Masterarbeit

Development of a data acquisition system specialized in MEMS microphone arrays

erstellt von
Simon Gapp
Matrikel Nr.: 351468

Erstprüfer: Prof. Dr.-Ing. Ennes Sarradj
Zweitprüfer: Prof. Giuseppe Caire, Ph.D.
Betreuer: Dipl.-Ing. Gert Herold
Dr.-Ing. Andreas Kortke

Technische Universität Berlin, Fachgebiet Technische Akustik
Institut für Strömungsmechanik und Technische Akustik

17. Dezember 2020

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 17. Dezember 2020

Unterschrift

Danksagung

Ich möchte mich bei Prof. Sarradj vom Lehrstuhl für Technische Akustik und bei Prof. Caire vom Fachgebiet Theoretische Grundlagen der Kommunikationstechnik für die Möglichkeit, dieses Thema zu bearbeiten zu können, herzlich bedanken. Ein ganz besonderer Dank geht an Gert Herold und Dr. Andreas Kortke für alle Kommentare, Tipps und Tricks, praktische Unterstützung sowie die äußerst prompten Antworten auf alle meine Anfragen. Weiterer Dank geht an Simon Jekosch für die unkomplizierte Urlaubsvertretung und Torsten Daniel für die handwerkliche Unterstützung beim Messaufbau sowie an das XCore exchange Forum für alle meine XMOS Anfängerfragen.

Abstract

This thesis describes the design process of a modular Data Acquisition System (DAQ) for Micro-Electro-Mechanical Systems (MEMS) microphone arrays.
Compared to condenser microphones, MEMS microphones come at a significantly lower price, have a smaller footprint, are more resistant towards aging and temperature influences and can contain an integrated Analog-to-Digital Converter (ADC). On the other hand, they have a smaller range with a flat frequency response and usually a lower sensitivity. This results in a lower Signal-to-Noise Ratio (SNR), which can be compensated by increasing the number of microphones and cross-correlating the data in post-processing.
Solutions based on MEMS microphones usually have a fixed array shape with a defined number of microphones. The modular approach of this thesis detaches the Digital Signal Processor (DSP) unit physically from the microphone part, making it possible to adjust the array form at any time. Furthermore, it is possible to cascade multiple DSP units, making the available Universal Serial Bus (USB) bandwidth the limiting factor of the system.

Zusammenfassung

Diese Arbeit beschreibt den Entwurf eines modularen Data Acquisition System (DAQ) für Micro-Electro-Mechanical Systems (MEMS)-Mikrofonarrays.
Im Vergleich zu Kondensatormikrofonen sind MEMS Mikrofone deutlich preiswerter, haben einen geringeren Platzbedarf, sind resistenter gegen Alterung und Temperatureinflüsse und können einen bereits integrierten Analog-to-Digital Converter (ADC) enthalten. Auf der anderen Seite haben sie einen kleineren Bereich mit einem flachen Frequenzgang und meist eine geringere Empfindlichkeit. Daraus ergibt sich ein geringeres Signal-to-Noise Ratio (SNR), das aber durch eine grössere Anzahl von Mikrofonen und einer Kreuzkorrelation der Daten in der Nachbearbeitung kompensiert werden kann.
Bestehende Lösungen, die auf MEMS Mikrofonen basieren, haben meist eine feste Array Form und eine definierte Anzahl von Mikrofonen. Der modulare Ansatz dieser Arbeit trennt die Digital Signal Processor (DSP) Einheit physisch vom Mikrofonteil, sodass eine Anpassung der Array Form jederzeit möglich ist. Darüber hinaus ist es möglich, mehrere DSP Einheiten zu kaskadieren, wodurch die verfügbare Universal Serial Bus (USB) Bandbreite zum begrenzenden Faktor des Systems wird.

Contents

List of Figures

List of Tables

List of abbreviations

AC AudioControl

ACK Acknowledgement

ADC Analog-to-Digital Converter

AoIP Audio over IP

AOP Acoustic Overload Point

AS AudioStreaming

ASIC Application Specific Integrated Circuit

AVB Audio Video Bridging

CIC Cascaded-Integrator-Comb

COTS Commercial off-the-shelf

CRC Cyclic Redundancy Check

CS Control Selector

DAC Digital-to-Analog Converter

DAQ Data Acquisition System

dB Decibel

dBFS Decibel Full Scale

dBSPL Decibel Sound Pressure Level

DC Direct Current

DFU Device Firmware Upgrade

DMA Direct Memory Access

DSP Digital Signal Processor

ECL Emitter-coupled Logic

EMF Electromagnetic field

EOP End of Packet

ESD Electro Static Discharge

FFT Fast Fourier Transformation

FIFO First in, First out

FIR Finite Impulse Response Filter

FPGA Field-Programmable Gate Array

FU Feature Unit

GPIO General Purpose Input Output

GUI Graphical User Interface

HID Human Interface Device

HPF High-Pass Filter

ID Identifier

I/O Input/Output

IAD Interface Association Descriptor

IBN In-Band Noise

IC Integrated Circuit

IDE Integrated Development Environment

IFFT Inverse Fast Fourier Transformation

IIR Infinite Impulse Response Filter

ITD Input Terminal Descriptor

LDO Low Drop-out

LLVM Low Level Virtual Machine

LPF Low-Pass Filter

LSB Least Significant Bit

LVDS Low Voltage Differential Signaling

MEMS Micro-Electro-Mechanical Systems

MIPS Million Instructions Per Second

MIPS Million Instructions per Second

MSB Most Significant Bit

NRZI Non-return-to-zero, inverted

NTF Noise Transfer Function

OSR Oversampling Ratio

PCB Printed Circuit Board

PCM Pulse-Code Modulation

PDF Probability Density Function

PDM Pulse-Density Modulation

PECL Positive Emitter-coupled Logic

PHY Physical Layer

PID Packet Identifier

PLL Phase-Locked Loop

PSD Power Spectral Density

PSU Power Supply Unit

PTP Precision Timing Protocol

RTOS Real-Time Operating System

S-FTP Screen Foiled Twisted Pair

SHARC Super Harvard Architecture Single-Chip Computer

SNR Signal-to-Noise Ratio

SOF Start of Frame

SPST Single Pole Single Throw

STF Signal Transfer Function

THD Total Harmonic Distortion

TP Test Point

TSN Time Sensitive Networks

UAC1 Universal Serial Bus (USB) Audio Class 1.0

UAC2 USB Audio Class 2.0

USB Universal Serial Bus

VHDL Very High Speed Integrated Circuit Hardware Description Language

XUD XMOS USB Device Driver

Chapter 1

Introduction

A microphone array consists of several microphones placed in different locations. Typical applications of microphone arrays are speech extraction in noisy environments [1, 2] or source localization [3] often in combination with noise measurements [4]. Noise measurement applications usually consist of a larger number of microphones compared to speech extraction applications. Since the focus of this thesis is on a larger number of microphones, the introduction to this topic will be explained using examples from the field of noise measurements based on [5].
Typical measurement scenarios are in wind tunnels, during aircraft overflights and engine noise tests.
Depending on the goal of the measurement, various parameters can be adjusted, which are presented below.
The Signal-to-Noise Ratio (SNR) and the spatial resolution are defined by the microphone count. In general, it can be said that an increased number of microphones increases both, the SNR and the spatial resolution. The dynamic range, the frequency range and the sensitivity can be roughly approximated by the dimensions of a microphone. As a rule of thumb, it can be said that smaller microphones can measure higher frequencies compared to a larger microphone that usually has a higher sensitivity. The most accurate results can be achieved if the Data Acquisition System (DAQ) samples all microphones coherently.
Arranging the microphones in spirals or several circles with an odd number of regularly spaced microphones seems to give the best results.
Considering all hardware requirements, adequate results depend on the chosen algorithm. In general, the selection of the most suitable algorithm is a trade-off between spatial resolution, dynamic range and computational effort.

Microphone arrays based on Micro-Electro-Mechanical Systems (MEMS) microphones provide a cost-effective alternative to established systems based on condenser microphones. Advantages of the MEMS technology are a small footprint, low cost and a high resistance to aging and temperature influence. The microphones have, on the other hand, a smaller area with a flat frequency response due to their physical design and a lower SNR.

Commercially available arrays with up to 16 microphones are provided by [6, 7, 8]. Larger arrays are commercially available from [9]. Research projects on large scale arrays can be found in [10, 11, 12].
This thesis distinguishes between different processing stages with the terms preprocessing and post-processing. Preprocessing describes the conversion of the raw microphone signals into suitable signals for post-processing. Post-processing describes the application of an algorithm to evaluate the preprocessed microphone signals. All works presented apply preprocessing in a Digital Signal Processor (DSP) which is either a specialized microcontroller, a Field-Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC). In most of the works presented, post-processing is carried out on the same platform on which the preprocessing is performed.

A modular system is designed in this thesis in which the microphones are separated from the processing unit. The preprocessing is performed in the processing unit, while the post-processing is performed on an external device. The system consists of two physically separate units: the base board, which contains the processing unit as well as the peripherals necessary for its operation, and microphone Printed Circuit Boards (PCBs), which contain two microphones each as well as some peripherals necessary for their operation.
The term modular describes two aspects: Firstly, the possibility of operating a variable number of microphones with one processing unit and, secondly, the possibility of operating several processing units with the same clock signal for audio sampling. This overcomes the limitation of the ability to use only a single processing unit and the available transmission bandwidth becomes the limiting factor. Furthermore, the shape of the microphone array can be adjusted at any time.
In this thesis the following questions are evaluated:

- Does the modular system provide sufficiently accurate data for post-processing?
- Is it feasible to physically detach the microphones from the DSP?
- Is the selected transmission protocol suitable for transferring the array data?

1.1 Chapter overview

This thesis is divided into five chapters, covering the theoretical principles, the description of the design process and the subsequent validation of the developed system. It concludes with a summary of the work and presents suggestions for future implementations. The chapters are structured as follows:

Chapter 1 Introduces the topic and presents the research questions.

Chapter 2 Presents the operating principles of the microphones used and describes the review process for a suitable processing unit. Subsequently, details on the architecture of the selected processing unit are described. A further chapter is dedicated to the transmission protocol between a host and the processing unit and deals with the specific parts of the Universal Serial Bus (USB) protocol used in this work.

Chapter 3 Describes the implementation process for the hardware and the firmware.

Chapter 4 Provides details on the validation process of the system from an electrical and acoustic point of view.

Chapter 5 Summarizes this thesis, presents an answer to the research questions and provides suggestions for future implementations.

Chapter 2

Operating principles

This chapter presents the operating principles of the hardware and firmware that is used in this thesis. It is structured as follows: Sec. 2.1 reviews the principles that are used in a Micro-Electro-Mechanical Systems (MEMS) transducer. In Sec. 2.2, a brief review of the applied transmission techniques are presented. Sec. 2.3 describes the selection process of a suitable processing unit and Sec. 2.4 describes the architecture of the selected unit in more detail. Lastly, Sec. 2.5 summarizes the relevant parts of the selected Universal Serial Bus (USB) protocol.

2.1 Data acquisition

The used microphones are based on the MEMS technology. As illustrated in Fig. 2.1, the two main components of the microphone are a MEMS transducer and a $\Sigma\Delta$ Analog-to-Digital Converter (ADC). Components other than the MEMS transducer are summarized below by the term Application Specific Integrated Circuit (ASIC).

2.1.1 Micro-Electro-Mechanical Systems transducer

This section describes the MEMS transducer principles based on [13, 14]. The transducer can be described as a silicon capacitor consisting of two plates. While one plate is at a fixed position, the other plate is moving accordingly to the pressure change caused by a sound wave. One could describe the MEMS transducer as a capacitor with variable capacitance. The capacitance of a plate capacitor is calculated as follows:

$$C = \varepsilon_0 \varepsilon_r \frac{A}{d} \tag{2.1}$$

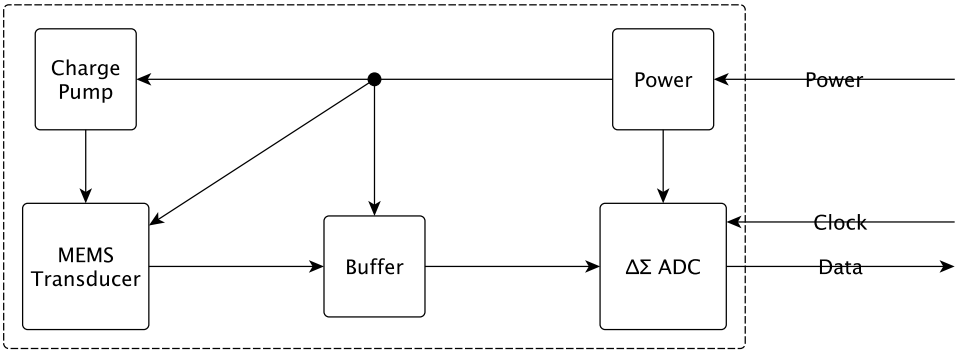


Figure 2.1: Components of a PDM MEMS Microphone, Figure from [13]

With A: Plate surface, d: Distance between plates, ε_0: vacuum permittivity and ε_r: relative permittivity. By changing the distance of the two plates, a change in capacity is achieved, which results in a change in voltage:

$$U = \frac{Q}{C} \tag{2.2}$$

With U: voltage, Q: charge and C: capacitance. The voltage change is detected by the buffer between the transducer and the $\Sigma\Delta$ ADC.

The mechanical specification for a top port MEMS microphone is depicted in Fig. 2.2. Sound waves enter from the top and pass the acoustic holes. The change in pressure causes the membrane to move, which eventually leads to a change in the capacitance of the transducer. This leads to a change in the charge of the capacitor plates, changing the input voltage at the ASIC.
Fig. 2.3 depicts a simplified model of a bottom port MEMS microphone with the ASIC and the enclosure. A bottom port microphone is also used in this thesis. The back volume is defined by the dimensions of the enclosure and the components placed inside, while the front volume is defined by the dimensions of the underlaying substrate.

