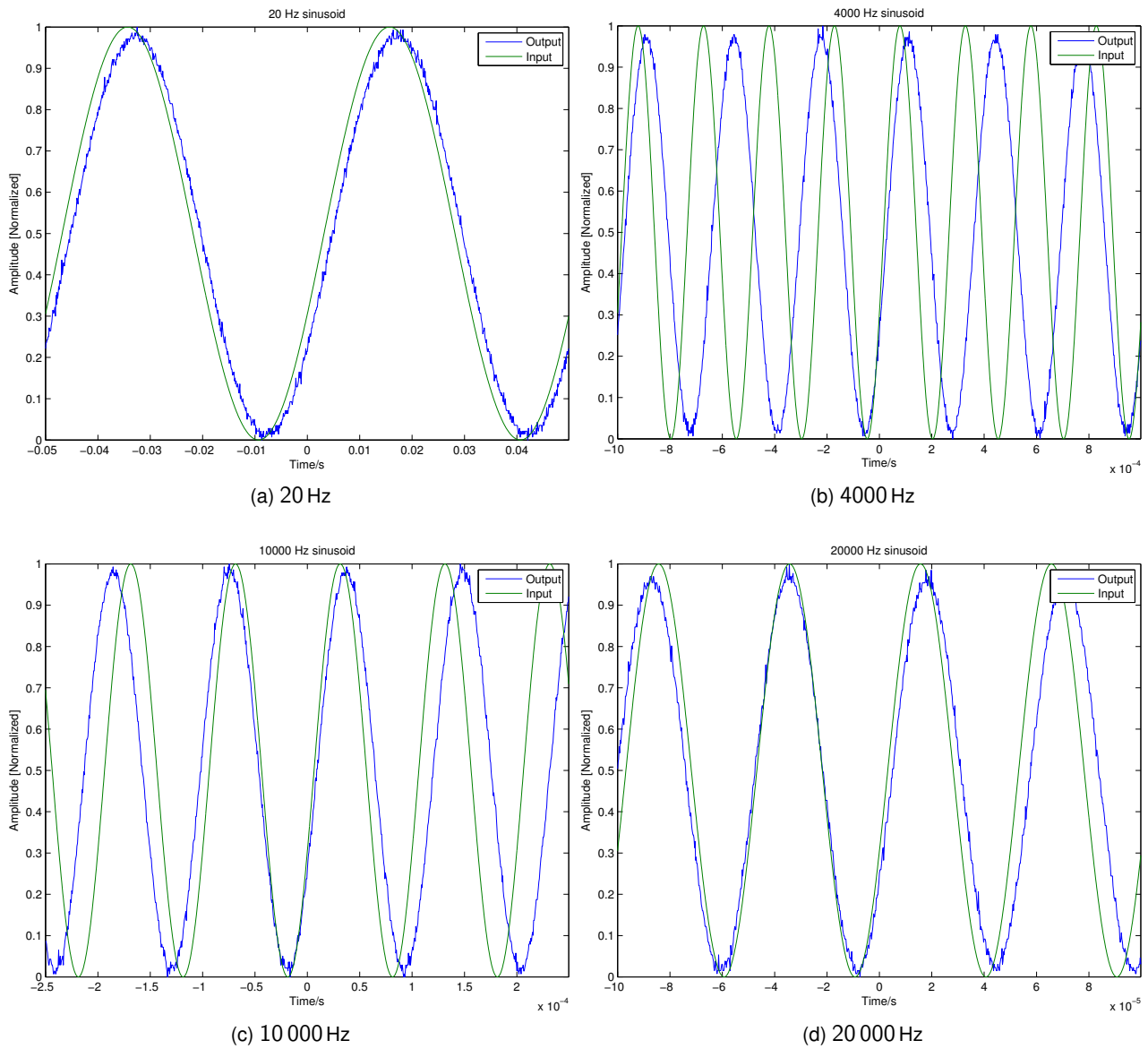


Figure A.34: The waveform at different frequencies.

A.5.10 Uncertainty of Measurements

It's hard to say which uncertainties there could be, when during the raw measurements on a oscilloscope. In this case the small calculation and reading uncertainties of the raw measuring results is not important to the conclusion, which were made by looking at the waveform comparisons. The raw measuring results were made to see if there were any big changes in the frequency and V_{pp} values and therefore the small uncertainties in the calculation and reading is not important to get the overall picture.

B Worksheets

B.1 Calculations for the Signal Amplifier for the Bluetooth Module

Component values

First the input voltage V_s is chosen to be 5 V, so it can be implemented with USB. Next a BC547 NPN transistor is chosen, the collector I_c is set to 1 mA. The node V_c is chosen to be 2 V, due to the input signal being around 350 mV_{pp} and since the amplification should be around 8 times, which makes $V_{out} = 2.8$. The following equation can be made,

$$V_c = 0,3 \text{ V} + 0,7 \text{ V} + I_c R_e = 2 \text{ V} \quad (\text{B.1})$$

Now the resistor R_c can be calculated, since we know there are 3 V over R_c because V_c is 2 V therefore R_c is given by the following equation.

$$R_c = \frac{3 \text{ V}}{1 \text{ mA}} = 3 \text{ k}\Omega \quad (\text{B.2})$$

Now the R_e resistor can be calculated, this resistor decides the amplification, and is given by[23]:

$$A_v = \frac{-g_m \cdot R_c}{g_m \cdot R_e + 1} \quad (\text{B.3})$$

Assumed that $R_e \gg 1$, the expression can be simplified to:

$$A_v = -\frac{R_c}{R_e} \quad (\text{B.4})$$

And now R_e can be calculated:

$$R_e = \frac{3 \text{ k}\Omega}{10} = 300 \Omega \quad (\text{B.5})$$

The bias-voltage V_b can be calculated:

$$V_b = I_c \cdot R_e + 0.6 = 0.4 + 0.6 = 1 \text{ V} \quad (\text{B.6})$$

The ratio between R_1 and R_2 is 5 : 1, because a high input impedance is wanted, therefore is $R_1 = 500 \text{ k}\Omega$ and $R_2 = 100 \text{ k}\Omega$

Next up is to calculate the capacitor values. The cutoff frequency is set to 2 Hz. First the value of C_2 is found—for this the output impedance is needed, so:

$$Z_o \approx R_c = 3 \text{ k}\Omega \quad (\text{B.7})$$

and thus:

$$2\pi f = \omega_0 = \frac{1}{RC_2} \quad (\text{B.8})$$

now we can calculate C_2 :

$$C_2 = \frac{1}{2\pi \cdot 2 \text{ Hz} \cdot 3 \text{ k}\Omega} = 20 \text{ }\mu\text{F} \quad (\text{B.9})$$

The same method is used to calculate C_1 and the input impedance is:

$$Z_i \approx R_1 \parallel R_2 = 83,3 \text{ k}\Omega \quad (\text{B.10})$$

thus C_1 is:

$$C_1 = \frac{1}{2\pi \cdot 2 \text{ Hz} \cdot 83,3 \text{ k}\Omega} = 955,31 \text{ nF} \quad (\text{B.11})$$

B.1.1 Impedances of the Voltage Amplifier

It is important to calculate and then measure the input and output impedance of the bipolar junction transistor amplifier circuit built in the Section 10.4. According to the Requirement 26 the output impedance must be less than 10 times the input impedance.

Calculation of The Impedances

To calculate impedances one must make a small signal equivalent circuit diagram for a BJT transistor. Then impedances can be calculated as $Z = \frac{V_Z}{I_Z}$

Input Impedance The diagram of the transistor circuit shown on the Figure B.1 is used to calculate input impedance. The resistor $R_B = R_1 \parallel R_2 = 83,333 \text{ k}\Omega$. From the data sheet it is known that $h_{FE} = 290$. Thermal voltage is known to be $25,7\text{E}-3$. The transconductance can be found from collector current and thermal voltage $gm = \frac{I_C}{V_T} = 38,91 \text{ mS}$. The basis-emitter resistance can be found as $r_\pi = \frac{h_{FE}}{gm} = 7,453 \text{ k}\Omega$. The current source can be written as $gm \cdot V_\pi$ or as $h_{FE} \cdot I_B$.

Input impedance can be written as:

$$Z_i = \frac{R_B \cdot (r_\pi + (1 + h_{FE}) \cdot R_E)}{R_B + (r_\pi + (1 + h_{FE}) \cdot R_E)} = 49,815 \text{ k}\Omega \quad (\text{B.12})$$

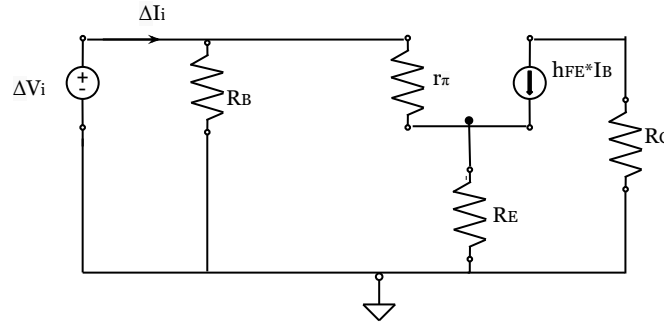


Figure B.1: Small signal equivalent diagram for the BJT transistor circuit when calculating input impedance

Output Impedance The diagram of the transistor circuit shown on the Figure B.2 is used to calculate output impedance.

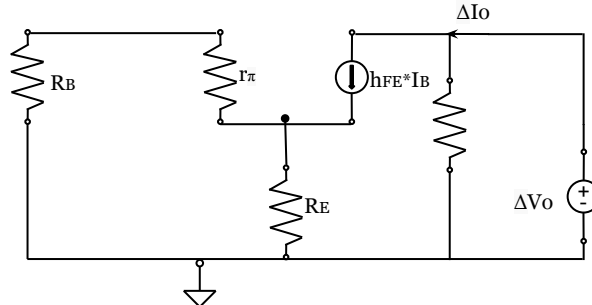


Figure B.2: Small signal equivalent diagram for the BJT transistor circuit when calculating output impedance

Output impedance can be written as:

$$Z_o = \left[\frac{R_B \cdot (r_\pi + (1 + h_{FE}) \cdot R_E)}{R_B + (r_\pi + (1 + h_{FE}) \cdot R_E)} \right] \parallel R_C = 3,703 \text{ k}\Omega \quad (\text{B.13})$$

Mesearments of the Impedances

The input impedance has been measured as described in Section 27. The output impedance has been measured as described in Section 26.

Input Impedance The voltage of the voltage generator is measured to $V_g = 0,328 V_{\text{peak}}$. The resistance has been chosen to be $10 \text{ k}\Omega$. The voltage measured after the voltage drop in resistance is $V_{Z_i} = 0,281 \text{ V}$.

Current can be found from the voltage drop across the resistance.

$$I = \frac{V_g - V_{Z_i}}{R_s} = 4,7 \mu\text{A} \quad (\text{B.14})$$

The input impedance can be found as:

$$Z_i = \frac{V_{Z_i}}{I} = 59,787 \text{ k}\Omega \quad (\text{B.15})$$

Output Impedance The voltage without load has been measured to $V_o = 2,47 \text{ V}$. Load has been chosen to be $34 \text{ k}\Omega$, as it is the input impedance of the TRS input on sound card. Voltage with load is measured to be $V_{R_L} = 2,22 \text{ V}$.