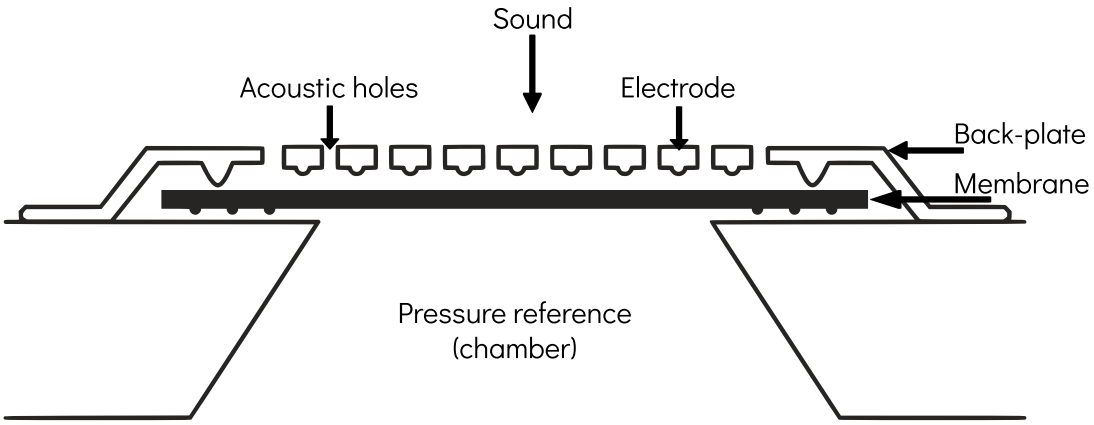


Figure 2.2: Mechanical specifications of a top port MEMS transducer, Figure from [15] licensed under CC BY 4.0

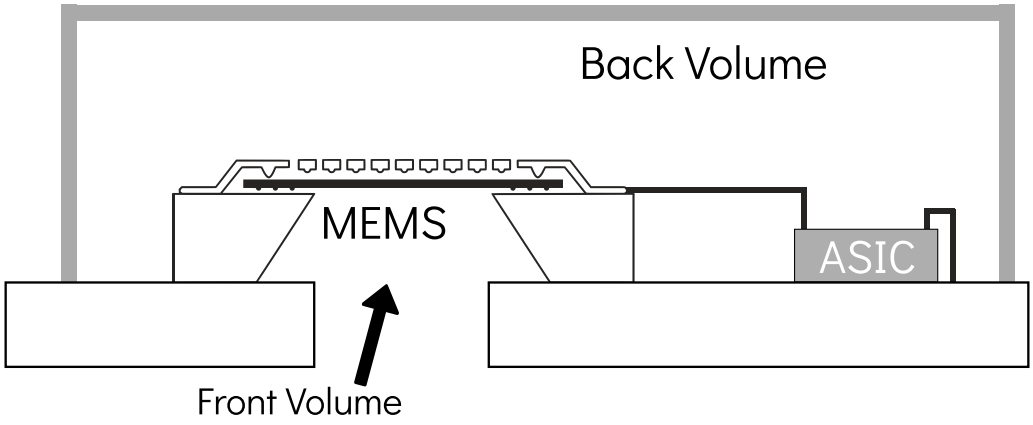


Figure 2.3: Simplified model of a bottom port MEMS microphone, MEMS part from [15] licensed under CC BY 4.0

The typical frequency response of a MEMS microphone as shown in Fig. 2.4. Two areas show a larger deviation from a flat frequency response: In the low frequency range from 10 Hz to 200 Hz and in the high frequency range above 10 kHz.

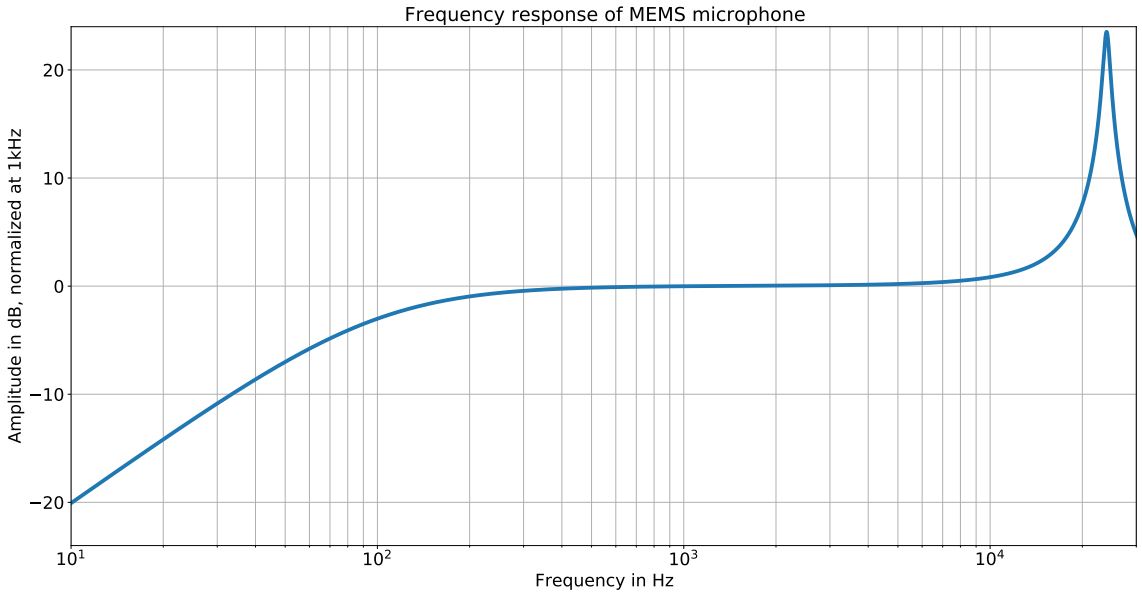


Figure 2.4: Typical frequency response of a MEMS microphone, Figure from [13]

These deviations are defined by the physical properties of the microphone. The main parameters are the form and size of the ventilation hole, the front chamber geometry and the back chamber geometry. It can be assumed that the ASIC has a flat frequency response.
While the attenuation at low frequencies is influenced by the geometry of the ventilation hole and the back chamber geometry, the gain at high frequencies is only influenced by the geometry of the front chamber. The high frequencies are amplified by a resonance peak caused by the Helmholtz effect. This effect describes the air resonance in a cavity. The resonance frequency is approximated as follows: [16]:

$$f_R \approx \frac{c}{2\pi}\sqrt{\frac{S}{V \cdot L_{\text{eff}}}} \tag{2.3}$$

With c: speed of sound, S: cross-section of the neck, V: volume of the back chamber and L_{eff}: effective length of the neck which uses a correction factor of $\Delta l = 0.85r$, with r: radius of the tube. The correction factor takes the effects at an open flange tube into account.

This approximation does not account for effects such as thermal-viscous and acoustic impedances and other material-related influences. Therefore, an approximate calculation of the resonance frequency for the microphone used is omitted.

2.1.1.1 Acoustic parameters

A microphone can be described with the following parameters:

Sensitivity The sensitivity of a digital microphone describes a percentage of the full scale output that is generated at 94 dBSPL. The value is given in Decibel Full Scale (dBFS) where the maximum value describes the highest signal level that can be output by the microphone. It is calculated as follows [17]:

$$\text{Sensitivity}_{\text{dBFS}} = 20 \cdot \log_{10} \left(\frac{\text{Sensitivity}_{\text{\%FS}}}{\text{Output}_{\text{DREF}}} \right) \tag{2.4}$$

With $\text{Output}_{\text{DREF}} = 1.0$. There is no relation between the digital output in dBFS and an analog output in dBV. The analog output is referenced to $1\,\text{V}_{\text{RMS}}$ while the digital output is referenced to the digital full scale.

Directivity The directivity of a microphone describes the variation of the sensitivity response with respect to the direction of arrival of the sound [14]. The MEMS transducer used is a pressure transducer and has an omnidirectional characteristic. This means that the sensitivity is independent from the location of the sound source and therefore identical for all locations.

Signal-to-Noise Ratio The Signal-to-Noise Ratio (SNR) specifies the ratio between a given reference signal (usually 1 kHz at 94 dBSPL) to the amount of residual noise at the microphone output [14].
The main contributor of the noise is the MEMS sensor while the noise of the ASIC can be neglected. The SNR is specified in Decibel (dB) and calculated as follows [18]:

$$\text{SNR} = 10 \cdot \log_{10} \frac{P_{\text{Signal}}}{P_{\text{Noise}}} [\text{dB}] \tag{2.5}$$

Dynamic range The dynamic range describes the difference between the minimum and maximum signal the microphone is able to generate as an output. While the minimum signal describes the signal distinctive from residual noise, the maximum audio signal is described by the microphone output with 10 % Total Harmonic Distortion (THD) at 1 kHz. This point is known as Acoustic Overload Point (AOP).

2.1.2 Analog to digital conversion

The description of the analog to digital conversion is based on [18]. The basic principles of the analog to digital conversion can be separated into two main tasks:

- Uniform sampling in time
- Uniform quantization in amplitude

Fig. 2.5 depicts the main components of an ADC. It consists of three operating blocks: an antialiasing filter, a sample and hold block and a quantizer.

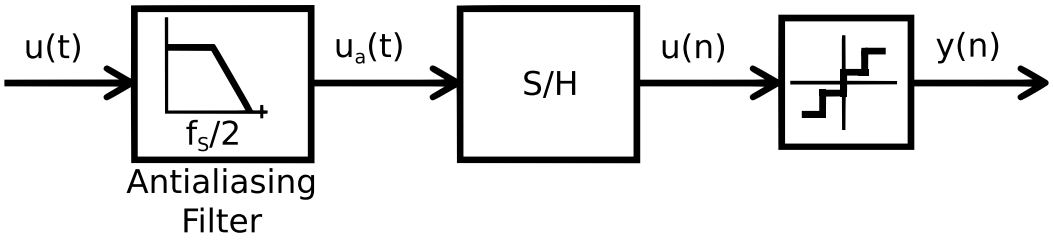


Figure 2.5: Analog to digital data converter system, Figure from [18]

With the following assumptions, it can be said that sampling in time is a completely invertible process: The signal information of the continuous input waveform $u(t)$ is contained in the signal band $|f_{\text{sig}}| \leq f_B$ where f_B is defined as the signal bandwidth. The input signal is sampled at uniform time intervals T_S or with a fixed frequency f_S, resulting in a periodicity of the original spectrum at multiples of f_s. This is illustrated in Fig. 2.6.
The periodization arises from the sampling process which is considered as a multiplication of the input signal $u(t)$ with a train of Dirac pulses spaced at $1/f_s$ in the time domain or the convolution of the signal spectrum with a periodic train of Dirac pulses spaced at f_s in the frequency domain. It can be seen in Fig. 2.6 that the original baseband spectrum can be reconstructed if the Nyquist theorem is fulfilled:

$$f_s \geq 2 \cdot f_B = f_N \tag{2.6}$$

with f_N as Nyquist frequency. This avoids overlapping and aliasing of the baseband. To only convert the bandwidth defined in Eq. 2.6, an antialiasing filter precedes the ADC.

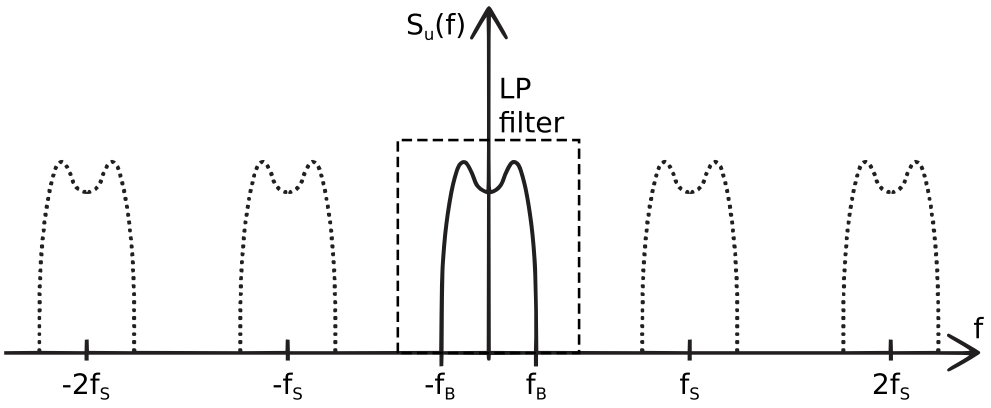


Figure 2.6: Spectrum of a sampling operation, Figure from [18]

Operating an ADC at the Nyquist frequency would result in zero transition band for the antialiasing filter to cut-off unwanted frequencies in the baseband.
Filters with a gentle roll-off are easier to design with the advantage that they require less power and introduce less phase distortion. Therefore, many ADCs work with sampling rates higher than f_N. This is known as oversampling, hence the name oversampling ADCs. The Oversampling Ratio (OSR) is defined as:

$$\mathrm{OSR} = \frac{f_s}{2 \cdot f_N} \tag{2.7}$$

Converting a continuous range of analog values into a set of discrete levels is called quantization. This is a non-invertible process, since an infinite number of input amplitude values of the sampled analog signal $u(n)$ are mapped into a finite number of output amplitude values. Even an ideal quantizer introduces an error into the output of the ADC.
The key characteristic of a quantizer is the number of Bits B_{int} which correlate to the different output levels. Mapping the analog input to $2^{B_{\mathrm{int}}}$ discrete output levels describes the resolution of the quantizer in terms of B_{int}-bits.
Assuming equally spaced levels, the quantizer operation is called uniform and has the following step width:

$$\Delta = \frac{FS}{2^{B_{\mathrm{int}}} - 1} \tag{2.8}$$

with FS: full-scale output range. The valid quantizer input range is from $\pm FS/2$ which is quantized into $2^{B_{\mathrm{int}}}$ different output levels, encoded into a binary digital

representation. The resulting quantization error is defined as the difference between the input signal and the output signal.
The gain of a single bit quantizer is not exactly defined as well as the quantization error itself. Assuming the input changes rapidly from sample to sample by amounts larger than the step size Δ, then the quantization error appears uncorrelated and has an equal probability of lying anywhere in the range of $[-\Delta/2, \Delta/2]$. This makes the assumption possible that the quantization error can have statistical properties. These properties define the basis of the additive white noise model of the quantizer which states that

- the quantization error can be expressed as a random variable.
- there is no correlation between the quantization error sequence $e(n)$ and the input $q(n)$.
- over the range $[-\Delta/2, \Delta/2]$, the Probability Density Function (PDF) of the quantization error process pdf_e is uniform.
- the Power Spectral Density (PSD) of the quantization error process $S_e(f)$ is flat

The linearized quantizer model is shown in Fig. 2.7. It represents the Input/Output (I/O) characteristics of the quantizer: a gain k_q and a quantization error $e(n)$.

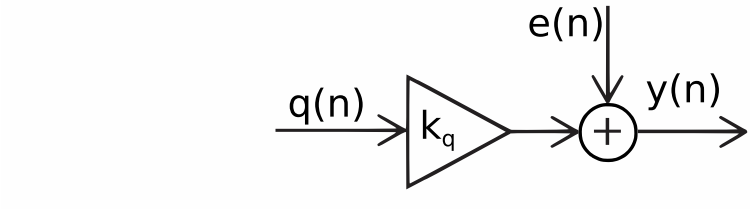


Figure 2.7: Linear quantizer model, Figure from [18]

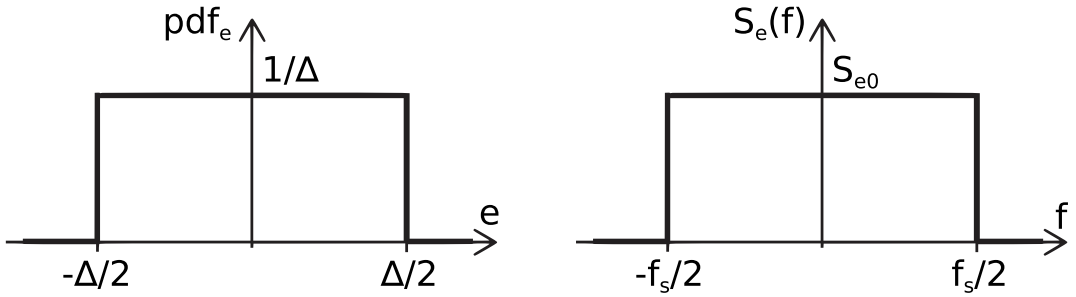
These assumptions convert the deterministic nonlinear system into a stochastic linear system with the quantization error described as quantization noise. The total quantization noise power can be calculated from Fig. 2.8(a) as follows:

$$\sigma_e^2 = \int_{-\infty}^{\infty} e^2 \mathrm{pdf}_e \partial e = \frac{\Delta^2}{12} \tag{2.9}$$

It is independent from the sampling frequency and is only determined by the quantizer resolution. The signals at the quantizer are sampled signals, which causes all quantization noise power σ_e^2 to be folded into the frequency range of $[-f_S/2, f_S/2]$. Using white noise as an approximation (compare to Fig. 2.8(b)) results in the PSD of the quantizer being:

$$S_e(f) = \frac{\Delta^2}{12}\frac{1}{f_S} \tag{2.10}$$

This means that the PSD reduces proportionally with an increased sampling frequency.



(a) Probability density function

(b) Power Spectral Density (PSD)

Figure 2.8: Quantization error according to the linear model, Figure from [18]

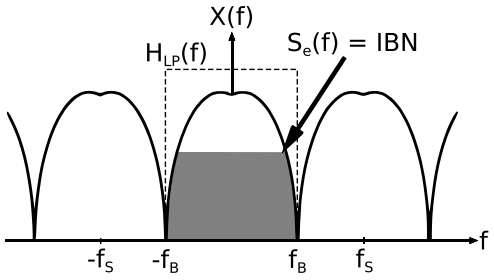
A digital low-pass filter follows the oversampled converter and eliminates all parts outside the in-band range, also leaving only a small part of the total quantization noise. The In-Band Noise (IBN) can be described as:

$$\mathrm{IBN}_{\mathrm{OSR}} = \int_{-f_B}^{f_B} S_e(f)df = \frac{\Delta^2}{12}\frac{2f_B}{f_s} = \frac{\Delta^2}{12}\frac{1}{\mathrm{OSR}} \tag{2.11}$$

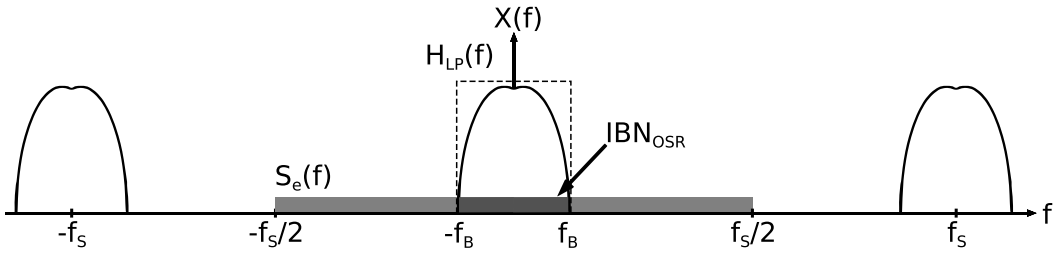
The maximum SNR of the ADC is described by the ratio between the corresponding input signal power $P_{\mathrm{FS/2}}$ and the IBN:

$$\begin{aligned}\mathrm{SNR}_{\mathrm{OSR}} &= 10 \cdot \log_{10}\left(\frac{P_{\mathrm{FS/2}}}{\mathrm{IBN}_{\mathrm{OSR}}}\right) \\ &= 6.02 \cdot B_{\mathrm{int}} + 10 \cdot \log_{10}\mathrm{OSR} + 1.76[\mathrm{dB}]\end{aligned} \tag{2.12}$$

Every doubling of the OSR increases the resolution by 3 dB or approx. 0.5 bit. The advantage of an increased OSR is depicted in Fig. 2.9.



(a) Quantization and in-band noise for OSR=1



(b) Quantization and in-band noise for OSR=4

Figure 2.9: Spectral effect of oversampling, Figure from [18]

All theory until now was based on a purely open-loop controlled system. This means that the output signal is steered by an input signal and affected by noise introduced through the conversion process.

2.1.2.1 $\Sigma\Delta$ modulation

By introducing a feedback path to the system, a closed control loop is achieved. This makes it possible to implement different transfer functions and divide the output into a Signal Transfer Function (STF) and Noise Transfer Function (NTF).
In an ideal system, the configuration of the STF and the NTF leads to a complete cancelation of the noise inside the desired signal bandwidth, with the signal itself being unaffected. Outside the bandwidth of interest, the NTF gain can be high while the STF gain can be low.

$$\mathrm{NTF}_{\mathrm{ideal}} = \overline{\mathrm{STF}_{\mathrm{ideal}}} = \begin{cases} 0, -f_B \leq f \leq f_B \\ 1, \mathrm{else} \end{cases} \tag{2.13}$$

It is shown in Eq. 2.13 that the ideal STF is described by an ideal Low-Pass Filter (LPF), while the ideal NTF is described by an ideal High-Pass Filter (HPF) with the cut-off frequency being at least at f_B. This results in the quantization noise being shaped away from the bandwidth of interest, leading to a noise shaping converter. The block diagram of a first order noise shaping converter is depicted in Fig. 2.10. Higher order noise shaping converters contain multiple integrator stages in the $\Sigma\Delta$ modulator block, which mainly increases the SNR.

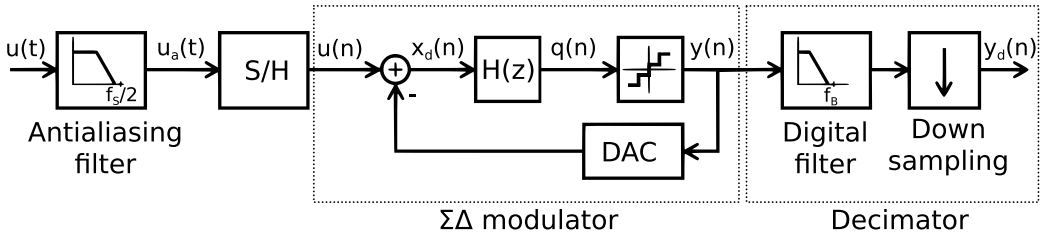


Figure 2.10: Block diagram of a first order $\Sigma\Delta$ ADC, Figure from [18]

A noise shaping converter consists of the following parts:

- **Antialiasing Filter:** The antialiasing filter eliminates spectral components above $f_s/2$ which leads to a band limited input for the modulator. This filter ensures that the subsequent sampling operation does not alias input signals from higher frequencies into the band of interest

- **Sample and Hold**: The sample and hold block performs a track and hold operation. The function of this block can be described best as a clocked switch that charges a capacitance with the input for half of the clock period and transfers this voltage level from the capacitor to the $\Sigma\Delta$ modulator during the other half of the period. Eventually, it is interpreted by the quantizer.

- $\Sigma\Delta$ **modulator:** This block performs the actual analog to digital conversion by quantizing the input signal. Furthermore, the quantization error from the internal quantizer is moved out of the band of interest. The quantizer is typically implemented with a low resolution and the internal feedback Digital-to-Analog Converter (DAC) has the same resolution, thus not introducing additional error to the system. The output of the $\Sigma\Delta$ modulator is a Pulse-Density Modulation (PDM) signal.
  A simulation of a sinusoidal input to a $\Sigma\Delta$ modulator is shown in Fig. 2.11. It

was created with the Delsig Toolbox [19] and shows that the density of zeros and ones is determined by the amplitude of the sine wave.

- **Decimator:** The decimation filter contains a digital low-pass filter to eliminate all noisy out of band components and performs the downsampling operation. A detailed description of the decimation process is presented in Sec. 2.1.2.2.

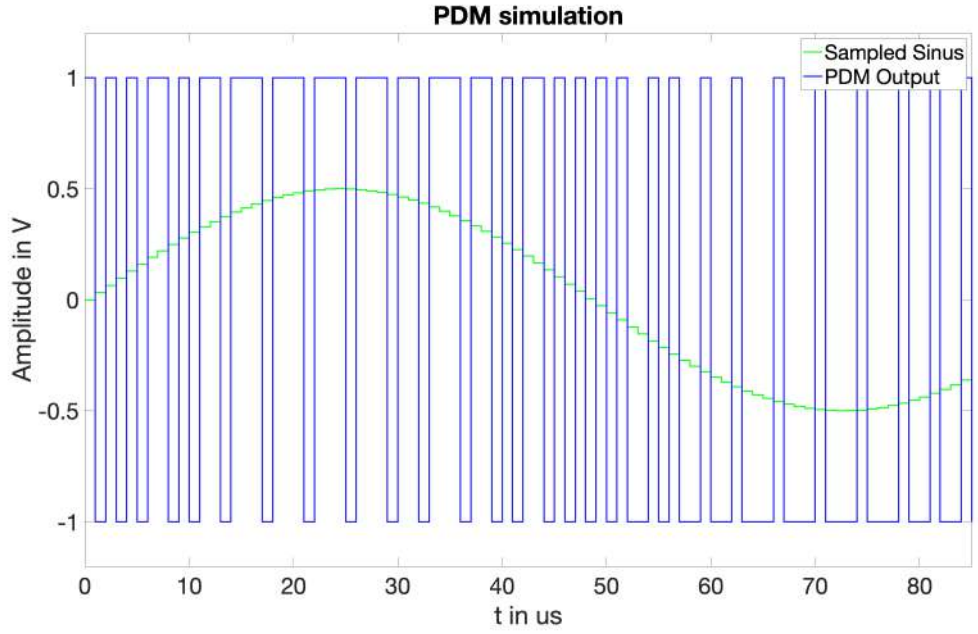


Figure 2.11: Exemplary PDM output and sampled sinus with 64x oversampling

Replacing the quantizer in Fig. 2.10 with the linearized quantizer model from Fig. 2.7 results in a linearized $\Sigma\Delta$ modulator with the following transfer function:

$$Y(z) = \mathrm{STF}(z)U(z) + \mathrm{NTF}(z)E(z) \tag{2.14}$$

with

$$\mathrm{STF}(z) = \frac{1}{\frac{1}{H(z)k_q} + 1}, \mathrm{NTF}(z) = \frac{1}{1 + H(z)k_q} \tag{2.15}$$

and $U(z)$: input signal, $Y(z)$: output signal, $U(z)$: quantization error.
The easiest filter for $H(z)$ is a first order integrator described by the following transfer

function:

$$I(z) = \frac{z^{-1}}{1 - z^{-1}} \tag{2.16}$$

For low frequencies, the linearized $\Sigma\Delta$ modulator from Eq. 2.15 becomes:

$$\mathrm{STF}(z) = z^{-1}, \mathrm{NTF}(z) \approx \frac{1 - z^{-1}}{k_q} \tag{2.17}$$

The effect of the high-pass filtered noise of the NTF is shown in Fig. 2.12. It is clearly visible that most of the noise is shaped out of the band of interest.

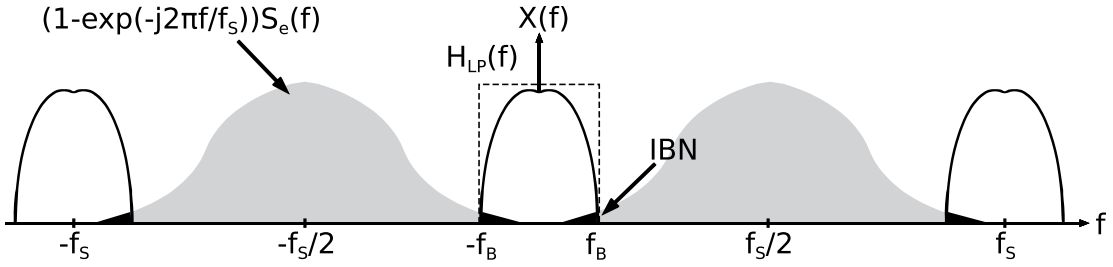


Figure 2.12: Effect of noise shaping in the $\Sigma\Delta$ modulator, Figure from [18]

The IBN can be approximated as follows:

$$\begin{aligned} \mathrm{IBN} &\approx \int_{-f_B}^{f_B} \frac{\Delta^2}{12 f_s} \frac{4\pi^2}{k_q^2} \left(\frac{f}{f_s}\right)^2 df \\ &= \frac{\Delta^2}{12} \frac{\pi^2}{3} \frac{1}{k_q^2 \mathrm{OSR}^3} \end{aligned} \tag{2.18}$$

Increasing the oversampling ratio decreases the IBN power by 9 dB per octave. The corresponding SNR is approximated as follows:

$$\mathrm{SNR} = 10 \cdot \log_{10}\left(\frac{P_{\mathrm{Sig}}}{\mathrm{IBN}_1}\right) \sim 10 \cdot \log_{10}\left(\mathrm{OSR}^3\right) \,[\mathrm{dB}] \tag{2.19}$$

This corresponds to 1.5 bit increase in resolution for doubling the OSR.