Current running through the load can be found from the voltage drop across the load.

$$I = \frac{V_{R_L}}{R_L} = 65,293 \mu\text{A} \quad (\text{B.16})$$

The output impedance can be found as:

$$Z_o = \frac{V_o - V_{R_L}}{I} = 3,8288 \text{ k}\Omega \quad (\text{B.17})$$

B.1.2 THD of the Voltage Amplifier

Calculation of the THD

THD can be found from the equation:

$$\text{THD} = \sqrt{\text{HD}_2^2 + \text{HD}_3^2} = 6,4547\text{E}-3 = 0,645 \% \quad (\text{B.18})$$

F can be found as:

$$F = 1 + \frac{I_C}{V_T} \cdot \left(\frac{R_s}{h_{FE}} + R_E \right) = 12.674 \quad (\text{B.19})$$

$$\hat{V}_s' = \hat{V}_s \cdot \frac{R_B}{R_s + R_B} = 99,99 \text{ mV} \quad (\text{B.20})$$

The second harmonic can be found as:

$$\text{HD}_2 = \frac{1}{4} \cdot \frac{\hat{V}'}{F^2 \cdot V_T} = 6,0499\text{E}-3 \quad (\text{B.21})$$

The third harmonic can be found as:

$$\text{HD}_3 = \left| \frac{3 - 2 \cdot F}{24 \cdot F^4} \right| \cdot \left(\frac{\hat{V}'}{V_T} \right)^2 = 2,25\text{E}-3 \quad (\text{B.22})$$

Measurment of the THD

THD has been measured with a frequency analyzer, the result can be seen on the Figure B.3

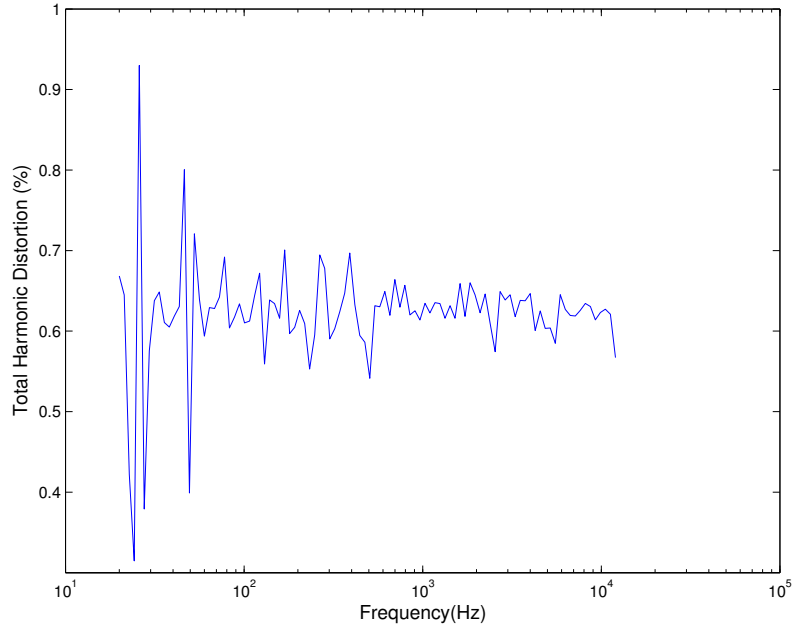


Figure B.3: Total Harmonic Distortion in the voltage amplifier

B.2 Frequency Analysis Procedures

The Base Station system is required to analyze some datasets containing the frequency content of both the reference signal and the microphone system input. This is suppose to be carried out in real time. In order for this design to be carried out it is necessary to outline the possibilities in this matter. There is the possibility of applying the fourier transform to the input signals. There is also the possibility of designing filter banks and downsampling. This will be discussed in the following.

B.2.1 Discrete Fourier Transform (DFT)

It is possible to apply the discrete fourier transform to the inputs, where the input samples are gathered in blocks through a window of a defined length in samples. This will add a delay to the response time of the filter adjustment. This delay will minimum consist of the the window time, added to the computation time. The Discrete Fourier Transform is described by Equation B.23.

$$X(m) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi mn/N} = \sum_{n=0}^{N-1} x(n)W_N^{mn} \quad (\text{B.23})$$

where

- m is a positive integer from 0 to $N-1$.
- $X(m)$ is the periodic frequency spectrum.
- $x[n]$ is the periodic input secuency.
- W_N is the twiddle factor $e^{-j2\pi/N}$.
- $N - 1$ is the filter order.

From this equation it is important to note that $X(m)$ will take N^2 complex multiplications and also $(N - 1)m$ additions.[19]

B.2.2 Fast Fourier Transform (FFT)

The required amount of calculations for the basic DFT is reduced by using the Fast Fourier Transformation (FFT) algorithm. The radix-2 FFT, by Tukey & Cooley, is done by separation of the input data sequence

$x(n)$ into its even and odd indexed elements. This will break Eq. B.23 into Eq. B.24.

$$X(m) = \sum_{n=0}^{(N/2)-1} x(2n)e^{-j2\pi(2n)m/N} + \sum_{n=0}^{(N/2)-1} x(2n+1)e^{-j2\pi(2n+1)m/N} \quad (\text{B.24})$$

Because $W_N^2 = e^{-j2\pi 2/(N)} = e^{-j2\pi/(N/2)}$ Equation B.24 can be expressed, as

$$X(m) = \sum_{n=0}^{(N/2)-1} x(2n)W_{N/2}^{nm} + W_N^{nm} \sum_{n=0}^{(N/2)-1} x(2n+1)W_{N/2}^{nm} \quad (\text{B.25})$$

Now the interesting part is the ability to give an N -point DFT, with only $N/2$ summations. This procedure of separating the input part into two parts, can then be done again as long as N allows for it. This means that N has to be chosen as a power of 2 to make this algorithm most efficient.[14]

B.2.3 Filter Banks

To be able to use great amount of filters, and reduce the amount of calculations in this matter, it is possible to use the technique of multirate signal processing, by downsampling. When downsampling a signal the data rate is lowered. This can be beneficial for algorithms that operates in real-time, because less data is processed. It is as always important to maintain the Nyquist sampling theorem, therefore it follows that the respective signal should be band limited when downsampling. When digital filters are designed in the normalized frequency domain of radians per sample, it is possible to move the cutoff frequency of a filter by changing the sample rate, as shown in Eq. B.26 [19].

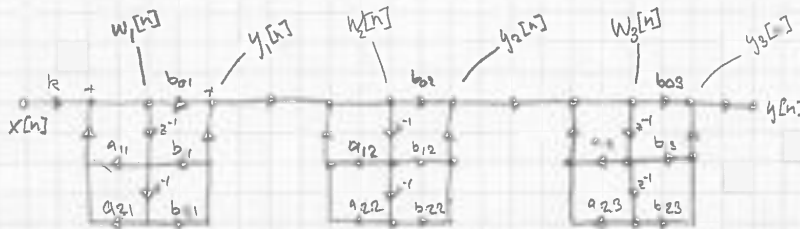
$$f_c = \frac{\Omega_c}{2\pi} f_s \quad (\text{B.26})$$

where

- f_c is the new cutoff frequency.
- Ω_c is the present normalized cut off frequency.
- f_s is the sampling frequency.

The cut off frequency will then be scaled to new frequency. In the case of using third octave design, described in Section 2.5 it is possible to use only three filters/channels. The octaves are separated by a factor of 2, so it would be possible to use one filter for the octave design, and decimate by a factor of 2.

B.3 Difference Equation for Cascaded Biquads



Filter example: 1000 Hz generated from Matlab Butter.
Oppenheim Schafer pp. 419

$$y_0[n] = x[n]$$

$$w_k[n] = a_{1k} w_k[n-1] + a_{2k} w_k[n-2] + y_{k-1}[n] \quad k = 1, 2, 3$$

$$y_k[n] = b_{0k} w_k[n] + b_{1k} w_k[n-1] + b_{2k} w_k[n-2] \quad k = 1, 2, 3$$

This gives a total of 6 difference equations.

$$w_1[n] = a_{11} w_1[n-1] + a_{21} w_1[n-2] + x[n]$$

code: $x = \text{new sample } k$
 $w1 = a11 \cdot w1_{-1} + a21 \cdot w1_{-2} + x$

$$y_1[n] = b_{01} w_1[n] + b_{11} w_1[n-1] + b_{21} w_1[n-2]$$

$$y1 = b0 \cdot w1 + b11 \cdot w1_{-1} + b21 \cdot w1_{-2}$$

$$w_2[n] = a_{12} w_2[n-1] + a_{22} w_2[n-2] + y_1[n]$$

$$w2 = a12 \cdot w2_{-1} + a22 \cdot w2_{-2} + y1$$

$$y_2[n] = b_{02} w_2[n] + b_{12} w_2[n-1] + b_{22} w_2[n-2]$$

$$y2 = b02 \cdot w2 + b12 \cdot w2_{-1} + b22 \cdot w2_{-2}$$

$$w_3[n] = a_{13} w_3[n-1] + a_{23} w_3[n-2] + y_2[n]$$

$$w3 = a13 \cdot w3_{-1} + a23 \cdot w3_{-2} + y2$$

$$y_3[n] = b_{03} w_3[n] + b_{13} w_3[n-1] + b_{23} w_3[n-2]$$

$$y' = b03 \cdot w3 + b13 \cdot w3_{-1} + b23 \cdot w3_{-2}$$

$$y[n] = y_3[n]$$

output sample is 'y'.

$$w3_{-2} = w3_{-1}$$

$$w3_{-1} = w3$$

$$w2_{-2} = w2_{-1}$$

$$w2_{-1} = w2$$

$$w1_{-2} = w1_{-1}$$

$$w1_{-1} = w1$$

Coefficients:

$$1 \quad a_{11} \quad a_{21} \quad b_{01} \quad b_{11} \quad b_{21}$$

$$2 \quad a_{12} \quad a_{22} \quad b_{02} \quad b_{12} \quad b_{22}$$

$$3 \quad a_{13} \quad a_{23} \quad b_{03} \quad b_{13} \quad b_{23}$$

And: k

MATLAB OUTPUT =

$$h1num \leftarrow b_{01} \quad b_{11} \quad b_{21}$$

$$h1den \leftarrow 1 - a_{11} - a_{21}$$

$$h2num \leftarrow b_{02} \quad b_{12} \quad b_{22}$$

$$h2den \leftarrow 1 - a_{12} - a_{22}$$

$$h3num \leftarrow b_{03} \quad b_{13} \quad b_{23}$$

$$h3den \leftarrow 1 - a_{13} - a_{23}$$

$$k \leftarrow k$$

Remember to change
Sign on a-parts.

C Extended Analysis

C.1 Speaker

When designing a loudspeaker system, there are many aspects to take into account. The speaker design is considered as a very important factor when building a loudspeaker system. In this project the goal is to design a product for any home environment and therefore this system must be designed to work with different speaker designs. The more relevant aspects in this project are the speaker pressure level and how sound waves travel through air.

C.1.1 Speaker Sensitivity

Sensitivity of a loudspeaker is often specified in terms of Sound Pressure Level for 2,83 V_{RMS} input [15]. Sensitivity is defined as a value of dB_{SPL} at 1 W electrical input. The measurement is usually carried out with a voltage of constant 2,83 V_{RMS}, at 8 Ω impedance, measured at the distance of 1 meter.

The power, P_L , which is transferred to the speaker load, R_L , is calculated from the voltage V across the load:

$$P_L = \frac{V^2}{R_L} \Rightarrow \frac{(2,83 \text{ V}_{\text{RMS}})^2}{8 \Omega} = 1 \text{ W} \quad (\text{C.1})$$

So if measuring 80 dB_{SPL} at 1 V_{RMS}, the loudspeaker sensitivity is calculated, for a 8 Ω speaker, 2,83 V as showed in Equation C.2, by calculating the difference to when 2,83 V is applied, and then adding this difference to the measured SPL.

$$\Delta \text{SPL} = 20 \log \left(\frac{2,83}{1} \right) = 9,4 \text{ dB} \Rightarrow \text{Sensitivity} = \text{SPL} + \Delta \text{SPL} = 80 \text{ dB} + 9,4 \text{ dB} = 89,4 \text{ dB} \quad (\text{C.2})$$

A measurements taken with this reference are quoted as 89,4 dB with 2,83 V@1 m. These ratings based on the SPL for a given input are called sensitivity ratings and are notionally quite similar to efficiency, but these are not the same. Speaker sensitivity is expressing how a given voltage translates to pressure variations in the air.