2.1.2.2 PDM to PCM conversion

For further signal processing the PDM signal from the ΣΔ modulator is converted into a Pulse-Code Modulation (PCM) signal which is suitable for post-processing. The two signal types are distinguished by their sampling rate and their bit-depth. The PDM signal has a high sampling rate and a low bit depth, while the PCM signal has a low sampling rate and a high bit-depth.
The decimation from a high sampling frequency to a low sampling frequency is performed by a Cascaded-Integrator-Comb (CIC)-Filter [20]. The filter structure is shown in Fig. 2.13. The filter is divided into two different structures: The integrator section and the comb section.

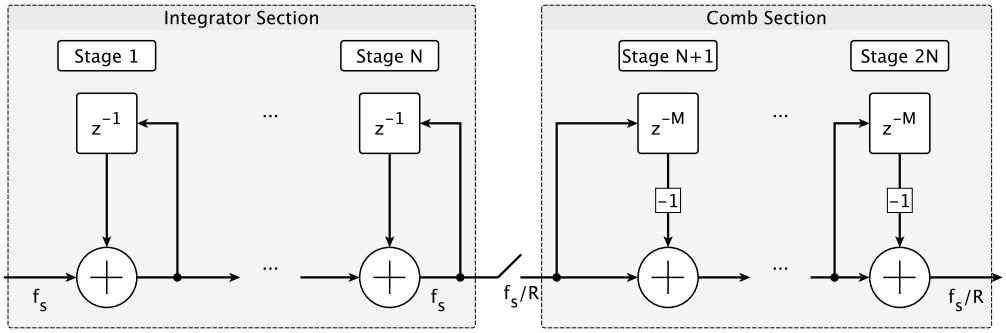


Figure 2.13: CIC decimation filter, Figure from [20]

The integrator section consists of N ideal digital integrator stages which operate at the high sampling frequency f_S. Each stage is implemented as a one-pole filter with a unity feedback coefficient. The transfer function for a single integrator stage is [20]:

$$H_I(z) = \frac{1}{1 - z^{-1}} \tag{2.20}$$

The comb section operates at the low sampling rate f_S/R. The integer R describes the rate change factor . The section consists of N comb stages with a differential delay of M samples per stage. This parameter is used to control the frequency response of the filter. The transfer function for a single comb stage is [20]:

$$H_C(z) = 1 - z^{-RM} \tag{2.21}$$

The switch between the two stages subsamples the output of the last integrator stage. This reduces the sampling rate from f_S to f_S/R. The transfer function for

the complete CIC-Filter is [20]:

$$H(z) = H_I^N(z) H_C^N(z) = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N \tag{2.22}$$

With R: Input sampling rate to output sampling rate ratio, M: number of samples per stage (usually 1 or 2) and N: Number of stages in the CIC-Filter.
Eq. 2.22 shows that the CIC-Filter is functionally equivalent to a cascade of N uniform FIR filter stages. This shows the advantage of the filter structure: The CIC-Filter does not need a multiplication unit since there are no coefficients other than ± 1 which can be easily processed with a signed binary representation. Furthermore, it uses limited storage which results in a very economic hardware implementation.
The frequency response is fully determined by the parameters R, M and N which limits the range of filter characteristics. It is recommended to use conventional filter structures to shape the frequency response after the CIC-Filter in the lower sampling rate domain where the number of multiplications per second is low.

The analog to digital conversion that is used in this project is divided into two stages. The MEMS microphone contains the $\Sigma\Delta$ modulator and sends the modulated data to the Digital Signal Processor (DSP) unit. In the DSP unit, the decimation of the signal is performed. This approach allows to specify the bit depth and sampling rate of the decimated PDM signal at runtime.

2.2 Data transmission

The microphones are physically separated from the processing unit. To ensure reliable transmission of the signals, the propagation velocity of the signals and the delay caused by the components used must be taken into account. This limits the maximum distance between the microphones and the processing unit.
The clock and data transmission are carried out over a distance of 2 m. In order to make the transmission reliable and less susceptible to external influences, the Low Voltage Differential Signaling (LVDS) principle is used for all transfers between the Printed Circuit Board (PCB)s.

All transmissions utilize CAT6 Screen Foiled Twisted Pair (S-FTP) cables with RJ45 connectors. CAT6 cables are affordable Commercial off-the-shelf (COTS) products. The cables have four twisted pairs with an impedance of 100 Ω. The cut-off frequency is at 250 MHz and the shielding is improved because each twisted pair has its own shield. The pin assignment is shown in Tab. 2.1

No	Pair	Signal
1	3	Microphone 2 Data N
2	3	Microphone 2 Data P
3	2	Clock P
4	1	GND
5	1	5 V
6	2	Clock N
7	4	Microphone 1 Data P
8	4	Microphone 1 Data N

Table 2.1: Pin assignment of the CAT6/RJ45 connector

2.2.1 Transmission line termination

In an ideal transmission line model, a wave would propagate to infinity. In the real world the transmission line has an end where the propagating wave encounters a load resistance R_L. Consider a circuit with a voltage source u_S, an output resistance R_S and a load resistance R_L as shown in Fig. 2.14. The arrows in the signal path indicate the transmission distance between source and load.

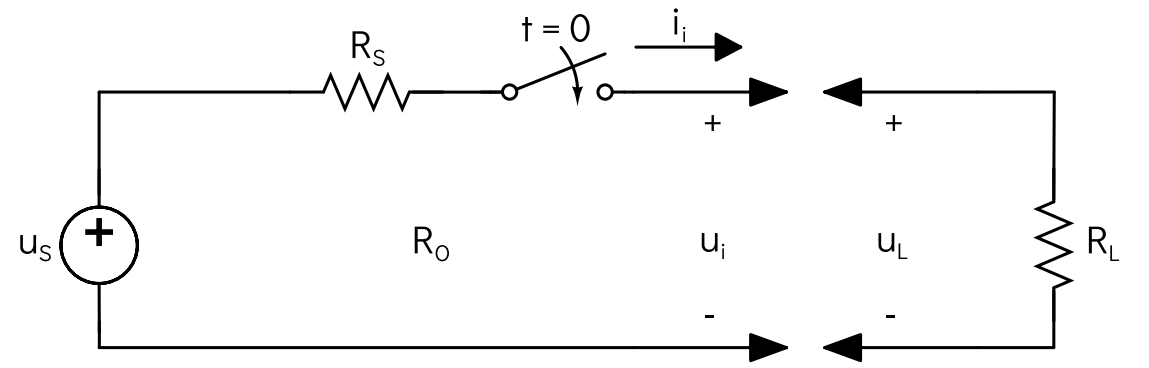


Figure 2.14: Exemplary circuit for line termination, Figure from [21]

The conservation of energy principle leads to [21]:

$$P_i = P_L + P_r \tag{2.23}$$

where:

$$P_i = \frac{u_i^2}{R_O} \tag{2.24}$$

$$P_L = \frac{u_L^2}{R_O} \tag{2.25}$$

$$P_r = \frac{u_r^2}{R_O} \tag{2.26}$$

With P_i: input signal power, P_L: load power and P_r: power not absorbed by the load and reflected back into the transmission line. Inserting Eq. 2.24ff. into Eq. 2.23:

$$\frac{u_i^2}{R_O} = \frac{u_i^2}{R_L} + \frac{u_r^2}{R_O} \tag{2.27}$$

$$\Leftrightarrow \frac{u_r}{u_i} = \frac{R_L - R_O}{R_L + R_O} = \rho_{\mathrm{vL}} \tag{2.28}$$

With ρ_{vL} as the voltage reflection coefficient.
Eq. 2.28 shows that the voltage reflection coefficient equals zero if $R_L = R_O$. A value of 1 indicates a complete reflection of the wave, while a value smaller than 0 indicates a mismatch between the load and output impedance.

2.2.2 Single ended and differential transmission

Single ended transmission A signal transmission over a single line is called single ended transmission. An example is shown in Fig. 2.15.

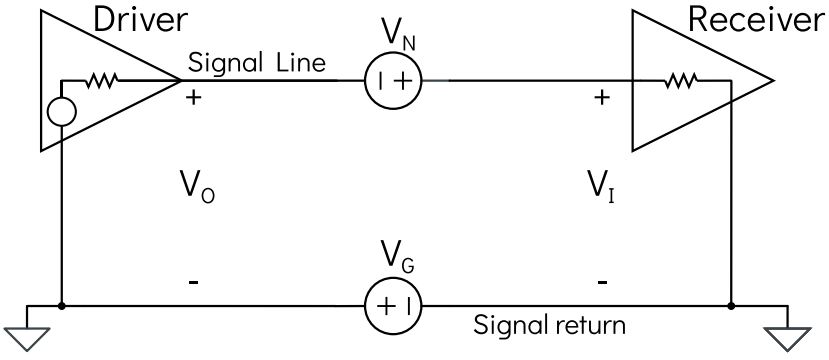


Figure 2.15: Exemplary single ended transmission, Figure from [21]

A sender and receiver share a common ground path. The input of the receiver is interpreted with respect to the ground level. The receiver input is calculated as follows [21]:

$$V_I = V_O + V_N + V_G \tag{2.29}$$

With V_I: input voltage at the receiver, V_O: output voltage at the driver, V_N: noise source for high frequency electromagnetic coupling and V_G: noise introduced by changes in the ground current. As it can be seen in Eq. 2.29, the transmitted signal is susceptible to noise sources in the transmission line.
The advantages of a single ended transmission are its simplicity and low cost of implementation. The disadvantages are a poor noise performance at high signaling rates or long distances. This is mainly because the noise coupled into the circuit adds to the signal voltage.

Differential transmission A differential signal transmission describes a transmission with complementary outputs. An example is illustrated in Fig. 2.16.

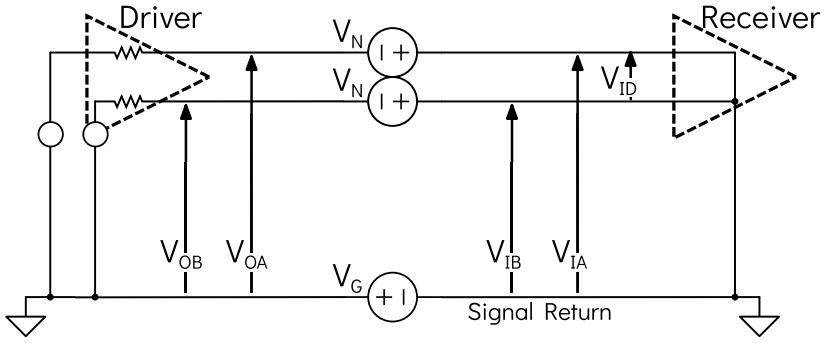


Figure 2.16: Exemplary differential transmission, Figure from [21]

The difference to the single ended transmission is that the receiver detects the voltage difference between the complementary input signals. The input at the receiver is defined as follows [21]:

$$V_{\mathrm{IA}} = V_{\mathrm{OA}} + V_{\mathrm{N}} + V_{\mathrm{G}} \tag{2.30}$$

$$V_{\mathrm{IB}} = V_{\mathrm{OB}} + V_{\mathrm{N}} + V_{\mathrm{G}} \tag{2.31}$$

$$\begin{aligned} V_{\mathrm{ID}} &= V_{\mathrm{IA}} - V_{\mathrm{IB}} \\ &= V_{\mathrm{OA}} + V_{\mathrm{N}} + V_{\mathrm{G}} - (V_{\mathrm{OB}} + V_{\mathrm{N}} + V_{\mathrm{G}}) \\ &= V_{\mathrm{OA}} - V_{\mathrm{OB}} \end{aligned} \tag{2.32}$$

As it can be seen in Eq. 2.32, the common noise terms are canceled out due to the difference detection between the two signals rather than detecting a difference to a common reference point. It can be assumed that the differential signal pairs are exposed to the same noise sources, since they usually are close together.
Twisting signal pairs together ensures similar exposure to the noise sources and cancels out the Electromagnetic field (EMF) from magnetic field coupling by reversing the polarity in adjacent loops created by the twist.
Furthermore, differential circuits radiate substantially less noise than single-ended circuits. The complementary currents in each transmission line cancel out the field generated by the other transmission line.
A disadvantage of the differential signaling is the increased cost and complexity in the circuitry as well as in the silicon.

Low Voltage Differential Signaling LVDS is a standard that specifies electrical characteristics of a differential signal transmission. It is optimized for low power consumption and moderate transmission speeds.
A driver provides 350 mV nominal into a 100 Ω load [22]. This results in a power consumption (from Eq. 2.24) of $P_i = \frac{u_i^2}{R_O} = \frac{350\,\mathrm{mV}^2}{100\,\Omega} = 1.225\,\mathrm{mW}$. The transmissions are specified for data transfers up to 1 Gbit/s for a length up to 10 m. For higher data transmission rates, principles like Emitter-coupled Logic (ECL) or Positive Emitter-coupled Logic (PECL) are suitable but consume considerably more power.

2.2.3 Timing limitations

The PDM output of the microphones is read by the processing unit on the rising edge of the clock. This requires the microphone output to reach a stable point (either a digital zero or a digital one) before the rising edge of the clock and to hold this value for a certain time until the next value of the output is set on the data line which introduces two phases:
The setup phase t_{su} describes the time before the rising edge of the clock signal where the data is already valid. The hold phase t_h describes the time after the rising edge of the clock, where the data is still valid. Outside these two phases, there is no guarantee that the data is valid. The principle is depicted in Fig. 2.17.

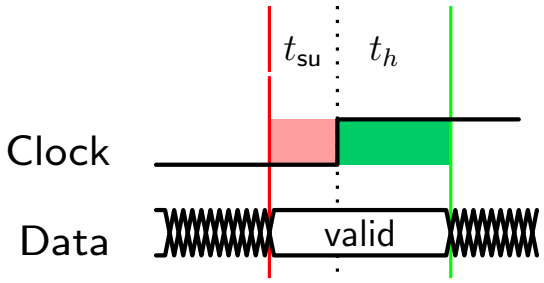


Figure 2.17: Setup and hold timing [23]

2.3 Review of available processing units

The processing units range from small scale commercially available processors to Field-Programmable Gate Array (FPGA)s. As a communication protocol to an external device for post-processing, USB was favored. USB is a widely used and well documented protocol.

2.3.1 Commercially available products.

A commercially available product is the MCHstreamer from miniDSP [24]. It already contains a microcontroller which performs preprocessing. The MCHstreamer provides connections for up to 16 MEMS microphones. There is no possibility to synchronize a MCHstreamer to an external clock source. This makes it impossible to cascade multiple MCHstreamer boards in order to increase the available number of microphones.

2.3.2 Field-Programmable Gate Array

A Field-Programmable Gate Array (FPGA) consists of logical blocks, which can be interconnected by programmable connections. This provides the maximum degree of freedom when programming the processor. However, the high degree of freedom can also become quite complex and time consuming very quickly.

Lattice Semiconductor Lattice provides a reference implementation for a PDM to PCM decimation for eight microphones [25]. Unfortunately, there is no reference implementation for a communication protocol like USB. A complete implementation of the USB stack is beyond the scope of this work and therefore not feasible.

opencores.org Opencores.org provides IP cores (reusable, already configured logic cells in a FPGA) in the typical hardware description languages Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog. The hardware description languages make the IP cores platform independent, they can be used with any compiler that can generate the bit file for the target. For the PDM to PCM decimation, a CIC filter core is available from [26]. However, the IP core was published in 2011 and last updated in 2014.
The website also provides an IP core for USB communications [27], which was last updated in 2019.

It is questionable whether the IP cores can be used directly without extensive changes in the source code. The long time since the release of the PDM IP core and the specific application for USB audio leads to the conclusion that it is too complex to use an FPGA as processing unit.

2.3.3 Microcontrollers

There is a large number of microcontrollers available on the market. Specific microcontrollers have been examined in more detail. The main criterion was on the availability of reference implementations for the data acquisition of MEMS microphones as well as a protocol for the communication with a host computer. A suitable reference implementation can serve as a good starting point and help to accelerate the development of the system.

ST Microelectronics ST Microelectronics provides a reference design for two microphones which utilizes the STM32F4 microcontroller [28]. The bit depth of the PCM output is fixed to 16 bit and at first glance, the generation of the sampling rate seems to be rather inaccurate (e.g. rounded sampling rates from 44.1 kHz to 44 kHz). An implementation of larger systems is not described and therefore, the feasibility is questionable.

Cypress Cypress provides a reference design for two microphones which utilizes their PSoC6 microcontroller [29]. In principle, the same problems occur here as in the implementation of ST microelectronics. There is a fixed bit depth of 16 bit and a rather limited sampling frequency generation. As with ST Microelectronics, it is not clear what issues might occur when implementing a larger number of microphones.

Analog Devices Analog Devices offers a large variety of Super Harvard Architecture Single-Chip Computer (SHARC) processors that can be used to process the digital microphone signal to a suitable signal for post-processing. An exemplary model is the ADSP-21489 [30]. This processor is used in the UMA-16 USB Mic Array from miniDSP [7]. Unfortunately, there is no corresponding reference implementation available and the miniDSP product uses proprietary code.

XMOS XMOS provides a reference design for seven microphones [31].There are reports of successful implementations of a larger number of microphones (e.g. in [32, 33]). The bit depth of the PCM signal can be set to either 16 bit or 24 bit and the generation of sample frequencies up to 48 kHz are supported.

2.3.4 Review results

A microcontroller from XMOS was selected as the processing unit. XMOS manufactures microcontrollers that are specialized in audio processing [34]. These microcontrollers are multicore controllers optimized for parallel processing. The designed system utilizes USB to transmit audio data to a host for post-processing. The selected device is the XUF216 [35] which provides an integrated USB Physical Layer (PHY) and a reference design for it.

2.4 xCore architecture

This section describes the most important features of the xCORE architecture which is utilized in all current XMOS products. Presented details are based on [36]. The xCORE architecture contains elements of a Real-Time Operating System (RTOS), e.g. a task scheduler, timers, channel communications and separate logical cores for real-time processing.
All xCore devices consist of one or more tiles which contain several logical cores and hardware resources like a reference clock. Each tile contains 64 KB of memory without cache as well as access to an I/O subsystem. The memory has no data bus contention since all peripherals are implemented with the I/O subsystem. There is no Direct Memory Access (DMA) for peripherals. The xCore architecture is illustrated in Fig. 2.18.

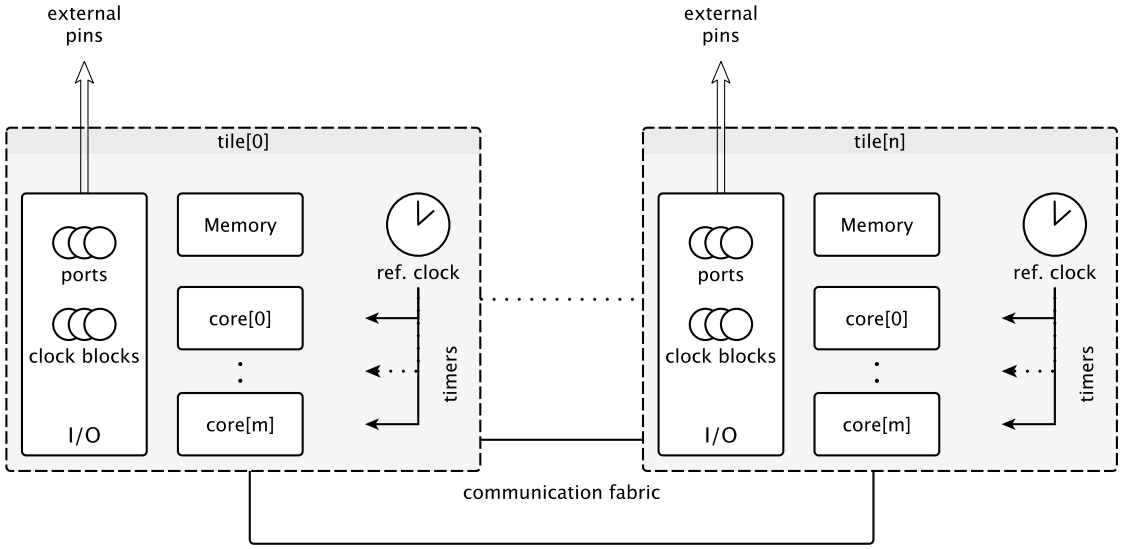


Figure 2.18: xCore hardware architecture, Figure from [36]

2.4.1 Cores

Cores are the units which execute code that result in tasks. Usually, multiple cores are combined in one tile. All cores in a tile run in parallel. Depending on the device configuration, cores can run in different speeds with a guaranteed minimum Million Instructions per Second (MIPS). Each task manages its own state and resources and can interact with other tasks by performing transactions.

2.4.2 Timers

Each tile comes with a reference clock running at a specified rate. The default rate is 100 MHz. Each reference clock is associated with a 32 bit wide counter. The counters on each tile are usually synchronized but not necessarily at the same value. With the 32 bit range of the counter it is possible to measure $2^{32} - 1$ ticks, which is approximately 42 s at 100 MHz.

2.4.3 Communication

The communication between cores is provided by the communication fabric. This allows any core on any tile to access any other core on any other tile. Each software task can perform transactions with other tasks, whether it is on the same core (or

tile) or not.
Communication between tasks is realized with explicit transactions. All communication is point to point and can be either realized with an interface or a channel.

Synchronous communication

Assume two tasks that run in parallel. Each task runs independently on its own data. At some point communication between the two tasks is necessary. The procedure is as follows:

1. Task 1 initiates a transaction with task 2
2. Task 1 waits until task 2 is ready to accept the transaction
3. Both tasks exchange relevant data
4. As soon as the data exchange is finished each task carries on

Since task 1 has to wait until task 2 responds to the transaction request, synchronous communication is called blocking communication.

Asynchronous communication

In synchronous communication, the initiating task needs to wait for the requested task to answer the transaction request. This blocks further execution in the initiating task until it gets a response.
Sometimes it makes sense that the triggering task is not dependent on the requested task. Independence from the requested task is considered to be asynchronous and therefore can be called non-blocking communication. Asynchronous communication can by realized by the following approaches:

1. Notifications
   This allows a task to raise a flag which indicates a wish to communicate with an other task. After the flag is raised, the initiating task can continue until the request is answered. This can be compared to a hardware interrupt.
2. Insert third task
   A third task acts solely as a buffering task. It is inserted in the communication chain. Since the only job of this task is to buffer data it can be very responsive. The communication is realized with push and pull transactions to the buffer task. This can be compared to a shared memory First in, First out (FIFO) between two tasks.

Interfaces

An interface provides a way of communication between cores. It defines the transaction type and what kind of data is exchanged between tasks. The interface connection is made of three parts:

- A definition of the connection via *interface T*
- A client type via *client interface T*
  The client initiates the communication
- A server type via *server interface T*
  The server responds to the communication request from the client and can send a notification to the client.

Only the client can initiate the communication. However, the server can notify the client by setting a flag. The notification is asynchronous and non-blocking. Notifications do not have arguments and their return value is always void. This means that the client has to initiate a communication after receiving a notification.

Channels

Channels provide a primitive method of communication between tasks. Communication through a channel is blocking, but there is no prior definition of the transaction type.
A non-blocking channel is realized with a *streaming channel*. The data is exchanged over a streaming channel without synchronization between the tasks.

2.4.4 Event based reactions

It is possible for tasks to react to events. This is realized with a *select()* construct which pauses the task and waits for one or more events to occur. As soon as a defined event in the select statement occurs, the code in the relevant case is executed.
The select statement can contain multiple events but does only handle one event at a time. Select statements can be used as a substitute for interrupt routines.
The cases in a select statement are normally not prioritized. This means if two events happen at the exact same time it is undetermined which select case is executed. By adding the *[[ordered]]* attribute to the select construct, the cases are prioritized from the first (highest priority) to the last (least priority). Furthermore, it is possible to add a default case to a select statement. The default case is executed if no other statement is fulfilled.

2.4.5 Shared memory access

Tasks never share data by accessing common data types (e.g. global variables) which means that the usual synchronization methods (e.g. locks, mutex or semaphores) are not used. Common data is shared by making a request to an intermediate task that owns the data.

2.4.6 I/O

Each device has up to 64 physical digital outputs distributed across the tiles of the device. The pins are referred to as XnDpq, where n describes the tile number and pq the pin number.

Ports

Pins are accessed over the hardware response ports. A port is responsible for driving output data on the pins or sampling input data. There are 29 ports available that have different port widths. The distribution is as follows: 16x 1 bit, 6x 4 bit, 4x 8 bit, 2x 16 bit and 1x 32 bit.
A port inputs or outputs all assigned bits at once. Therefore, ports should be used for I/O that work logically together. The pin to port mapping is fixed and can partially overlap. In general, overlapping ports should not be used together.

Clock blocks

All ports are clocked and are therefore attached to a clock block in the device. This controls the read and write operations of the port.
Each port has a shift register which holds either output or input data, depending on the current port definition. At each clock tick, the port samples the input pins into the shift register or drives the output pins based on the contents of the shift register.
A tile provides six clock blocks to which any port can connect. Every port can be set to either "divide" or "externally driven".
In the "divide" mode, the port operates at an integer ratio of the reference clock (which is either 100 MHz or 500 MHz). In the "externally driven" mode, the port is synchronized to an external clock. All ports are connected to clock block 0 by default. This clock block provides the reference clock which always runs at 100 MHz.

Writing and reading data to a port

The output of data on a port can be considered as a write operation. Inputting data on a port as a read operation. In both cases, a port declaration is necessary. For a write operation the appropriate data must be supplied, for a read operation the appropriate target.
A port can trigger an event when it is equal or not equal to a defined value. Selecting on conditional inputs is more power efficient than polling a port because it allows the processor to idle.

Buffered ports

Using a buffered port can improve the performance of programs that perform input or output on clocked ports. A buffer can hold the data output by the processor until the next falling edge of the assigned clock. This allows the processor to execute other instructions during this time. A single thread can perform input and output operations on multiple ports in parallel when using buffered ports.

2.4.7 Parallelism

The xC programming language is an extension to the well-known C/C++ programming language. It provides an integration of parallelism and task management. Details on xC are presented in Sec. 2.4.9. It is up to the programmer to specify on which tile and core the tasks run in parallel. If no specifications are made, the compiler will automatically place the task on a free core.

2.4.8 Threads

In xC, three different tasks are distinguished from each other: normal, combinable and distributable. The differentiation between these three types help to optimize resource usage of the device.

Normal tasks

Normal tasks run on a logical core and are independent from other tasks. They have a predictable running time and respond efficiently to external events. Unless otherwise specified, a task is considered “normal” by default.

Combinable tasks

Combinable tasks are used to have several tasks running on the same logical core. The core swaps the tasks context based on cooperative multi-tasking. A combinable task has the following requirements:

- Void return type
- The last statement of the function must be a while(1) statement which contains a single select statement
- No channel communication between combined tasks. Interfaces are allowed.

Distributable tasks

A task can contain a certain state and provides services to other tasks that do not need to react to external events. In this case, the task does not need an own core. It can share a core with the task(s) it communicates with. A distributable task has the following requirements

- It satisfies the requirements for a combinable task (while(1) loop and a single select statement)
- The cases in the select statement only respond to interface transactions

When a distributed task is connected to several other tasks, they cannot safely change their state at the same time. In this case, the compiler uses a lock to protect the state of the task.

2.4.9 Integrated Development Environment

XMOS provides an Integrated Development Environment (IDE) that contains fully standard compliant C/C++ compilation and their own xC extension. For the programming concepts in C and C++, please refer to [37, 38].
The xC extension provides task based parallelism and communication, accurate timing and access to the I/Os. All applications can be written in a mixture of xC and C/C++. It is possible to call a C/C++ function from xC and vice versa. The IDE used for this thesis is the xTimeComposer 14.4.1 on OS X 10.11.
It comes with an Eclipse[39] based Graphical User Interface (GUI) and provides [40]:

- Low Level Virtual Machine (LLVM) C, C++ and xC compilers

- xDEBUG: GDB multicore debugger
- xSIM: Cycle accurate simulator
- xSCOPE: In-circuit instrumentation and real-time logic analyzer
- XTA: Static timing analysis
- Platform support for Windows, Linux and OS X

2.4.9.1 References

The xC programming language introduces a data type called "reference" to indirectly refer to some data. The notation is not to be confused with pointers which are notated differently than in C/C++. The difference between pointers and references is the fact that pointers can be reassigned, while references cannot. Also, a reference must be assigned at initialization. A reference is indicated by an ampersand (&).