C.1.2 Sound Proagation Through Air

The definition of sound pressure level, L_p , is expressed in Equation C.3, where P_{RMS} is the pressure variation. It is relative to the minimum pressure level that the human ear can hear, which is 20 μPa.

$$L_p = 20 \log \left(\frac{P_{\text{RMS}}}{P_{\text{ref}}} \right) \quad (\text{C.3})$$

where

- L_p is the sound pressure level in dB
- P_{RMS} is the sound pressure in Pa
- $P_{\text{ref}} = 20 \text{ μPa}$ is the reference sound pressure in Pa.

The velocity of any sound wave through air is given by the following Expression C.4

$$v = 331,5 \text{ m/s} \cdot \sqrt{1 + \frac{t}{273 \text{ K}}} \quad (\text{C.4})$$

where

- v is the velocity in m/s.

- t is the measured temperature in $^{\circ}\text{C}$.
- K is the degree in Kelvin.

[5]

There is a linear relationship between the velocity, the wavelength and the frequency as shown in Equation C.5.

$$\text{velocity} = \text{Wavelength} \cdot \text{Frequency} \quad (\text{C.5})$$

From the expression in Equation C.5 the wavelength can be found in Equation C.6:

$$\text{Wavelength} = \frac{\text{velocity}}{\text{Frequency}} \quad (\text{C.6})$$

Thus the lower the frequency the greater the wavelength. This means that it has to be taken into account that low frequency sound waves contains great amount of energy, and also takes up more space in a digital memory. Another important factor when talking about sound waves through air is that sound travels in spherical direction and reflects when meeting hard surface mediums such as walls. As described earlier the idea is to design a product that will work in any home environment and therefore this must be considered in the design of this system.

C.2 Capacitor Microphone

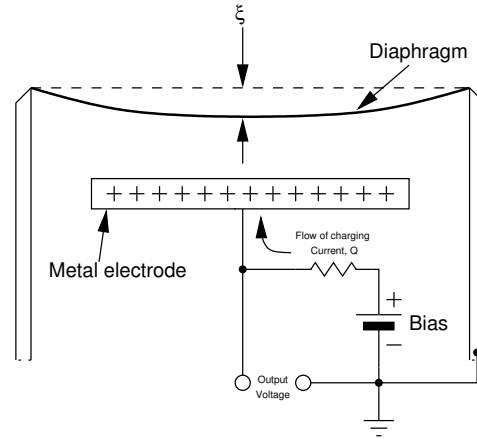


Figure C.1: Diagrammatic representation of the principle of a capacitor microphone

C.2.1 Transducer Principles

As shown in Figure C.1 the diaphragm is a membrane that acts as one plate of a capacitor. A second electrode serves as the other plate. When sound pressure variations moves this diaphragm inwards, the capacitance of this capacitor increases. The charge, Q , of the capacitor is kept constant. Then, the potential across the capacitor drops, because $V = Q/C$. When the cycle reverses and a negative pressure is applied to the diaphragm, the potential across the capacitor increases. The output voltage, therefore, is proportional to the displacement, ξ , of the diaphragm, as shown in Equation C.7.

$$v = \frac{Q}{\epsilon A} \xi \quad (\text{C.7})$$

where

- Q is the supplied constant charge in A.
- A is the area of the overlap of the two electrodes in m^2
- ξ is the displacement of the diaphragm in m

- ϵ is the dielectricum of the capacitor which is a medium-dependant constant.
- v is the output voltage signal in V.

[5] The disadvantage of the conventional capacitor microphone is the necessity of a 12 to 48 V polarizing voltage [4]. A variant has been developed, without this necessity, which is build with pre-polarized electret membrane. These electret microphones have the advantages of freedom from a power supply and also lower self-noise in high humidity [3].

C.3 Microphone Sensitivity and Sources of interference

The microphone sensitivity is the factor of signal voltage output, which is produced when a given input of sound pressure is applied to the microphone. The common notation for sensitivity is M in V/Pa. The microphone sensitivity is usually close to 1 Pa, which is equal to 94 dB_{SPL}.

$$L_p(1 \text{ Pa}) = 20 \log \left(\frac{1 \text{ Pa}}{20 \mu\text{Pa}} \right) = 94 \text{ dB} \quad (\text{C.8})$$

Some typical values of microphone sensitivities are shown in Table C.1. Since the output of a microphone is

Type	V/Pa	dB re 1 V/Pa
Dynamic	1 – 5 mV	–60 – –46 dB
Capacitor	5 – 50 mV	–60 – –26 dB
Electret	2,5 – 10 mV	–52 – –40 dB

Table C.1: Typical values of sensitivity for different types of microphones [4].

linear with pressure, the units of V/Pa fall out naturally. Although sensitivity units of V/Pa are commonly used, the older units are still in use, and microphone sensitivity can be expressed in dB using a calculation that simply references the output of the microphone to a theoretical microphone which can pump out 1 V_{RMS} for 1 Pa of RMS pressure, as shown in Equation C.9, where M is the sensitivity, usually in mV.

$$\text{sensitivity} = 20 \log \left[\frac{M}{1} \right] \quad (\text{C.9})$$

C.3.1 Sources of Interference

It is desirable to have an electrical output from the microphone which is a faithful, undistorted image of the sound pressure acting on the microphone. Practical microphones may fall short of ideal performance. To accomplish an overview in the design phase, the error sources are listed here.

- Noise or induced signal may be present when there is no acoustical signal
- Nonlinear distortion can generate harmonics from a pure tone and causes the appearance of sum and difference frequencies when two or more pure tones are involved.
- Frequency distortion may suppress or accentuate some frequencies.
- Phase distortion, which delays the response of a microphone to different frequencies by different amounts, is likely to be present
- The sensitivity of the microphone may change as a result of temperature, humidity, or ambient pressure changes, rough usage, or age.

If the nature and magnitude of the imperfections of the device are known, these can be cancelled out. Some techniques are listed here.

- Changes in sensitivity can be cancelled by compensatory changes in attenuation or amplification of the electric signal.

- Phase and frequency distortion can be cancelled, over a limited frequency range, by electric equalizing networks.
- Induced signals may be excluded by shielding or shortening leads.

This sums up some important factors to choose a microphone for precise measurement.

- Sensitivity tolerance.
- Frequency distortion tolerance.
- Phase distortion tolerance.
- Non-linear distortion tolerance.
- Self-noise tolerance.

[3]

C.4 Bluetooth Communication Scheme

Bluetooth was originally developed for the use with mobile phones, PDA's, etc. It is an Ad Hoc communication scheme that is in the 2.4 Ghz frequency spectrum. Today it is used in many applications and there are many different protocols on top of the Bluetooth stack.

C.4.1 Radio Spectrum

The operating frequency is chosen mainly such that the spectrum is public with any licenses worldwide. This requirement narrows it down to the unified worldwide unlicensed band that is 2.4 Ghz. However since this band is public available a lot of devices use it such as WiFi, Amateur Radio and even microwave ovens. So this channel is bound to have a lot of noise from other devices, which should be taking into account.

Interference

Instead of doing interference suppression in the Bluetooth hardware, it was found that avoidance was more practical since the 2.4 Ghz band provides around 80 MHz of bandwidth. So when a channel is jammed due to interference, it changes to avoid conflicts with interference. It performs around 1600 frequency hops per second, with AFH enabled (Adaptive Frequency-Hopping)

C.4.2 Modulation Scheme

In the original design Bluetooth only used a Gaussian frequency-shift keying modulation (GFSK), however since Bluetooth became widely spread and applications needed more throughput, new modulation schemes was introduced in the Bluetooth 2.0. The new modulation schemes are $\frac{\pi}{4}$ -DQPSK and 8DPSK modulation, which significantly improves the throughput, each having a throughput of respectively 2 and 3 Mbit/s .

C.4.3 Protocol and packets

Bluetooth implements a packet based communication, the stream of data is fragmented and wrapped in packets. All packets have the same format, starting with an access code, then a packet header and lastly the payload. This packet-format can be seen on Figure C.2.

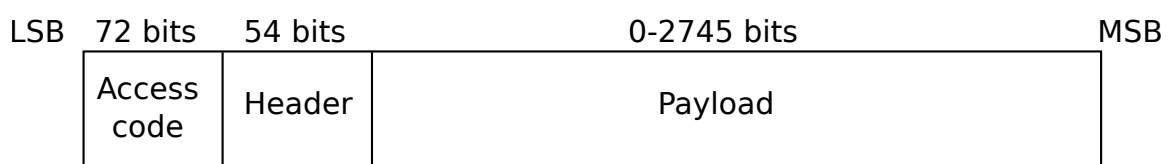


Figure C.2: Format of Bluetooth packets.

The Access code includes an identifier for the master, and every time a package is send it is identified by this master identity and only in the case it matches, the master accept the package. This avoids spoofed messages and also crossover if two devices happen to be on the same hop carrier frequency.

The Header includes a 3 bit slave address to separate different slaves, a 4 bit packet type that defines what type of payload it is transmitting, a 1 bit ACK/NACK, a 1 bit flow control bit, and an 8 bit CRC code for error detection.

The Payload includes nothing but the user data, and it has a variable length from 0 to 2745 bits.

Protocol stack

The Bluetooth protocol is build like many other network technologies, where different abstraction layers are used. The Bluetooth protocol stack is shown on Figure C.3.

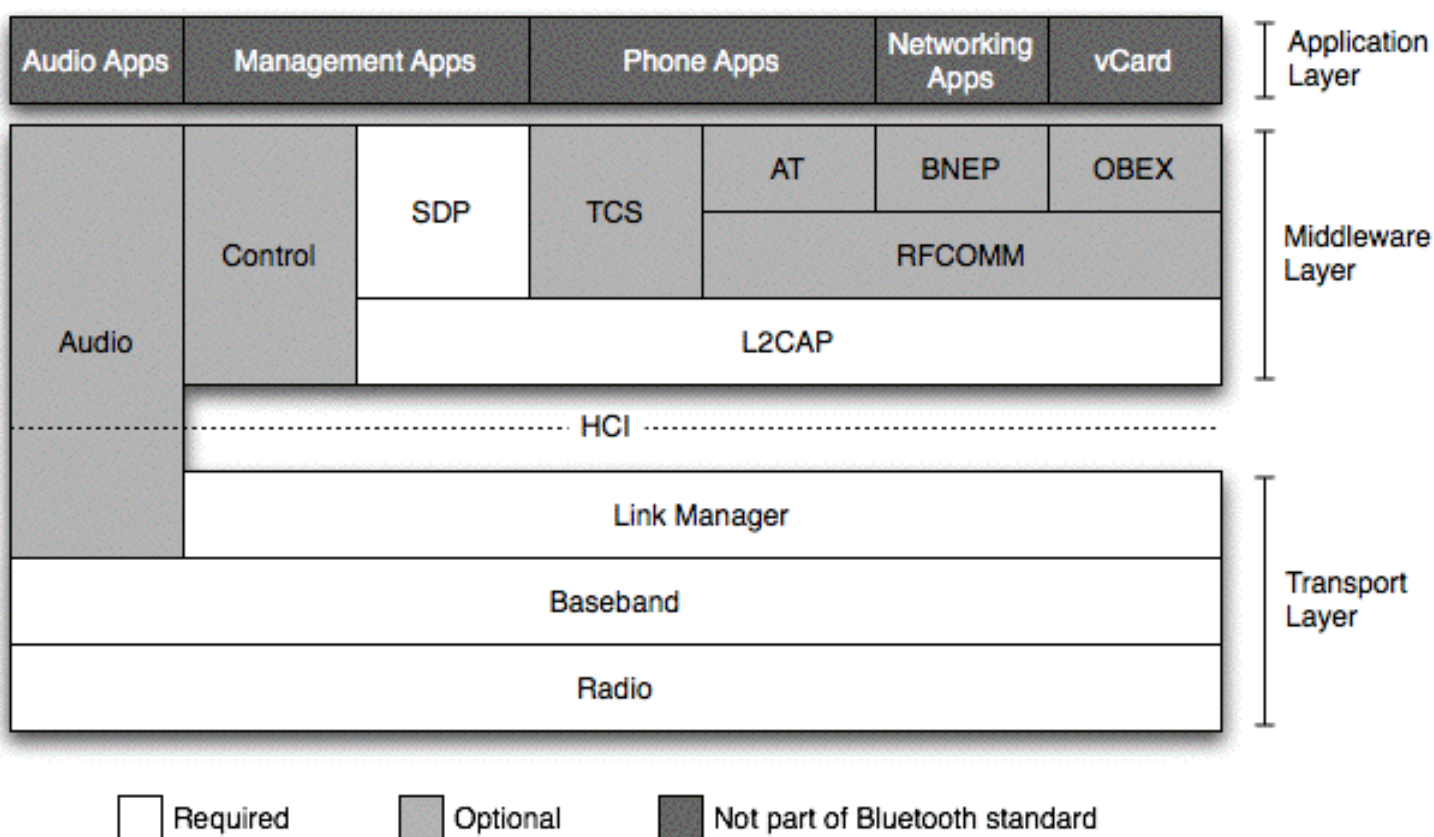


Figure C.3: Bluetooth protocol stack.