2.4.9.2 Pointers

Four different types of pointers are introduced which extend the pointer concept of C/C++: alias, restricted, movable and unsafe. A pointer is declared as follows:

```
1 pointer-type * pointer-kind variable-name
```

If no kind is specified the compiler assumes the default kind which depends on the declaration location. Details are shown in Tab. 2.2 and are described in the following.

Declaration location	**Default pointer kind**
Global variable	Restricted
Parameter	Restricted
Local variable	Alias
Function return	No default, must be explicitly declared

Table 2.2: Default kind for different pointer types

Alias

Pointers are called alias pointers when two program elements refer to the same region of memory. Alias pointers have the following restrictions:

- It is not possible to pass alias pointers to different tasks that run in parallel.
- It is not possible to access an alias pointer indirectly (e.g. a pointer to an alias pointer)

It is possible to pass an alias pointer to a function that takes a restricted pointer as an argument if the passed pointer does not alias any of the other arguments.

Restricted

A restricted pointer is a pointer that can not alias. The restricted pointer-kind allows to access the specific memory location only through that pointer. The non-alias property is checked by the xC compiler. A restricted pointer can be passed to a function which takes an alias pointer as an argument. Since restricted pointers can not be copied, they can not be used as return types for functions.

Movable

A movable pointer is a pointer of which the ownership can be changed between parts of a program. This is used e.g. to

- transfer ownership to a global variable, which is used at a later time
- transfer ownership between tasks running in parallel

The movable kind is only possible with non-alias pointers since alias pointers could lead to race conditions or dangling pointers.
A moving pointer must point to the same region it was initialized with when it leaves the scope. It can be transferred between tasks that run in parallel via an interface function argument. Interfaces with pointer arguments can not be used across tiles due to the separate tile memory.

Unsafe

Unsafe pointers are used for dynamic, aliasing data structures (e.g. linked lists). An unsafe pointer needs to be declared unsafe explicitly with the programmer ensuring memory safety. It comes with the following restrictions:

- Unsafe functions can only be called from unsafe regions.

- Within an unsafe region, an unsafe pointer can be cast to a safe pointer at the programmer's liability
- If an unsafe pointer is written from one task and read from an other this results in undefined behavior

2.4.9.3 Nullable types

Some resources must always have a valid value (e.g. interfaces, chanends, ports or clocks). Therefore, the nullable qualifier is introduced. This qualifier indicates that the resources has currently no value. The nullable qualifier is marked by a question mark (?). It is useful for optional parameters in function calls.

2.4.9.4 Multiple return values

It is possible in xC to define functions with multiple return values. The return statement sets all values that need to be returned in brackets.

2.4.9.5 Memory safety

In C memory is allocated with the "malloc()" function. Access is only allowed to this allocated memory areas. Attempts to access memory outside the allocated area result in undefined behavior.
In xC an additional restriction is introduced: No task is allowed to access memory that is owned by an other task. Only the owning task can write to its memory area. It is possible though to change the ownership of a memory area.

2.4.9.6 Runtime exceptions and parallel usage checks

The xC compiler can detect out of bound errors during compile time. If the boundaries are unknown during the compilation (e.g. a function call with an array as a parameter), the compiler inserts a runtime check for safe memory operation.
If the operation is not safe an exception is raised and the program halts at the error where it can be debugged. For devices that go into final production the exception handler can make the device reboot if such an error occurs.

As already described, the memory safety of an xC program is ensured by the restriction that no other task can access the memory area than the owning task. This applies to parallel read/write operations, but not to parallel read-only operations.

2.5 USB overview

This section gives an overview of the general principles of the Universal Serial Bus (USB) as well as its audio specific part based on [41, 42, 43] and [32]. USB is a serial bus system that specifies hardware and a packet based communication protocol. It is used to transfer data between a host and at least one device in both directions. The term "host" describes the communication participant that can initiate the communication. The counterpart to the host is described by the term "device", which can respond to communication requests but does not initiate communication.
The selected USB 2.0 standard differentiates between three speed grades: High-speed (operates at max. 480 Mbit/s), Full-speed (operates at max. 12 Mbit/s) and Low-speed (operates at max. 1.5 Mbit/s).
The system designed in this work operates at a sampling rate of 48 kHz with a bit depth of 24 bit. For a total number of 16 microphones per base board, the required bandwidth can be estimated to:

$$BW_{\text{in, 24bit}} = 16 \cdot 48\,\text{kHz} \cdot 24\,\text{bit} \approx 18.43\,\text{Mbit/s} \tag{2.33}$$

The value of 18.43 Mbit/s exceeds the Low-speed grade and the Full-speed grade. Therefore, the High-speed grad is used.
The maximum bandwidth of 480 Mbit/s is limited by the data overhead that does not contribute to the payload. Examples are acknowledgements or the events that signal the start and end of a packet. A maximum effective bandwidth of 458.8 Mbit/s is specified for an isochronous transmission [41] (Details on the transmission mode are presented in Sec. 2.5.5). Assuming that the available bandwidth is distributed equally among all devices by a single USB controller, up to 24 base boards can be connected. Per base board, 16 microphones can be preprocessed which results in a total of 384 microphones. This estimate does not account for the write to memory speed, which is usually slower and further reduces bandwidth.

2.5.1 Endpoints

The USB provides a communication service between a host (running a specific software for the device) and the function on the device. Each communication flow from or to the device is terminated in an endpoint. Therefore endpoints can be seen as data sinks or data sources. The USB standard specifies that every device needs to provide at least one endpoint, called "endpoint 0", which provides configuration information, the USB status and control access.

If more information needs to be transferred, additional endpoints can be implemented. In general, an endpoint is defined by the following properties:

- Bus access frequency/ latency requirement
- Bandwidth requirement
- Endpoint number
- Error handling behavior requirements
- Maximum packet size that the endpoint is capable to send or receive
- Transfer type of the endpoint
- Transfer direction.

The endpoint address is a unique combination of the device address, the endpoint number and the transfer direction. One endpoint is necessary per communication direction. This means that a bidirectional communication requires two endpoints, one that handles the data from the host to the device and the second in the reversed direction. Each interface can have up to eight endpoints for each communication direction.

2.5.2 Interfaces

An interface groups a number of endpoints together. Individual functions of the USB device can be accessed via interfaces. Examples of interfaces are:

- Input stream interface
- Output stream interface
- Audio control interface

Alternate settings

Interfaces can have different modes of operation. The interface Identifier (ID) stays the same, while the alternate settings ID changes. The host knows the alternate settings ID. An example for an alternate setting is the option to use different sampling rates in the system.

2.5.3 Configuration

Different configurations can be used for a more flexible use of the device. A configuration is a collection of interfaces. Each configuration can define its own interfaces

or can have interfaces in common. The USB host chooses the configuration based on power consumption, available bandwidth and speed requirements of the device. The hierarchy between the configurations, interfaces and endpoints is illustrated in Fig. 2.19. The information about the device and the available configurations is provided in descriptors. More information on descriptors is given in Sec. 2.5.6.
There is always only one Device Descriptor but there can be multiple Configuration Descriptors, Interface Descriptors and Endpoint Descriptors.

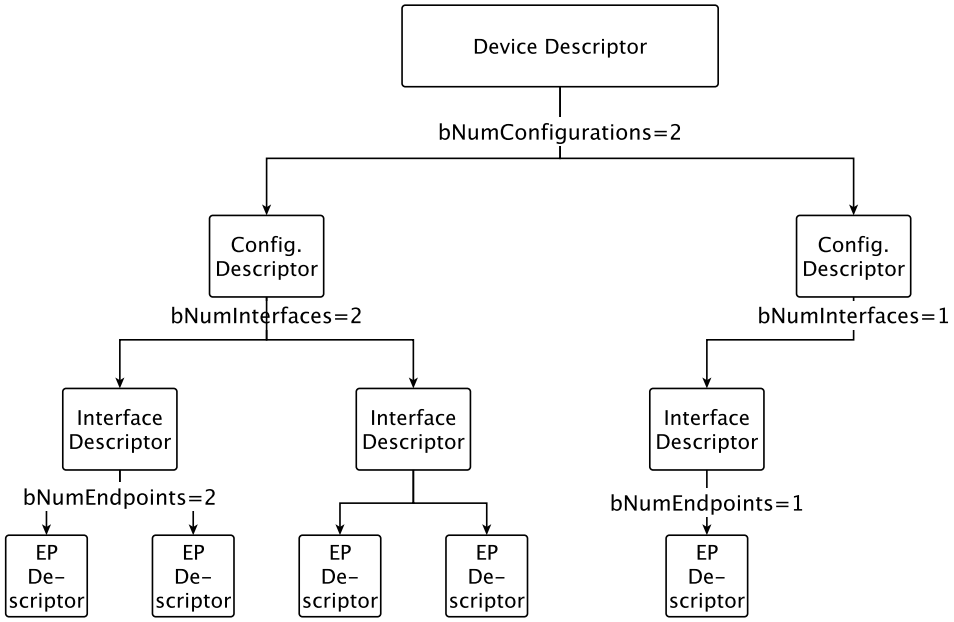


Figure 2.19: Descriptor hierarchy [41]

2.5.4 Device classes

The USB standard divides all devices into classes (e.g. human interface, printer or mass storage, etc.). All devices are required to provide information for their identification and generic configuration.
Each physical device can contain one or more device classes. A sound card could contain the Audio Device Class as well as the Human Interface Device (HID) class. If the device contains only one class, it is defined for the whole device in the Device Descriptor. If there is more than one class, each interface has its own class.
The relevant classes for this thesis are:

- Human Interface Device (HID)
  Everything a human can interact with related to a computer.

- Audio Class
  Specially for audio input and output, as well as all devices that control audio directly or indirectly (e.g. mixers, volume control).

2.5.5 Communication

The USB communication is a packet based. Data is transmitted in little endian, meaning the Least Significant Bit (LSB) is transmitted first. All transactions are initiated by the host, therefore the direction (e.g. input and output) refers to the point of view of the host. The host controls the transactions on the bus in order to avoid collisions.
The data is transmitted over a differential pair of wires. The transmission of the clock is encoded into the differential data with the Non-return-to-zero, inverted (NRZI) scheme, if necessary with adequate bit stuffing. To avoid confusion, the NRZI scheme is not part of the following descriptions.
Transmission errors can be detected by applying a Cyclic Redundancy Check (CRC) which calculates a checksum. This checksum can be used to identify erroneous data. In the best case transmission errors can be corrected with it.
As a further mechanism for minimizing transmission errors, bit stuffing is used. Here a zero is inserted after six consecutive ones to force a transition in the NRZI data stream. After seven bits at the latest, there is a transition that guarantees data and clock locks.

For the descriptions of data widths the binary and the hexadecimal notation are used in the following. The notation is identified in each case by a corresponding prefix: 0b<zeros or ones> for binary numbers and 0x<0 to F> for hexadecimal numbers. Decimal numbers are written without a prefix.

2.5.5.1 Packet types

Packets can be considered the smallest unit in the communication. The beginning and the end of each packet are signaled with specific events. This allows for a variable packet size, which is initially unknown to the receiver. Each packet is preceded by a SYNC field, allowing the receiver to synchronize their bit recovery clock. There are four different packet types with specific subtypes. A description of the packet types can be found in Tab. 2.3.

Type	Subtype	PID	Description
Token	SOF	0xA5	Start of frame marker
	SETUP	0x2D	Initiates a control transfer
	OUT	0xE1	Initiates host to device data transfers
	IN	0x69	Initiates device to host data transfers
Data	DATA0	0xC3	Data packet for toggled transfers, even
	DATA1	0x4B	Data packet for toggled transfers, odd
	DATA2	0x87	Data packet for isochronous transactions
	MDATA	0x0F	Data packet for split and isochronous transactions
Handshake	ACK	0xD2	Receiver accepts error free data packet
	NAK	0x5A	Receiver can not accept data/ transmitter can not send
	STALL	0x1E	Endpoint is halted/ control request is not supported
	NYET	0x96	Receiver buffer full, unable to receive more data
Special	PRE	0x3C	(Token) Host issued preamble. Enables downstream traffic bus to Low-speed devices
	SPLIT	0x78	(Token) High-speed Split Transaction Token
	ERR	0x3C	(Handshake) Split transaction Error Handshake (reuse of PRE value)
	PING	0xB4	(Token) High-speed flow control probe for a bulk/control endpoint

Table 2.3: USB packet types

Each packet can be identified by its Packet Identifier (PID) field. This field is eight bits long and can be divided into two portions: the normal portion and the inverted portion. The PID field is depicted in Fig. 2.20.

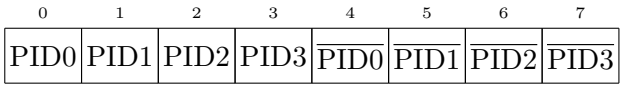


Figure 2.20: PID field, Figure from [41]

Token Packets

The specification distinguishes between Start of Frame (SOF) packets and all other token packets listed in Tab. 2.3. The structure of the IN, OUT and SETUP packets is identical and is shown in Fig. 2.22.
Token packets have a five bit Cyclic Redundancy Check (CRC) that covers the address and endpoint field in case of a IN, OUT and SETUP packet. The CRC of

the SOF packet covers the Frame Number field. The PID field has its own check field.

Start of Frame (SOF) This packet is sent at each start of a microframe. In lower speed grades, a SOF is sent every 1 ms. In High-speed mode, a microframe starts each 125 µs ± 0.0625 µs. The receiver generates no Acknowledgement (ACK) packet for a SOF packet. Therefore, the delivery of a SOF packet can not be guaranteed. The frame number is calculated in a binary manner. It is increased every millisecond to be compliant to the lower speed grades. Due to the binary calculation of the counter, an overflow sets the frame number back to zero. For the given time span of 1 ms this results in the same frame number for eight consecutive SOF packets. The SOF structure is depicted in Fig. 2.21.