C.4.4 Power usage

Because there are so many different wireless communication methods and devices, this section will focus on the power consumption of various Bluetooth versions.

A new kind of Bluetooth is being develop, called Bluetooth Low Energy, which is a new kind of energy efficient Bluetooth. There have been made several tests, comparing regular Bluetooth to the new power efficient Bluetooth Low energy. By looking closer at these tests it is clear these results can be concluded.

- Peak current consumption for the regular Bluetooth: < 30 mA.[6]
- Peak current consumption for the Bluetooth Low Energy: < 20 mA.[6]

Since most Bluetooth devices uses the regular Bluetooth technology, it can be assumed that the power consumption will be around 30 mA, according to the tests performed.

C.5 Filter Definition

Filters are used widely in electronics for adjusting the frequency bands by removing the unwanted frequencies. Filters can be divided in low pass, high pass, band pass, band reject, and all pass types. It's possible to combine these into more complex filter systems. Filters can be designed as passive or active. Active filters use active components, such as amplifiers, and passive filters only uses passive components like capacitors, inductors, and resistors.

Filters can be designed in two ways: analog or digitally. Analog filters are only suitable for continuous-time signals where digital filters are suitable for discrete-time signals. The transfer function of an analog filter can be obtained with the Laplace transform, which can be translated to discrete-time with the z-transform.

When designing a filter, the chosen frequency response can be achieved by designing a transfer function, which depends on component values and circuit layouts.

Digital Filters Filters can be designed with or without feedback circuits. Filters with feedback are called infinite impulse response (IIR) filters, and filters without feedback are called finite impulse response (FIR) filters. FIR filters are more used than IIR filters. FIR filters are easy to implement on digital signal processors and can be done just by looping simple instructions. With FIR filters, it is possible to achieve performance levels which are not with analog filters. For example linear phase response. On the other hand, an IIR filter is based on classical analog filter, and its impulse response extends over an infinite period of time. Another thing that characterizes IIR filters is non-linear phase response and higher efficiency than FIR filters. A most common method for designing a IIR filter is to first design an analog equivalent filter and then transform the function $H(s)$ into the z-domain $H(z)$. All common analog filter types like Butterworth, Chebyshev, elliptical, and so on, can be implemented with IIR digital filter techniques. [13]

Series and Parallel Filters Filters can be built up into bigger filter systems which can consist of more than one filter. Generally there are two filter topologies.

- When two or more filters are linked in series their transfer functions multiply together.
- When two or more filters are linked in parallel their transfer functions sum together.

Filters in Series On the Figure C.4 we can see two transfer functions $H_1(z)$ and $H_2(z)$, which are linked in series. The combined transfer function $H(z)$ can be expressed as

$$\frac{H_2(z)V(z)}{X(z)} = H_1(z)H_2(z) \quad (C.10)$$

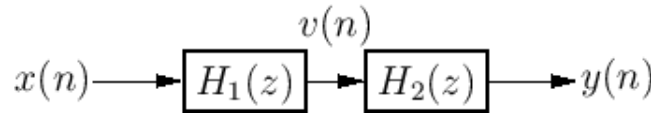


Figure C.4: transfer functions linked in series[10].

Filters in Parallel On the Figure C.5 can be seen two transfer functions $H_1(z)$ and $H_2(z)$, which are linked in parallel. The combined transfer function $H(z)$ can be expressed as

$$\frac{Y_1(z) + Y_2(z)}{X(z)} = \frac{Y_1(z)}{X(z)} + \frac{Y_2(z)}{X(z)} = H_1(z) + H_2(z) \quad (C.11)$$

C.5.1 Filter Comparisons from analysis

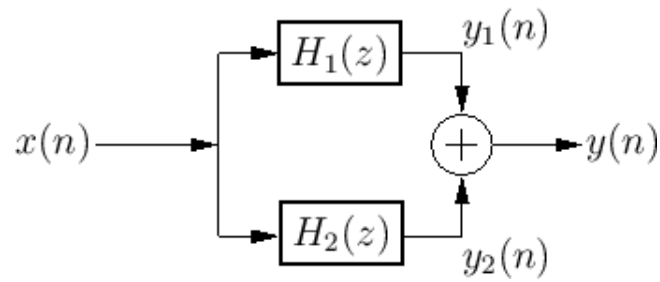


Figure C.5: Two transfer functions linked in parallel[10].

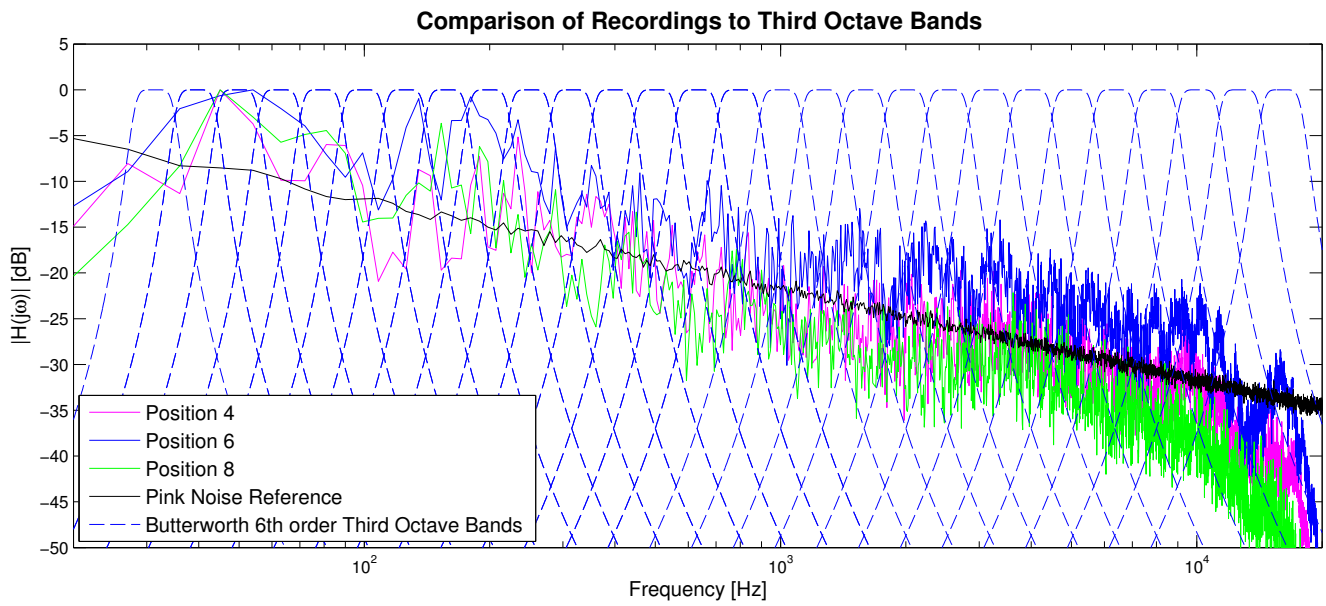


Figure C.6: Comparison of recordings and third octave design

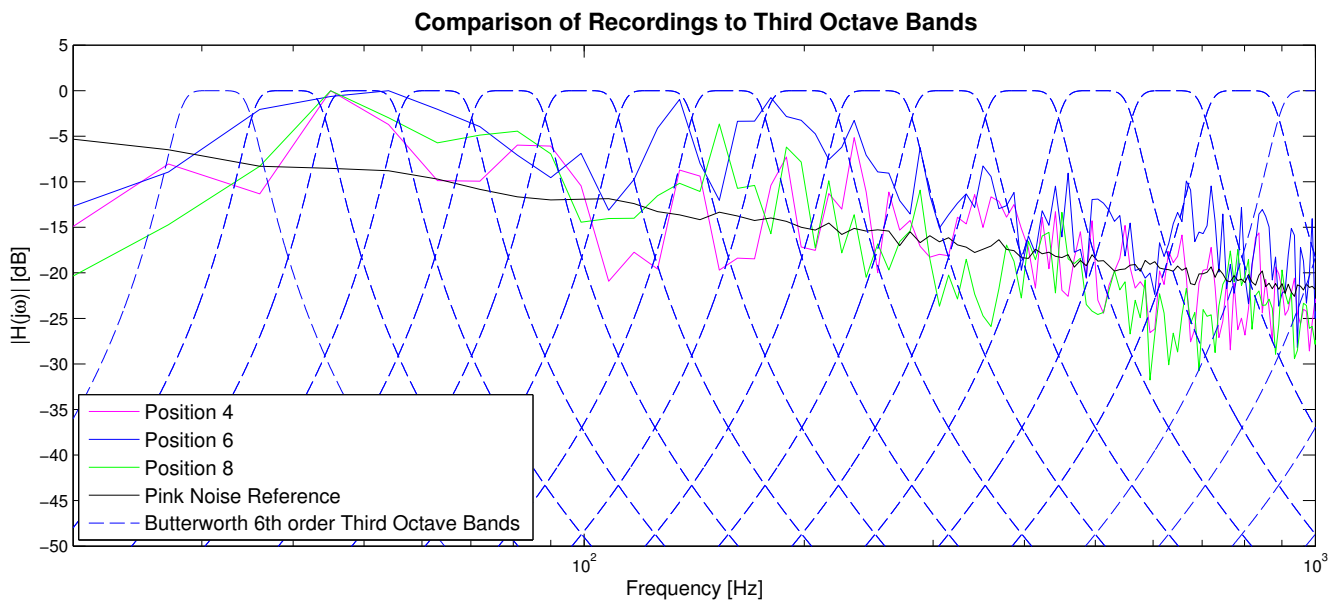


Figure C.7: Comparison of recordings and third octave design in the lower end of the audible range .