Figure 2.21: Start of Frame (SOF) packet, Figure from [41]

SETUP A control transfer is initiated by the setup packet. More details on the transfer types are presented in Sec. 2.5.5.2. The address and endpoint field identify the endpoint that will receive the subsequent data packet. The structure is depicted in Fig. 2.22.

OUT/IN A data transfer is initiated by the OUT/IN packets either from or to the host. This packet is followed by a data packet from either the host or the device. The address and endpoint field identify the endpoint that either receives (OUT) or sends (IN) the data packet. The structure is depicted in Fig. 2.22.

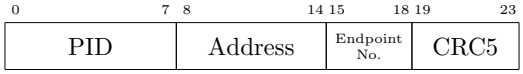


Figure 2.22: IN, OUT, SETUP packet, Figure from [41]

DATA Packets

After the communication is initiated by either a SETUP, OUT or IN packet, the DATA packet is sent. The content of the packet depends on the transfer type and application type. The 16 bit CRC covers the data field. The structure of a DATA packet is depicted in Fig. 2.23.

PID	Data	CRC16
8 bit	0 to 8192 bit	16 bit

Figure 2.23: Data packet structure, Figure from [41]

There are four different data packets PIDs: DATA0, DATA1, DATA2 and MDATA. DATA0 and DATA1 PIDs are used for toggled data transmission. In a normal transmission, each packet needs to be acknowledged by the receiver. After the sender receives the acknowledgement, the next packet is sent. If an acknowledgement is lost, the sender retransmits the previous packet. This avoids transmission errors. The behavior is shown in Fig. 2.24.

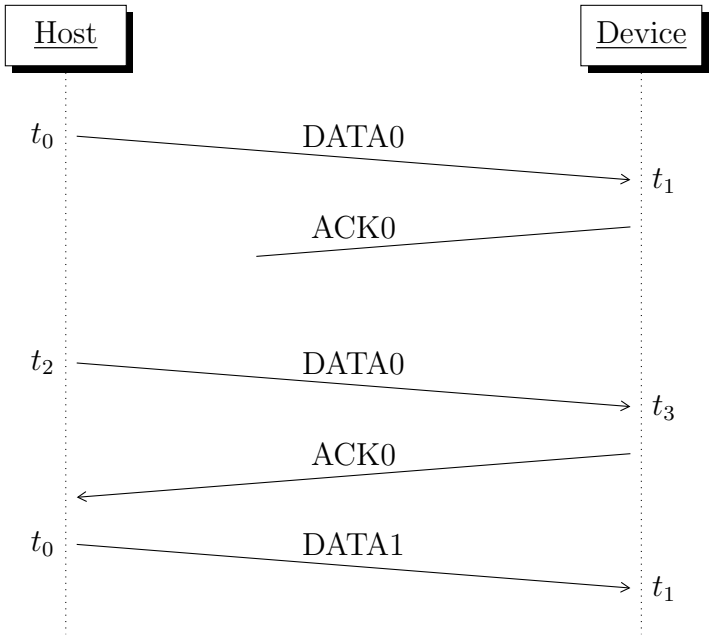


Figure 2.24: Retransmission after lost ACK

Up to three isochronous or interrupt transfers per microframe are possible for a

High-speed transmission (Sec. 2.5.5.2 describes the transfer types). The PID order is defined by the size of the data and the data direction. If the data size is larger than 1024 byte, the transfer is split into multiple transactions. The exact data size is class or vendor specific. The data sizes quoted in this document are suggestions from the specification. Since this thesis utilizes the isochronous transfer type, only the PID ordering for this mode is shown. For isochronous IN transmissions, the PIDs are:

- 1 transaction (< 1024 byte): DATA0
- 2 transactions (513 to 1024 byte): DATA1, DATA0
- 3 transactions (683 to 1024 byte): DATA2, DATA1, DATA0

For isochronous OUT transmissions, the PID sequence is:

- 1 transaction (< 1024 byte): DATA0
- 2 transactions (513 to 1024 byte): MDATA, DATA0
- 3 transactions (683 to 1024 byte): MDATA, MDATA, DATA2

If an erroneous OUT transaction is detected, all data transferred during that microframe is discarded. The identical PIDs for three transactions are intended. It is up to the vendor to implement a method to detect which MDATA packet has been lost.

Handshake Packets

Handshake packets are used for feedback on the previous transmission. A handshake packet can be used to signal successful data reception, command acceptance/rejection, flow control and holding conditions. The packet consists only of the PID field and is shown in Fig. 2.25.

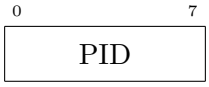


Figure 2.25: Handshake packet, Figure from [41]

ACK Indicates that the previous packet was received correctly without bit stuffing or CRC errors. An ACK packet is transmitted by the host for all IN transactions and by the device for OUT, SETUP and PING transactions.

NAK Indicates a communication error with the previously sent packet. For an OUT packet this means that the device was unable to receive the packet, for an IN packet that the device has no data to transmit. A NAK is only issued by a device and never by the host.

STALL Reports that the device is currently unable to receive or send packets. Exception: STALL after a SETUP packet indicates that the device does not support the host request.

NYET Used for bulk out transfers. Indicates that the buffer of the device is full and it can not receive any more packets. The host has to poll the device with ping packets.

Special packets

PING Packets A PING packet is used to poll whether the device has enough space in the internal buffer to receive bulk messages. The device answers with ACK if it can receive more data or with NAK if not. The structure of a PING packet is identical to the Token structure which is depicted in Fig. 2.22.

PRE, SPLIT and ERR Packets These packets are used to enable lower speed communication on a higher speed bus over a hub. They are not relevant for this work and are therefore not explained in detail here.

2.5.5.2 Transfer types

USB 2.0 provides four different transfer types that are optimized for specific purposes: Control Transfer, Isochronous Transfer, Interrupt Transfer and Bulk Transfer. A transfer type is determined by the following characteristics:

- Data format imposed by the USB
- Direction of communication flow
- Packet size constraints
- Bus access constraints
- Latency constraints
- Required data sequences
- Error handling

Control Transfer

Control Transfers can be described as bursty, non-periodic communications that are initiated by the host. This transfer type is usually used for requesting USB descriptors, the device configuration or for status operations. It is the first transfer mode when a new device is added to the bus. The stages of a Control Transfer are:

- Setup stage
  The host initiates control transfer with a SETUP packet, followed by a DATA0 packet containing the request. The device acknowledges the packet. The sequence is shown in Fig. 2.26.

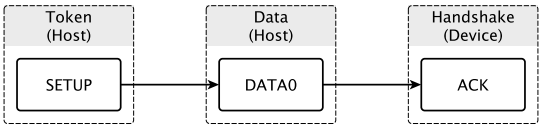
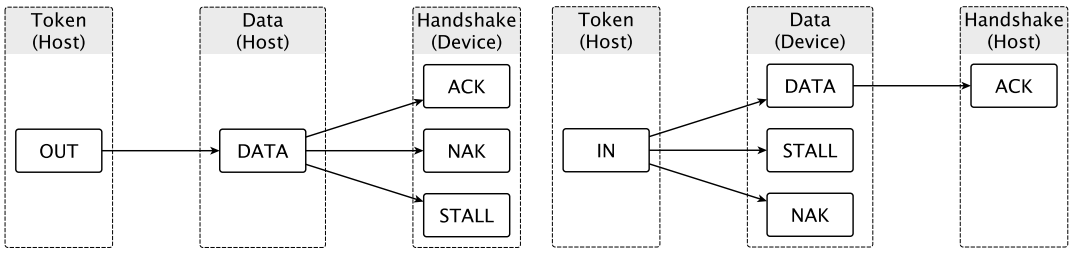


Figure 2.26: Control Transfer Setup stage, Figure from [41]

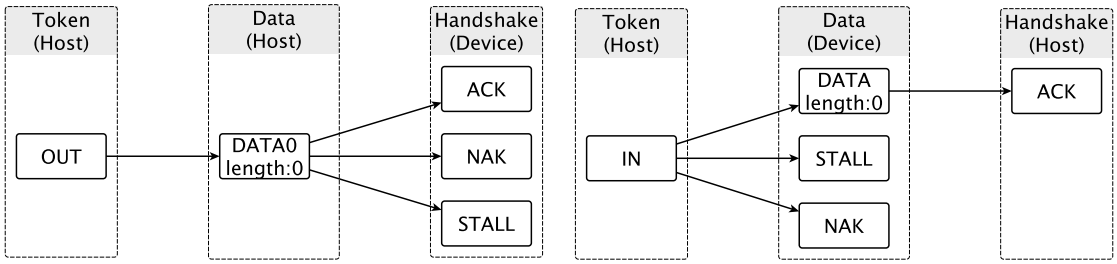
- Data stage
  Used for control transfers or requests where data must be transferred to or from the device. In the previous stage, the size of the data to be transferred is specified. Depending on the size of the data and the maximum packet size defined by the device, this stage may consist of one or more IN or OUT transmissions. The sequence is shown in Fig. 2.27.



(a) OUT (b) IN

Figure 2.27: Control Transfer Data stage, Figure from [41]

- Status stage
  The success or failure of whole Control Transfer is reported in this stage. It starts with an IN/OUT packet which is opposite to the direction from the data stage. The successful reception of the data is acknowledged by a DATA0 packet of zero length. The sequence is shown in Fig. 2.28.



(a) OUT (b) IN

Figure 2.28: Control Transfer Status stage, Figure from [41]

Isochronous Transfer

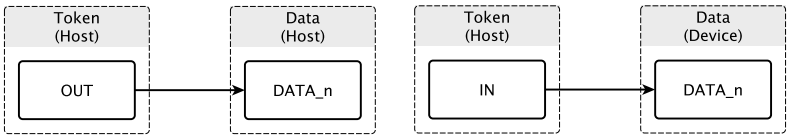
Isochronous Transfers are used for data that is streamed continuously. With this transfer type, a time slot is reserved in each frame, which guarantees a certain bandwidth. The error handling scheme for the Isochronous Transfer does not allow for retransmission of faulty or lost packets. This transfer mode should be used if data loss can be tolerated. The transmission sequence is shown in Fig. 2.29.
The required bandwidth is set in the Endpoint Descriptor. If there is not enough bandwidth left on the bus, the device is rejected at startup. If an Alternate Setting is available (for example a data reduced transmission) this fallback is used.
The data rate can be synchronized between the host and the device. Synchronization can be achieved using the following approaches:

- No synchronization
  Only used if there is no real-time application and the data rate is not of importance

- Synchronous
  The device synchronizes with the host by locking to the SOF packets from the host.

- Asynchronous
  The sender adjusts the transmission rate to prevent receiving buffer over- or underflow. Requires a feedback endpoint on the receiver side which reports on the fill level of the receiving buffer.

- Adaptive
  The device is able to synchronize itself to the data rate required by the host.

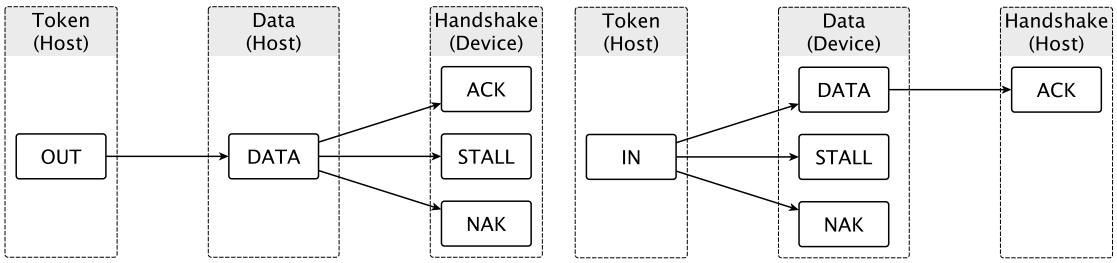


(a) OUT (b) IN

Figure 2.29: Isochronous transfer, Figure from [41]

Interrupt Transfer

Interrupt Transfers are used to receive interrupts from the device. The term interrupt should not be confused with the conventional definition of an interrupt. The host has to poll the device periodically in order to receive the interrupts. With interrupt transfers there is a guaranteed latency. If a checksum error occurs the transfer is retried in the next polling period. The polling period is defined in USB frames. The transmission scheme is shown in Fig. 2.30.

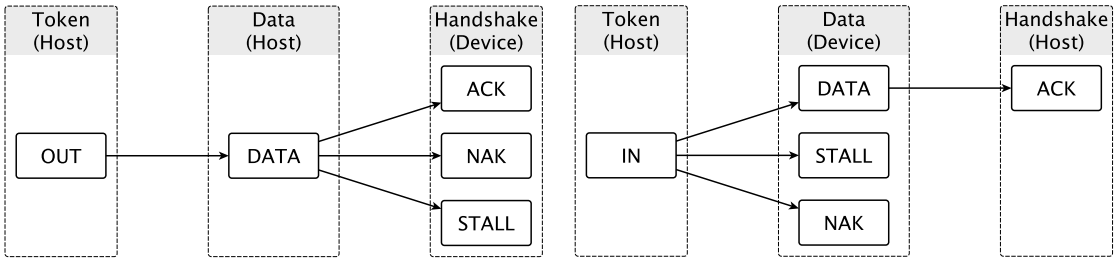


(a) OUT (b) IN

Figure 2.30: Interrupt transfer, Figure from [41]

Bulk Transfer

A Bulk Transfer is used for large data transfers that are not time-critical but often occur in bursts. Data is transferred if there are time slots available after the Isochronous and the Interrupt Transfers. Therefore, this transmission mode neither guarantees a certain latency or bandwidth. Error detection and retransmission mechanisms are provided. The transmission sequence is shown in Fig. 2.31.



(a) OUT (b) IN

Figure 2.31: Bulk transfer, Figure from [41]

2.5.5.3 Request types

Requests are used to retrieve information about the device, configuring the device or checking the status of the device. Every device has to implement standard and class specific requests. Due to identical naming of the packets, requests are not to be confused with transfers.