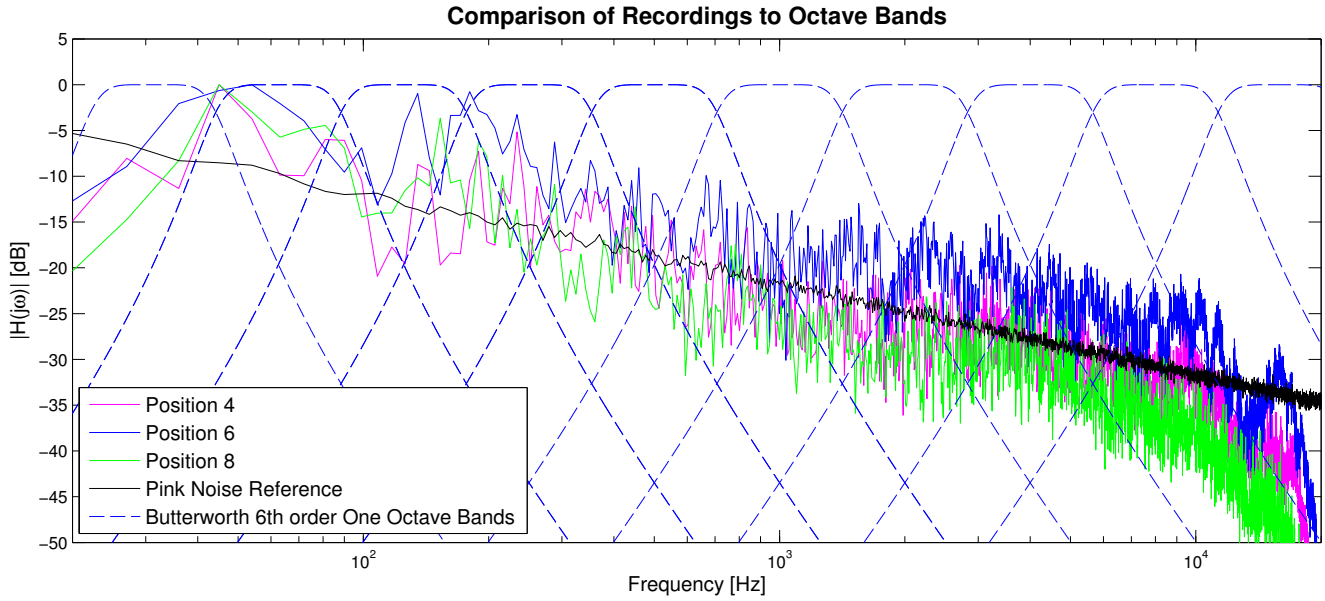


Figure C.8: Comparison of recordings and one octave design

C.6 Filtering in Audio

Acoustics filters are widely used to damp different frequencies in the frequency spectrum. As it is known, most people can not hear sounds wider than spectrum of 20 Hz to 20 kHz. This is why most acoustic filters are built to adjust frequencies in that spectrum.

Equalizing in Audio Equipment To have an overview of the existing equalizing technologies it has been chosen to take a look at the professional hardware equalizer DN360, made by Klark Teknik Group and the cheaper car equaliser PIONEER DEH-X3500UI. The DN360 equalizer has 30 frequency bands, which range from 25 Hz to 20 kHz with maximum boost/cut-off of 12 dB.

On the Figure C.9 can be seen all 30 frequency responses of the DN360 equalizer filters. It can be seen that DN360 uses 30 band pass filters with different band width. Band width of the filters increases to the centre frequency, because of the logarithmic nature of the design.[12] On the other side car equaliser PIONEER DEH-X3500UI, uses design with only five bands with centre frequencies in 80 Hz/250 Hz/800 Hz/2,5 kHz/8 kHz, the equalisation range is ± 12 dB with 2 dB step.[20]

Transition band ratio Transition band ratio is the ratio between stop band frequency and pass band frequency (ω_s/ω_p). The lower the ratio—the steeper is the curve of the filter. In case of analog filters the transition band ratio can also depend on component reaction times. In this particular project it is important to choose the frequency band ratio in such way, that all important frequency spectrum are covered.

Attenuation Attenuations are the values of the $H(j\omega)$ equivalent to the values of the ω_p and ω_s . In some cases values of attenuations α_p and α_s determinate the amplitude of the ripple and the type of the filter to be used.

Boost/Cut-off The maximum boost or cutoff in the equaliser is in dB the amount frequency that can be damped or boosted. This value will be affected by the transition band ratio of the filter.

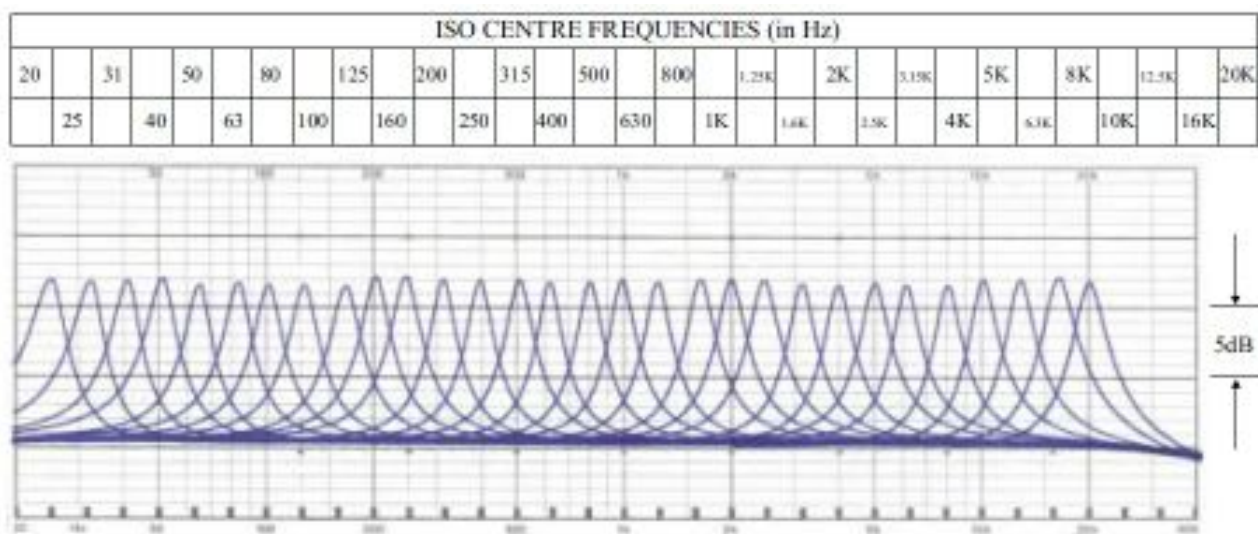


Figure C.9: Frequency bands and width of each band in DN360[12]

C.7 Description of the System Functionality by Use Case Description

This section is used to initialize the outlining in defining the system functionality before the system is drawn in a block diagram. It will describe the main functions of the system and give an indication of the system interfaces. In the usecase diagramme Figure C.10 the functionalities are clustered into two general

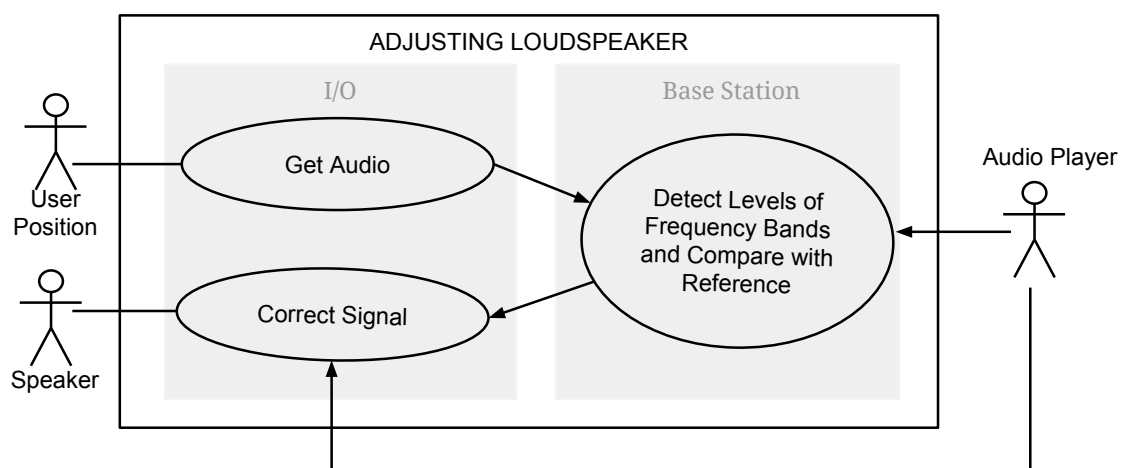


Figure C.10: Use-case diagram of the system

groups with the titles I/O and Base Station. All functionalities are described in the following use-cases, to give an overview of the system. The diagramme Figure C.10 will identify the outer interfaces to the system, as well as the main interfaces inside the system. Furthermore the following use case descriptions will imply how to divide the system down into smaller blocks, and where to define the interfaces of these.

Use Case	Get Audio
Description	The audio is detected via a microphone from where the 'User' is. The audio signal is converted into an electric signal and passed on to Detect Audio'.
Actors	'User' and 'Detect Audio'.
Steps	<ol style="list-style-type: none"> 1. REPEAT <ol style="list-style-type: none"> 1.1 Get audio signal 1.2 Convert audio into electric signal 1.3 Output electric signal to Detect audio' UNTIL turned off

Figure C.11: Use case description of 'Get Audio'

Use Case	Detect Level of Frequency Bands and Compare with Reference
Description	Measure the average level of the frequency bands in the input signals from 'Audio Player' and 'Get Audio' and calculate the information of difference between these.
Actors	'Get Audio' & 'Output and Correct Signal'
Steps	<ol style="list-style-type: none"> 1. REPEAT <ol style="list-style-type: none"> 1.1 Read signals from 'Get Audio' & 'Audio Player' 1.2 Align Signals in time. 1.3 Convert into frequency domain. 1.4 Divide both of these signals into several frequency bands. 1.5 Compare the mean level of the two signals within the consistent frequency bands. 1.6 Calculate the correction information signal. 1.7 Write the correction signal to 'Output and Correct Signal' UNTIL turned off

Figure C.12: Use case of description of 'Detect Level of Frequency Bands and Compare with Reference'.

Use Case	'Output' and 'Correct Signal'
Description	Equalize audio signal from 'Audio Player' and playback audio to the 'Speaker'. Use information from 'Detect Level'.
Actors	'Detect Level', 'Speaker' and 'Audio Player'
Steps	<ol style="list-style-type: none"> 1. REPEAT <ol style="list-style-type: none"> 1.1 Read the correction signal from 'Compare' 1.2 Set filters by using the information from the correction signal. 1.3 IN PARALLEL <ol style="list-style-type: none"> 1.3.1 Output the audio to 'Speaker' UNTIL turned off

Figure C.13: Use case description of 'Output' and 'Correct Signal'

D Source Code

```
1 #include "butt6bp.h"
2 #include "mypcm.h"
3
4
5 //-----Functions-----
6
7 /**
8  * Function to calculate RMS value.
9  * @param buf pointer to data_buffer.
10  * @return float containing RMS value.
11  */
12 double calculate_RMS (signed short *buf,int size)
13 {
14
15     double squared_sum;
16     double data_RMS;
17     int i;
18
19     for ( i = 0 ; i < size ; i++ )
20         squared_sum += buf[i] * buf[i];
21     data_RMS = squared_sum / size;
22     data_RMS = sqrt(data_RMS);
23     return data_RMS;
24 }
25
26
27 /**
28  * Calculates the gain factor from the difference of RMS value of two buffers
29  * @param *ref_buf pointer to the reference buffer
30  * @param *mic_buf pointer to the microphone buffer
31  * @return gain_factor calculated gain factor
32  */
33 double calculate_coeff(signed short *ref_buf,signed short *mic_buf)
34 {
35     double ref_rms;
36     double mic_rms;
37     double gain_factor;
38     double temp;
39
40     ref_rms = calculate_RMS (ref_buf,BUF_SIZE/2);
41     // printf ("reference rms: %f\n",ref_rms);
42
43     mic_rms = calculate_RMS (mic_buf,BUF_SIZE/2);
44     // printf ("mic rms: %f\n",mic_rms);
45
46     if((mic_rms-ref_rms)==0)
47     {
48         gain_factor=1;
49     }
50
51     else if((mic_rms-ref_rms)<0)
52     {
53         gain_factor=1;
54     }
55     else
56     {
57         temp=1/(mic_rms - ref_rms);
58         if (temp > 1)
59             gain_factor=1;
60
61         else
62             gain_factor=temp;
63     }
64     // printf ("gain factor %f\n\n", gain_factor);
```

```

65     return gain_factor;
66
67 }
68
69 /**
70  * Split stereo buffer into two mono buffers
71  * @param stereo_buf pointer to the stereo buffer
72  * @param left pointer to left mono buffer
73  * @param right pointer to right mono buffer
74  */
75 void split_stereo (signed short* stereo_buf, signed short* left, signed short* right)
76 {
77     int i;
78     for (i = 0; i < BUF_SIZE/2; i++)
79     {
80         left[i]=stereo_buf[2*i];
81         right[i]=stereo_buf[2*i+1];
82     }
83 }
84
85 void filter_input (signed short* mic_input, signed short* ref_input, double* filter_gains, int
86     number_of_filters)
87 {
88     signed short filtered_mic_input[BUF_SIZE/2];
89     signed short filtered_ref_input[BUF_SIZE/2];
90     int i;
91     for (i = 0; i < number_of_filters; ++i)
92     {
93         butt6bp1(mic_input, filtered_mic_input, BUF_SIZE/2, i, 1);
94         butt6bp2(ref_input, filtered_ref_input, BUF_SIZE/2, i, 1);
95         filter_gains[i]= calculate_coeff(filtered_ref_input, filtered_mic_input);
96     }
97 }
98
99
100
101 //-----Main-----
102
103
104 int main (int argc, char *argv[])
105 {
106     double filter_gains[16];
107     filter_gains[15]=1;
108     signed short stereo_input[BUF_SIZE];
109     signed short mic_input[BUF_SIZE/2];
110     signed short ref_input[BUF_SIZE/2];
111
112     snd_pcm_t *capture_handle;
113     snd_pcm_t *playback_handle;
114     snd_pcm_hw_params_t *capture_params;
115     snd_pcm_hw_params_t *playback_params;
116     signed short output_buf[BUF_SIZE/2];
117
118     //-----ALSA setup-----
119
120     open_pcm(&capture_handle, PCM, SND_PCM_STREAM_CAPTURE, 0);
121     snd_pcm_hw_params_malloc (&capture_params);
122     snd_pcm_hw_params_any (capture_handle, capture_params);
123     set_params(capture_handle, capture_params, STERIO, RATE);
124     write_params(capture_handle, capture_params);
125     prepair_interface(capture_handle);
126     snd_pcm_hw_params_free (capture_params);
127
128
129
130     open_pcm(&playback_handle, PCM, SND_PCM_STREAM_PLAYBACK, 0);
131     snd_pcm_hw_params_malloc (&playback_params);
132     snd_pcm_hw_params_any (playback_handle, playback_params);
133     set_params(playback_handle, playback_params, MONO, RATE);
134     write_params(playback_handle, playback_params);
135     prepair_interface(playback_handle);
136     snd_pcm_hw_params_free (playback_params);

```

```

137
138     for (; ;)
139     {
140         record(capture_handle, stereo_input, BUF_SIZE/2);
141         split_stereo(stereo_input, mic_input, ref_input);
142         filter_input(mic_input, ref_input, filter_gains, NUM_OF_FILTERS-1);
143         apply_all_filters(ref_input, output_buf, BUF_SIZE/2, filter_gains);
144         prepair_interface(playback_handle);
145         play(playback_handle, output_buf, BUF_SIZE/2);
146     }
147     return 0;
148 }

```

Listing D.1: main.c

```

1  #include <alsa/asoundlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <math.h>
5
6
7
8  //-----defines-----
9  #define PCM "plughw:1,0"
10 #define BUF_SIZE 2048 /* 8192 */ /* 16384 */ /* 32768 */ /* 65536 */
11 #define STERIO 2
12 #define MONO 1
13 #define RATE 44100
14
15
16 //-----prototypes-----
17
18 /**
19  * Open the PCM device in playback mode
20  * and writes error if device
21  * @param **pcm_handle handle to pcm
22  * @param *card audio card to play to
23  * @param stream Wanted stream
24  * @param mode Open mode
25  */
26 void open_pcm (snd_pcm_t **pcm_handle,
27               char *card,
28               snd_pcm_stream_t stream,
29               int mode);
30
31
32 /**
33  * Restrict a configuration space to contain only
34  * one access type
35  * and writes an error if interleaved mode can't be set
36  * @param **pcm_handle handle to pcm
37  * @param *params configuration space
38  */
39 void set_access (snd_pcm_t *pcm_handle,
40                snd_pcm_hw_params_t *params);
41
42 /**
43  * Restrict a configuration space to contain only one format
44  * and writes an error if format can't be set
45  * @param **pcm_handle handle to pcm
46  * @param *params configuration space
47  */
48 void set_format (snd_pcm_t *pcm_handle,
49                 snd_pcm_hw_params_t *params);
50
51
52 /**
53  * Restrict a configuration space to one channels count
54  * and writes an error if channels number can't be set
55  * @param **pcm_handle handle to pcm
56  * @param *params configuration space
57  * @param channels channels count
58  */

```

```

59 void set_channels(snd_pcm_t *pcm_handle,
60                 snd_pcm_hw_params_t *params,
61                 int channels);
62
63
64 /**
65  * Restrict a configuration space to contain only one rate
66  * and writes an error if rate can't be set
67  * @param **pcm_handle handle to pcm
68  * @param *params configuration space
69  * @param rate approximate rate
70  */
71 void set_rate(snd_pcm_t *pcm_handle,
72              snd_pcm_hw_params_t *params,
73              int rate);
74
75
76 /**
77  * Set hardware parameters
78  * and writes an error if parameters can't be set
79  * @param **pcm_handle handle to pcm
80  * @param *params configuration space
81  * @param channels channels count
82  * @param rate approximate rate
83  */
84 void set_params(snd_pcm_t *pcm_handle,
85                snd_pcm_hw_params_t *params,
86                int channels,
87                int rate);
88
89
90 /**
91  * Write hardware parameters
92  * and writes an error if hardware parameters can't be set
93  * @param *pcm_handle handle to the pcm
94  * @param *params configuration space
95  */
96 void write_params(snd_pcm_t *pcm_handle,
97                  snd_pcm_hw_params_t *params);
98
99
100 /**
101  * Prepare audio interface for use
102  * and writes an error if it can't be prepared
103  * @param *pcm_handle handle to the pcm
104  */
105 void prepare_interface(snd_pcm_t *pcm_handle);
106
107
108 /**
109  * Extracts period time from a configuration space
110  * and writes an error if period time can't be extracted
111  * @param *params configuration space
112  * @return tmp approximate period duration
113  */
114 unsigned int get_period_time(snd_pcm_hw_params_t *params);
115
116
117 /**
118  * Write contents of a buffer to the PCM
119  * write error when unable to write to PCM device
120  * @param **pcm_handle handle to playback
121  * @param frames period size
122  * @param *buffer pointer to buffer
123  * @param buff_size size of the buffer
124  */
125 void play(snd_pcm_t *pcm_handle,
126          signed short *buff,
127          int buffer_size);
128
129
130 /**
131  * Reading from audio card to the buffer
132  * write errors too

```



```

132 * @param **pcm_handle handle to playback
133 * @param frames period size
134 * @param period length of period
135 * @param *buffer pointer to buffer
136 * @param buff_size size of the buffer
137 */
138 void record(snd_pcm_t *pcm_handle,
139             signed short *buff,
140             snd_pcm_uframes_t frames);

```

Listing D.2: mypcm.h

```

1  #include <alsa/asoundlib.h>
2  #include "mypcm.h"
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <math.h>
6
7
8  /**
9   * Open the PCM device in playback mode
10  * and writes error if device
11  * @param **pcm_handle handle to pcm
12  * @param *card audio card to play to
13  * @param stream Wanted stream
14  * @param mode Open mode
15  */
16 void open_pcm (snd_pcm_t **pcm_handle,
17                char *card,
18                snd_pcm_stream_t stream,
19                int mode)
20 {
21     unsigned int pcm;
22     pcm = snd_pcm_open(pcm_handle, card, stream, 0);
23     if (pcm < 0)
24     {
25         printf("ERROR: Can't open \"%s\" PCM device. %s\n",
26               card, snd_strerror(pcm));
27         exit(1);
28     }
29 }
30
31
32
33 /**
34  * Restrict a configuration space to contain only
35  * one access type
36  * and writes an error if interleaved mode can't be set
37  * @param **pcm_handle handle to pcm
38  * @param *params configuration space
39  */
40 void set_access (snd_pcm_t *pcm_handle,
41                  snd_pcm_hw_params_t *params)
42 {
43     unsigned int pcm;
44     pcm = snd_pcm_hw_params_set_access(pcm_handle, params,
45                                       SND_PCM_ACCESS_RW_INTERLEAVED);
46     if (pcm < 0)
47     {
48         printf("ERROR: Can't set interleaved mode. %s\n",
49               snd_strerror(pcm));
50         exit(1);
51     }
52 }
53
54 /**
55  * Restrict a configuration space to contain only one format
56  * and writes an error if format can't be set
57  * @param **pcm_handle handle to pcm
58  * @param *params configuration space
59  */
60 void set_format (snd_pcm_t *pcm_handle,
61                  snd_pcm_hw_params_t *params)

```

```

62 {
63     unsigned int pcm;
64     pcm = snd_pcm_hw_params_set_format(pcm_handle, params,
65                                         SND_PCM_FORMAT_S16_LE);
66     if (pcm < 0)
67     {
68         printf("ERROR: Can't set format. %s\n",
69               snd_strerror(pcm));
70         exit(1);
71     }
72 }
73
74
75 /**
76  * Restrict a configuration space to one channels count
77  * and writes an error if channels number can't be set
78  * @param **pcm_handle handle to pcm
79  * @param *params configuration space
80  * @param channels channels count
81  */
82 void set_channels(snd_pcm_t *pcm_handle,
83                  snd_pcm_hw_params_t *params,
84                  int channels)
85 {
86     unsigned int pcm;
87     pcm = snd_pcm_hw_params_set_channels(pcm_handle, params, channels);
88     if (pcm < 0)
89     {
90         printf("ERROR: Can't set channels number. %s\n",
91               snd_strerror(pcm));
92         exit(1);
93     }
94 }
95
96
97 /**
98  * Restrict a configuration space to contain only one rate
99  * and writes an error if rate can't be set
100  * @param **pcm_handle handle to pcm
101  * @param *params configuration space
102  * @param rate approximate rate
103  */
104 void set_rate(snd_pcm_t *pcm_handle,
105              snd_pcm_hw_params_t *params,
106              int rate)
107 {
108     unsigned int pcm;
109     pcm = snd_pcm_hw_params_set_rate(pcm_handle, params, rate, 0);
110     if (pcm < 0)
111     {
112         printf("ERROR: Can't set rate. %s\n",
113               snd_strerror(pcm));
114         exit(1);
115     }
116 }
117
118
119 /**
120  * Set hardware parameters
121  * and writes an error if parameters can't be set
122  * @param **pcm_handle handle to pcm
123  * @param *params configuration space
124  * @param channels channels count
125  * @param rate approximate rate
126  */
127 void set_params(snd_pcm_t *pcm_handle,
128                snd_pcm_hw_params_t *params,
129                int channels,
130                int rate)
131 {
132     set_access(pcm_handle, params);
133     set_format(pcm_handle, params);
134     set_channels(pcm_handle, params, channels);

```

```

135     set_rate(pcm_handle, params, rate);
136 }
137
138
139 /**
140  * Write hardware parameters
141  * and writes an error if hardware parameters can't be set
142  * @param *pcm_handle handle to the pcm
143  * @param *params configuration space
144  */
145 void write_params(snd_pcm_t *pcm_handle,
146                  snd_pcm_hw_params_t *params)
147 {
148     unsigned int pcm;
149     pcm = snd_pcm_hw_params(pcm_handle, params);
150     if (pcm < 0)
151     {
152         printf("ERROR: Can't set hardware parameters. %s\n",
153               snd_strerror(pcm));
154         exit(1);
155     }
156 }
157
158 /**
159  * Prepare audio interface for use
160  * and writes an error if it can't be prepared
161  * @param *pcm_handle handle to the pcm
162  */
163 void prepare_interface(snd_pcm_t *pcm_handle)
164 {
165     unsigned int pcm;
166     pcm = snd_pcm_prepare(pcm_handle);
167     if (pcm < 0)
168     {
169         fprintf(stderr, "cannot prepare audio interface for use (%s)\n",
170               snd_strerror(pcm));
171         exit(1);
172     }
173 }
174
175 /**
176  * Extracts period time from a configuration space
177  * and writes an error if period time can't be extracted
178  * @param *params configuration space
179  * @return tmp approximate period duration
180  */
181 unsigned int get_period_time(snd_pcm_hw_params_t *params)
182 {
183     unsigned int tmp, pcm;
184     pcm = snd_pcm_hw_params_get_period_time(params, &tmp, NULL);
185     if (pcm < 0)
186     {
187         printf("ERROR: Can't extract maximum period time. %s\n",
188               snd_strerror(pcm));
189         exit(1);
190     }
191     return tmp;
192 }
193
194 /**
195  * Write contents of a buffer to the PCM
196  * write error when unable to write to PCM device
197  * @param **pcm_handle handle to playback
198  * @param frames period size
199  * @param *buffer pointer to buffer
200  * @param buff_size size of the buffer
201  */
202 void play(snd_pcm_t *pcm_handle,
203           signed short *buff,
204           int buffer_size)
205 {
206     unsigned int pcm;
207     pcm = snd_pcm_writew(pcm_handle, buff, buffer_size);

```

```

208     if (pcm == -EPIPE)
209     {
210         printf("ERROR: an underrun occurred\n");
211         snd_pcm_prepare(pcm_handle);
212     }
213     if (pcm < 0)
214     {
215         printf("ERROR. Can't write to PCM device. %s\n",
216             snd_strerror(pcm));
217         exit(1);
218     }
219 }
220
221
222
223 /**
224  * Reading from audio card to the buffer
225  * write errors too
226  * @param **pcm_handle handle to playback
227  * @param frames period size
228  * @param period length of period
229  * @param *buffer pointer to buffer
230  * @param buff_size size of the buffer
231  */
232 void record(snd_pcm_t *pcm_handle,
233             signed short *buff,
234             snd_pcm_uframes_t frames)
235 {
236     unsigned int pcm;
237     pcm = snd_pcm_readi (pcm_handle, buff, frames);
238     if (pcm != frames)
239     {
240         fprintf (stderr, "ERROR: read from audio interface failed (%s)\n",
241             snd_strerror (pcm));
242         exit (1);
243     }
244     if (pcm == -EBADFD)
245     {
246         fprintf (stderr, "ERROR: PCM is not in the right state (%s)\n",
247             snd_strerror (pcm));
248         exit (1);
249     }
250     if (pcm == -EPIPE)
251     {
252         fprintf (stderr, "ERROR: an overrun occurred (%s)\n",
253             snd_strerror (pcm));
254         exit (1);
255     }
256     if (pcm == -ESTRPIPE)
257     {
258         fprintf (stderr, "ERROR: a suspend event occurred (%s)\n",
259             snd_strerror (pcm));
260         exit (1);
261     }
262 }
263 }

```

Listing D.3: mypcm.c

```

1  /**
2  * Implementation of 6th order Butterworth band pass filter.
3  *
4  * @author Søren B. Nørgaard
5  * @date 2013-11-11
6  */
7
8  #define MAX_LINE_LEN 30
9  #define NUM_OF_FILTERS 16
10 #define MAXGAIN 0.95
11 #define MINGAIN 0.2
12 #define F14MAXGAIN 0.8
13 #define F15MAXGAIN 0.9
14

```

```

15
16
17 /*
18  * Sample rate: 44100 Hz
19  * Filter coeffs: 39, 50, 63, 79, 99, 125, 157, 198, 250, 315, 397, 500, 630, 794, 1000, 1235 (hp)
    Hz
20 */
21 static const double k[NUM_OF_FILTERS] = {2.659367798839737e-10,5.601922571251911e
    -10,1.120115075688814e-09,2.207462238523890e-09,4.341416445165228e-09,8.731403398760432e
    -09,1.728206087410670e-08,3.461835287058210e-08,6.956463822613412e-08,1.388587192625464e
    -07,2.772330298352829e-07,5.519738778855591e-07,1.099473480985984e-06,2.189284544056890e
    -06,4.344426201580417e-06,7.114276985396413e-01};
22
23 static const double b01[NUM_OF_FILTERS] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
24 static const double b11[NUM_OF_FILTERS] = {-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2};
25 static const double b21[NUM_OF_FILTERS] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
26 static const double a11[NUM_OF_FILTERS] = {1.999254893262780e+00,1.999031193373356e
    +00,1.998759143806201e+00,1.998412901137566e+00,1.997962397703144e+00,1.997347360271839e
    +00,1.996544830960514e+00,1.995443198012516e+00,1.993927584299521e+00,1.991847178583609e
    +00,1.988928732726706e+00,1.984799688788573e+00,1.978855077139464e+00,1.970195247752909e
    +00,1.957497456588057e+00,1.684322653006238e+00};
27 static const double a21[NUM_OF_FILTERS] = {-9.992926112858296e-01,-9.990931825011605e
    -01,-9.988575458503474e-01,-9.985676089307506e-01,-9.982053083258046e-01,-9.977345186383979e
    -01,-9.971553989286375e-01,-9.964139111863375e-01,-9.954743177319568e-01,-9.943011450069916e
    -01,-9.928232607286578e-01,-9.909702966601559e-01,-9.886371212065588e-01,-9.857027277765992e
    -01,-9.820314903483434e-01,-7.107375583308553e-01};
28
29 static const double b02[NUM_OF_FILTERS] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
30 static const double b12[NUM_OF_FILTERS] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-2};
31 static const double b22[NUM_OF_FILTERS] = {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,1};
32 static const double a12[NUM_OF_FILTERS] = {1.998683276491562e+00,1.998301043512366e
    +00,1.997843165472574e+00,1.997270485968592e+00,1.996540469688058e+00,1.995567940406467e
    +00,1.994334549015705e+00,1.992695615644863e+00,1.990522409584286e+00,1.987657618375656e
    +00,1.983809392459594e+00,1.978607175224339e+00,1.971459097009606e+00,1.961522172820266e
    +00,1.947602365475845e+00,1.752228186689679e+00};
33 static const double a22[NUM_OF_FILTERS] = {-9.987141319072618e-01,-9.983517499111645e
    -01,-9.979236495164594e-01,-9.973970081877148e-01,-9.967390963108960e-01,-9.958844574647879e
    -01,-9.948335959626691e-01,-9.934887914962190e-01,-9.917857830550013e-01,-9.896610927328785e
    -01,-9.869871374885393e-01,-9.836384874869053e-01,-9.794279789666762e-01,-9.741414201918076e
    -01,-9.67540329353541e-01,-7.797080413219983e-01};
34
35 static const double b03[NUM_OF_FILTERS] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
36 static const double b13[NUM_OF_FILTERS] = {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,-2};
37 static const double b23[NUM_OF_FILTERS] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
38 static const double a13[NUM_OF_FILTERS] = {1.999395853999915e+00,1.999216384139402e
    +00,1.998999140685960e+00,1.998724118188818e+00,1.998368479893290e+00,1.997886458021087e
    +00,1.997262659200254e+00,1.996414203353910e+00,1.995258673142130e+00,1.993689494811716e
    +00,1.991512493364511e+00,1.988466571507734e+00,1.984128668706655e+00,1.977873752705908e
    +00,1.968786741748784e+00,1.883771483682073e+00};
39 static const double a23[NUM_OF_FILTERS] = {-9.99421116542249e-01,-9.992578955771124e
    -01,-9.990650377548419e-01,-9.988277245149526e-01,-9.985311605539993e-01,-9.981457555329134e
    -01,-9.976716082524942e-01,-9.970644223042499e-01,-9.962948360575664e-01,-9.953336324915507e
    -01,-9.941222481332931e-01,-9.926024975216743e-01,-9.906872369088432e-01,-9.882754417837817e
    -01,-9.852525379431085e-01,-9.133143063151730e-01};
40
41 /*
42  * Gain factors for filters. Initialized to 1, but changed during runtime.
43  * TODO: These should be protected if they are written from at one point
44  * and read at an other. Gains should be 0 <= gain <= 1.
45  */
46
47
48 /**
49  * 6th order Butterworth band pass filter.
50  * @param *input Pointer to input signal buffer.
51  * @param *output Pointer to output signal buffer.
52  * @param len Length of buffer.
53  * @param f Number of filter, e.g. f=14 is center freq. 1000 Hz.
54  * @param g Filter gain 0<=g<=1. 0=no input, 1=unity gain,.
55  */
56 void butt6bp(signed short *input, signed short *output, int len, int f, double g);
57
58

```

```

59 /**
60  * 6th order Butterworth band pass filter.
61  * @param *input Pointer to input signal buffer.
62  * @param *output Pointer to output signal buffer.
63  * @param len Length of buffer.
64  * @param f Number of filter, e.g. f=14 is center freq. 1000 Hz.
65  * @param g Filter gain 0<=g<=1. 0=no input, 1=unity gain,.
66  */
67 void butt6bp1(signed short *input, signed short *output, int len, int f, double g);
68
69 /**
70  * 6th order Butterworth band pass filter.
71  * @param *input Pointer to input signal buffer.
72  * @param *output Pointer to output signal buffer.
73  * @param len Length of buffer.
74  * @param f Number of filter, e.g. f=14 is center freq. 1000 Hz.
75  * @param g Filter gain 0<=g<=1. 0=no input, 1=unity gain,.
76  */
77 void butt6bp2(signed short *input, signed short *output, int len, int f, double g);
78
79
80 /**
81  * Apply all 15 butt6bp filters to input and store in output.
82  * @param *input Pointer to input signal in buffer.
83  * @param *output Pointer to output signal buffer.
84  */
85 void apply_all_filters(signed short *input, signed short *output, int size, double* filter_gains);
86
87 /**
88  * Print a buffer of signed short.
89  * @param *buffer Pointer to the signed short buffer to be printed.
90  * @param len Length of the buffer to be printed.
91  */
92 void print_short_buffer(signed short *buffer, int len);

```

Listing D.4: butt6bp.h

```

1 /**
2  * Implementation of 6th order Butterworth band pass filter.
3  *
4  * @author Søren B. Nørgaard
5  * @date 2013-11-11
6  */
7 #include <stdio.h>
8 #include <string.h>
9 #include "butt6bp.h"
10
11 /**
12  * Local Functions
13  * *****/
14 /**
15  * Sets every element of an array equal to zero.
16  * @param *arr Array to be cleared.
17  * @param len Length of array.
18  */
19 static void clear_array(signed short *arr, int len)
20 {
21     int i;
22     for (i = 0; i < len; i++)
23     {
24         arr[i] = 0;
25     }
26 }
27
28 /**
29  * Global Functions
30  * *****/
31 void butt6bp(signed short *input, signed short *output, int len, int f, double g)
32 {
33     int i;
34     static double w1[NUM_OF_FILTERS] = {0};
35     static double w1_1[NUM_OF_FILTERS] = {0};
36     static double w1_2[NUM_OF_FILTERS] = {0};

```

```

37     static double w2[NUM_OF_FILTERS] = {0};
38     static double w2_1[NUM_OF_FILTERS] = {0};
39     static double w2_2[NUM_OF_FILTERS] = {0};
40     static double w3[NUM_OF_FILTERS] = {0};
41     static double w3_1[NUM_OF_FILTERS] = {0};
42     static double w3_2[NUM_OF_FILTERS] = {0};
43     static double y = 0;
44     static double x = 0;
45     static double y1 = 0;
46     static double y2 = 0;
47     static double y3 = 0;
48
49     if (g < 0)
50         g = 0;
51     else if (g < MINGAIN)
52         g = MINGAIN;
53     else if (MAXGAIN < g)
54         g = MAXGAIN;
55
56     /*
57     * XXX: This should probably be moved to the
58     * controller part instead of inside the filter.
59     */
60     if (f == 14 && F14MAXGAIN < g) // Numbers based on tests.
61         g = F14MAXGAIN;
62     else if (f == 15 && F15MAXGAIN < g)
63         g = F15MAXGAIN;
64
65     for (i = 0; i < len; i++)
66     {
67         x = (double)input[i] * k[f] * g;
68
69         w1[f] = a11[f]*w1_1[f] + a21[f]*w1_2[f] + x;
70         y1 = b01[f]*w1[f] + b11[f]*w1_1[f] + b21[f]*w1_2[f];
71         w2[f] = a12[f]*w2_1[f] + a22[f]*w2_2[f] + y1;
72         y2 = b02[f]*w2[f] + b12[f]*w2_1[f] + b22[f]*w2_2[f];
73         w3[f] = a13[f]*w3_1[f] + a23[f]*w3_2[f] + y2;
74         y3 = b03[f]*w3[f] + b13[f]*w3_1[f] + b23[f]*w3_2[f];
75
76         y = y3;
77         output[i] = (signed short)y;
78
79         w3_2[f] = w3_1[f];
80         w3_1[f] = w3[f];
81         w2_2[f] = w2_1[f];
82         w2_1[f] = w2[f];
83         w1_2[f] = w1_1[f];
84         w1_1[f] = w1[f];
85     }
86 }
87
88 void butt6bp1(signed short *input, signed short *output, int len, int f, double g)
89 {
90     int i;
91     static double w1[NUM_OF_FILTERS] = {0};
92     static double w1_1[NUM_OF_FILTERS] = {0};
93     static double w1_2[NUM_OF_FILTERS] = {0};
94     static double w2[NUM_OF_FILTERS] = {0};
95     static double w2_1[NUM_OF_FILTERS] = {0};
96     static double w2_2[NUM_OF_FILTERS] = {0};
97     static double w3[NUM_OF_FILTERS] = {0};
98     static double w3_1[NUM_OF_FILTERS] = {0};
99     static double w3_2[NUM_OF_FILTERS] = {0};
100     static double y = 0;
101     static double x = 0;
102     static double y1 = 0;
103     static double y2 = 0;
104     static double y3 = 0;
105
106     if (g < 0)
107         g = 0;
108     else if (g < MINGAIN)
109         g = MINGAIN;

```

```

110     else if (MAXGAIN < g)
111         g = MAXGAIN;
112
113     /*
114     * XXX: This should probably be moved to the
115     * controller part instead of inside the filter.
116     */
117     if (f == 14 && F14MAXGAIN < g) // Numbers based on tests.
118         g = F14MAXGAIN;
119     else if (f == 15 && F15MAXGAIN < g)
120         g = F15MAXGAIN;
121
122
123     for (i = 0; i < len; i++)
124     {
125         x = (double)input[i] * k[f] * g;
126
127         w1[f] = a11[f]*w1_1[f] + a21[f]*w1_2[f] + x;
128         y1     = b01[f]*w1[f]    + b11[f]*w1_1[f] + b21[f]*w1_2[f];
129         w2[f] = a12[f]*w2_1[f] + a22[f]*w2_2[f] + y1;
130         y2     = b02[f]*w2[f]    + b12[f]*w2_1[f] + b22[f]*w2_2[f];
131         w3[f] = a13[f]*w3_1[f] + a23[f]*w3_2[f] + y2;
132         y3     = b03[f]*w3[f]    + b13[f]*w3_1[f] + b23[f]*w3_2[f];
133
134         y = y3;
135         output[i] = (signed short)y;
136
137         w3_2[f] = w3_1[f];
138         w3_1[f] = w3[f];
139         w2_2[f] = w2_1[f];
140         w2_1[f] = w2[f];
141         w1_2[f] = w1_1[f];
142         w1_1[f] = w1[f];
143     }
144 }
145
146 void butt6bp2(signed short *input, signed short *output, int len, int f, double g)
147 {
148     int i;
149     static double w1[NUM_OF_FILTERS] = {0};
150     static double w1_1[NUM_OF_FILTERS] = {0};
151     static double w1_2[NUM_OF_FILTERS] = {0};
152     static double w2[NUM_OF_FILTERS] = {0};
153     static double w2_1[NUM_OF_FILTERS] = {0};
154     static double w2_2[NUM_OF_FILTERS] = {0};
155     static double w3[NUM_OF_FILTERS] = {0};
156     static double w3_1[NUM_OF_FILTERS] = {0};
157     static double w3_2[NUM_OF_FILTERS] = {0};
158     static double y = 0;
159     static double x = 0;
160     static double y1 = 0;
161     static double y2 = 0;
162     static double y3 = 0;
163
164     if (g < 0)
165         g = 0;
166     else if (g < MINGAIN)
167         g = MINGAIN;
168     else if (MAXGAIN < g)
169         g = MAXGAIN;
170
171     /*
172     * XXX: This should probably be moved to the
173     * controller part instead of inside the filter.
174     */
175     if (f == 14 && F14MAXGAIN < g) // Numbers based on tests.
176         g = F14MAXGAIN;
177     else if (f == 15 && F15MAXGAIN < g)
178         g = F15MAXGAIN;
179
180     for (i = 0; i < len; i++)
181     {
182         x = (double)input[i] * k[f] * g;

```



```

183
184     w1[f] = a11[f]*w1_1[f] + a21[f]*w1_2[f] + x;
185     y1    = b01[f]*w1[f]   + b11[f]*w1_1[f] + b21[f]*w1_2[f];
186     w2[f] = a12[f]*w2_1[f] + a22[f]*w2_2[f] + y1;
187     y2    = b02[f]*w2[f]   + b12[f]*w2_1[f] + b22[f]*w2_2[f];
188     w3[f] = a13[f]*w3_1[f] + a23[f]*w3_2[f] + y2;
189     y3    = b03[f]*w3[f]   + b13[f]*w3_1[f] + b23[f]*w3_2[f];
190
191     y = y3;
192     output[i] = (signed short)y;
193
194     w3_2[f] = w3_1[f];
195     w3_1[f] = w3[f];
196     w2_2[f] = w2_1[f];
197     w2_1[f] = w2[f];
198     w1_2[f] = w1_1[f];
199     w1_1[f] = w1[f];
200 }
201 }
202
203
204 void apply_all_filters(signed short *input, signed short *output, int size, double* filter_gains)
205 {
206     int i, j;
207     signed short tmp[size];
208
209     clear_array(output, size);
210     for (i = 0; i < NUM_OF_FILTERS; i++)
211     {
212         butt6bp(input, tmp, size, i, filter_gains[i]);
213         for (j = 0; j < size; j++)
214         {
215             output[j] += tmp[j];
216         }
217     }
218 }
219
220 void print_short_buffer(signed short *buffer, int len)
221 {
222     int i;
223     for (i = 0; i < len; i++)
224     {
225         printf("%d\n", buffer[i]);
226     }
227 }

```

Listing D.5: butt6bp.c

Bibliography

- [1] Andrea Goldsmith. Wireless Communications. <http://wsl.stanford.edu/~andrea/Wireless/SampleChapters.pdf>.
- [2] Bell Labs. Plan 9 on Raspberry Pi. http://www.plan9.bell-labs.com/wiki/plan9/Raspberry_Pi/index.html.
- [3] Leo L. Beranek. *Acoustical Measurements*. American Institute of Physics, 1988.
- [4] Eddy B. Brixen and Jan Voetmann. *Praktisk Elektroakustik*. Musikforlaget Fog, 2004.
- [5] Malcolm J. Crocker. *Handbook of Acoustics*. John Wiley and Sons Inc., 1998.
- [6] Digikey. Bluetooth Goes Ultra-Low-Power. <http://www.digikey.com/us/en/techzone/wireless/resources/articles/bluetooth-goes-ultra-low-power.html>.
- [7] eHow Contributor. The pros and cons of using digital technology. http://www.ehow.com/list_6383235_pros-cons-using-digital-technology.html.
- [8] FreeBSD. FreeBSD/ARM on Raspberry Pi. <https://wiki.freebsd.org/FreeBSD/arm/Raspberry%20Pi>.
- [9] IEC. *ISO IEC 60908 (1999): Audio recording - Compact disc digital audio system*. International Electrotechnical Commission, 1999.
- [10] Julius Orion Smith III. Series and parallel transfer functions. https://ccrma.stanford.edu/~jos/filters/Series_Parallel_Transfer_Functions.html.
- [11] Kai Harada. Basic Sound Principles. <http://harada-sound.com/sound/handbook/basicterms.html>, 1999.
- [12] klark teknik building. DN360 operators manual. http://www.le-brise-glace.com/medias/page_builder/DN360-Manual.pdf.
- [13] lowegian international. FIR filters. <http://www.dspguru.com/dsp/faqs/fir/basics>.
- [14] Richard G. Lyons. *Understanding Digital Signal Processing*. Prentice Hall, 2004.
- [15] Mackie. Hr624 oxners manual. <http://www.genelec.com/documents/datasheets/DS1032a.pdf>.
- [16] Genelec Active Monitoring. 1032a data sheet. http://www.google.dk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=0CDAQFjAB&url=http%3A%2F%2Fwww.mackie.com%2Fpdf%2Fhr624_om.pdf&ei=DEBRUsTFCMa24ASyyYDwDw&usg=AFQjCNHVICq9MUNv6QHeuQ31SzXStXGnSA&sig2=c8_N5GirS7NJYgofJywNkQ&bvm=bv.53537100,d.bGE.
- [17] Nokia. Detailed specifications for the Nokia Lumia 920. <http://www.nokia.com/global/products/phone/lumia920/specifications/>.
- [18] Bruel og Kjaer. Measuring sound. <http://www.bksv.com/doc/br0047.pdf>.
- [19] Ronald W. Schafer Alan V. Oppenheim. *Discrete-time Signal Processing*. Pearson, 2009.
- [20] Pioneer. Owners manual. <http://www.pioneerelectronics.com/PUSA/Unassigned-Content/Manuals/DEH-X3500UI+OWNERS+MANUAL>.
- [21] Raspberry Pi. Raspberry Pi. <http://www.raspberrypi.org/>.

- [22] Roving Networks. RN52 Bluetooth Audio Module. <http://ww1.microchip.com/downloads/en/DeviceDoc/rn-52-ds-1.0r.pdf>.
- [23] A.S. Sedra and K.C. Smith. *Microelectronic Circuits*. Oxf Ser Elec Series. Oxford University Press, Incorporated, 2010.
- [24] Nie Sen. Analog vs. digital wireless technology. http://www.tml.tkk.fi/Studies/Tik-110.300/1999/Wireless/analog_2.html, october 1999.
- [25] Dansk Standard. Electroacoustics octave band and fractional octave band filters. <http://webshop.ds.dk/templates/product.aspx?ProductGuid=43116>.
- [26] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks (5th Edition)*. Pearson, 2010.
- [27] Wikipedia. Ieee 802.11. http://en.wikipedia.org/wiki/IEEE_802.11.

Appendix CD

- Digital copy of the report—`Digital_copy_of_the_report/master.pdf`
- Source code—`Source_code/`
- Graphs and Figures—`Graphs_and_Figures/`
- Pcb's—`pcbs/`
- Datasheets—`Datasheets/`
- Downloaded web pages—`Downloaded_web_pages/`

List of Corrections