Setup Packet

A request starts with the setup stage and is followed by a DATA0 packet, which contains a 8 byte setup packet. The structure is described in Tab. 2.4. The *bmRequestType* field depends on the transfer direction, the type and the recipient. The values of the *wValue, wIndex* and *wLength* fields depend on the request. The layout of the field is shown in Fig. 2.32. In case the request needs data to be transferred, a Data Stage follows.

Offset	Field name	Bytes	Description
0	bmRequestType	1	Direction, Type and Recipient
1	bRequest	1	ID of the request
2	wValue	2	Request dependent value
4	wIndex	2	Request dependent value
6	wLength	2	Request dependent value

Table 2.4: Setup packet details

The *bmRequestType* field is coded as follows:

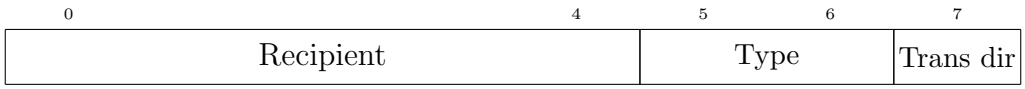


Figure 2.32: bmRequestType field for a SETUP packet, Figure from [41]

The subfields define the following:

- Recipient, 5 bit
  0: Device
  1: Interface
  2: Endpoint
  3: Other
- Type, 2 bit
  0: Standard
  1: Class
  2: Vendor
  3: Reserved
- Transfer direction, 1 bit
  0: OUT
  1: IN

Standard Request

All standard requests of the USB 2.0 protocol are shown in Tab. 2.5. The getter and setter requests are also depending on the direction and recipient bits in the bmRequestType field and therefore listed separately.

bmRequest	Recipient	wValue	wIndex	Data
Get status	Device	-	-	Status
	Interface	-	Interface	Status
	Endpoint	-	Endpoint	Status
Clear/Set Feature	Device	Feature	-	-
	Interface	Feature	Interface	-
	Endpoint	Feature	Endpoint	-
Set Address	Device	Address	-	-
Get/Set Descriptor	Device	Type or Index	- / Language	Descriptor
Get/Set Configuration	Device	-	-	Config. Value
Get/Set Interface	Interface	Alternate Setting	Interface	-
SYNCH_FRAME	Endpoint	-	Endpoint	Frame No.

Table 2.5: Standard Requests for USB 2.0

- Get status request
  The status of the device, interface or endpoint are retrieved with this request. Returns 2 byte, each bit represents a boolean status. The USB 2.0 standard defines the following recipients with the following return information:
  - Device
    Bit 0: Self Powered
    Bit 1: Remote Wakeup
  - Endpoint
    Bit 0: Halted
    Bit 1: Stalled
  - Interface
    No return information specified for interfaces.
- Clear/Set Feature
  Used to set or clear following features as a true/false value:
  - Device
    Set the remote wakeup feature
    Set the device into test mode
  - Endpoint
    Halt an endpoint
  - Interface
    The interface does not provide any features
- Set Address
  Sets address of the USB device after plugging in.
- Get/Set Descriptor
  Used in the initiation phase to request the descriptors of the device. In normal

operation, set functions are usually not used.

- Get/Set Configuration
  With the set function, a Configuration Descriptor is selected as the current configuration. The number of the currently active Configuration Descriptor is returned by the get function.

- Get/Set Interface
  Toggles between different alternative settings of a specific interface.

- SYNCH_FRAME
  Used for isochronous endpoints with varying frame sizes. The variation of the frames is performed in patterns (e.g. 2, 3, 3). The frame number of the beginning of this pattern is returned by this request.

2.5.5.4 Enumeration process

Communication begins when the host detects the connection of a device by measuring the voltage change caused by the pull-up resistors on the data lines of the device. A reset command is then sent by the host. This temporarily sets the device address to zero, which is why the host can only process one device at a time.
After the reset, the device is addressable and communication can start. At this point it acts as a basic USB device with no specific function. It can only answer to the standard control requests from and to endpoint 0.
In the next step, the host requests the Device Descriptor of the device to determine the maximum packet length (Descriptors are explained in Sec. 2.5.6). The packet length is defined by the receive buffer of the device. After this, the host resets the device again and then sends a "set address request" which assigns an unused address to the device.
As soon as the unique address is assigned to the newly connected device, the host can start the described procedure with any other newly connected device.
Once the device is assigned a unique address, the actual initialization and configuration of the device begins. The host requests all descriptors: the Device Descriptor, the Configuration Descriptors and all String Descriptors. In a last step, the host selects a configuration and sends the configuration request. The described process is called enumeration process.

2.5.6 Descriptor types

Descriptors are requested from the host as soon as a device is assigned a unique address. They are composed of a hierarchical system of sub descriptors. The hierarchy of the descriptors is shown in Fig. 2.19. In addition, they contain a list of strings that can be referenced in other descriptors. Each descriptor starts with the same two fields: *bLength* and *bDescriptorType*. The *bLength* field defines the length of the whole descriptor in bytes. The *bDescriptorType* field defines the descriptor type. The subsequent fields are descriptor dependent. The root element of the USB descriptor hierarchy is the Device Descriptor.

2.5.6.1 Device Descriptor

The device information is contained in the Device Descriptor, for example USB version, manufacturer, serial number, etc. The structure of the descriptor is shown in Tab. 2.6, the descriptor fields are explained in the following.

Offset	Field name	Bytes	Description
0	bLength	1	Descriptor length: 18 bytes
1	bDescription Type	1	0x01 (Device Descriptor ID)
2	bcdUSB	2	Version
4	bDeviceClass	1	Device Class Code
5	bDeviceSubClass	1	Subclass Code
6	bDeviceProtocol	1	Protocol Code
7	bMaxPacketSize	1	Max Packet Size of Endpoint 0
8	idVendor	2	Vendor ID (USB Org)
10	idProduct	2	Product ID
12	bcdDevice	2	Device Release Number
14	iManufacturer	1	Index of String Descriptor
15	iProduct	1	Index of String Descriptor
16	iSerialNumber	1	Index of String Descriptor
17	bNumConfigurations	1	Number of available configurations

Table 2.6: Format of the Device Descriptor

- bLength
  Descriptor length. Always 18 byte for USB 2.0

- bDescriptionType
  Set to 0x01. Identifies the descriptor type as the Device Descriptor.
- bcdUSB
  USB version in the format 0xJJMN. J: major, M: minor, N: sub minor version.
- bDeviceClass
  Specifies the class for the whole device
  - 0x00: Each interface specifies its own class code
  - 0xFF: Vendor specified class code
  - 0xFE: The interface defines the class and an Interface Association Descriptor is present
- bDeviceSubClass & bDeviceProtocol
  The correct driver for the device is selected here. Is normally only defined at the interface level.
- bMaxPacketSize
  Defines the maximum packet size in accordance with the buffer size of endpoint 0.
- iVendor & iProduct
  Uniquely identify the product. Vendor IDs are assigned by the USB Implementers Forum
- bcdDevice
  Device version number. Is in the same format as bcdUSB.
- iManufacturer, iProduct & iSerialNumber
  Index of the string descriptor for the human readable strings to describe the device
- bNumConfigurations
  Number of the available configurations of the device. This is used to request all Configuration Descriptors

2.5.6.2 String Descriptor

String Descriptors provide human-readable strings for the functions provided by the device. String Descriptors are optional. If a device does not support any String Descriptors, all String Descriptors must be set to zero.
All strings used in a descriptor are referenced by an ID and stored in a string descriptor. Using the combination of String Descriptor ID and Language ID, the host can request the desired string descriptors with a "String Descriptor request". The device then returns the corresponding string descriptor as shown in Tab. 2.8.
It is possible to provide the strings in different languages. Available language IDs can be found in the special string descriptor with ID 0. Its structure is shown in Tab. 2.7. The fields of the String Descriptor are described below.

Offset	**Field name**	**Bytes**	**Description**
0	bLength	1	String descriptor length
1	bDescriptionType	1	0x03, String Descriptor ID
2	wLANGID[0]	2	LANGID Code 0
...	...	...	...
$2 + 2 \cdot n$	wLANGID[n]	2	LANGID Code n

Table 2.7: Available Languages ID String Descriptor, addressed with ID 0

Offset	**Field name**	**Bytes**	**Description**
0	bLength	1	String descriptor length
1	bDescriptionType	1	0x03, String Descriptor ID
2	bString	N	String encoded in Unicode

Table 2.8: String Descriptor, addressed from ID 1 and above

- bLength
  Descriptor length. Depends on the string length or number of languages
- bDescriptionType
  Set to 0x03. Identifies the String Descriptor type.
- bString in Tab. 2.8
  The string encoded in Unicode. The size depends on the length of the string.
- wLANG[n] in Tab 2.7
  Language ID. E.g. English (United States): 0x0409 in little endian.

2.5.6.3 Configuration Descriptor

A Configuration Descriptor provides a configuration of the device. All the available configurations are defined in the *bNumConfigurations* field of the Device Descriptor. This number defines the valid range in which the host can request configurations. The field names are shown in Tab. 2.9 with explanations in the following.
This descriptor inherits all child descriptors in the descriptors tree. When the configuration descriptor with the length *wTotalLength* is requested, the host receives the entire descriptor tree except for the Device Descriptor and String Descriptors. An exemplary setup with different configurations is depicted in Fig. 2.33 where the *wTotalLength* parameter is represented by the gray boxes with the dashed line.

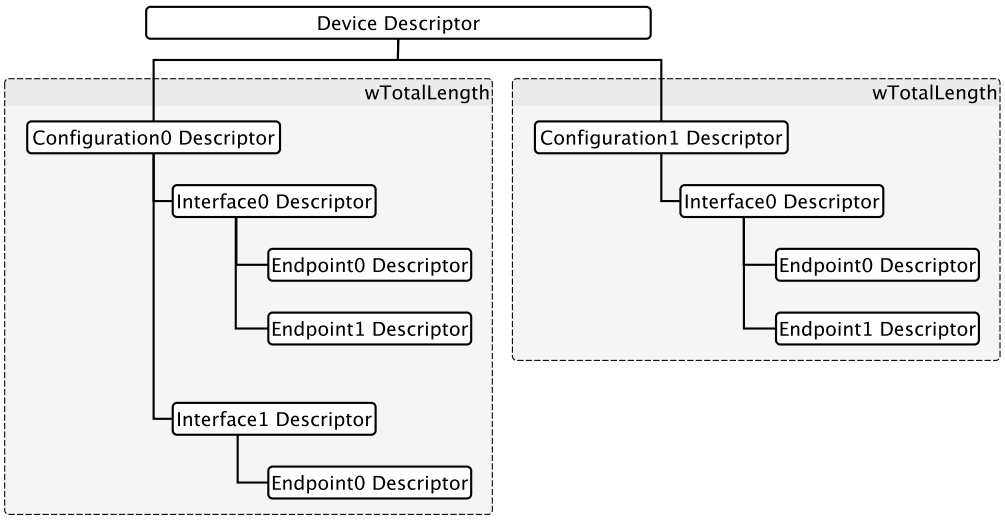


Figure 2.33: Exemplary length of two Configuration Descriptors with wTotalLength, Figure from [41]

- bLength
  Length of the Configuration Descriptor. The number of bytes in wTotalLength must be requested to retrieve all descriptors in the hierarchy.

- bDescriptionType
  Is set to 0x02 (ID of the Configuration Descriptor).

Offset	Field name	Bytes	Description
0	bLength	1	Length of descriptor
1	bDescriptionType	1	0x02: Configuration Descriptor ID
2	wTotalLength	2	Length including sub-descriptors
4	bNumInterfaces	1	Number of Interfaces
5	bConfigurationValue	1	Index of this configuration
6	iConfiguration	1	Index of String Descriptor
7	bmAttributes	1	Power settings
8	bMaxPower	1	Max. power used (x 2 mA)

Table 2.9: Configuration Descriptor format

- wTotalLength
  Length of all data including this configuration descriptor and all subsequent descriptors (Interface Descriptors, Endpoint Descriptors, Class specific descriptors, ...) which are direct children of this descriptor.
- bNumInterfaces
  Number of interfaces defined in this configuration.
- bConfigurationValue
  With this index the configuration descriptor can be queried. By transmitting this index in the "set configuration command" this particular configuration is selected.
- iConfiguration
  Index of the String Descriptor that provides a human-readable string for this configuration.
- bmAttributes
  Eight bit, each bit represents a boolean flag
  - Bit 7: Always set to 1 in USB 2.0 and higher
  - Bit 6: Self powered device
  - Bit 5: Host can wake device from a sleep state
  - Bit 4 to Bit 0: Reserved, set to 0
- bMaxPower
  Maximum power consumption of the device in this configuration. Sufficient criterion for the host to choose a configuration.

2.5.6.4 Interface Descriptor

When a Configuration Descriptor request of sufficient length is issued by the host, the Interface Descriptor is returned together with the Configuration Descriptor. One Interface Descriptor defines one of the interfaces in a configuration. In the case that this interface has alternative settings, an Interface Descriptor must exist for each alternative setting with the same interface ID. A description of the fields is shown in Tab. 2.10 and is explained below.

Offset	Field name	Bytes	Description
0	bLength	1	Descriptor length. Always 9 Byte
1	bDescriptorType	1	0x04 (Interface Descriptor ID)
2	bInterfaceNumber	1	Number of described Interface
3	bAlternateSetting	1	Number of this alternative setting
4	bNumEndpoints	1	Number of contained Endpoints
5	bInterfaceClass	1	Class Code
6	bInterfaceSubClass	1	Subclass Code
7	bInterfaceProtocol	1	Protocol Code
8	iInterface	1	Index of String Descriptor

Table 2.10: Interface Descriptor format

- bLength
  Length of the Interface Descriptor. Value is always set to 9 Byte.
- bDescriptionType
  Set to 0x04 (Interface Descriptor ID).
- bInterfaceNumber
  Number of the interface that this descriptor defines. More than one alternate setting can be assigned to an interface. Therefore multiple interface descriptors can describe the same interface for different settings. This is defined in the *bAlternateSetting* field.
- bAlternateSetting
  Number of alternate settings which this Descriptors defines.
- bNumEndpoints
  Number of endpoints this alternate setting uses.

- bInterfaceClass
  Class code for this interface.

- bInterfaceSubClass
  Subclass of the selected interface class. If the device descriptor defines the class, it should match iDeviceSubClass in the device descriptor.

- bInterfaceProtocol
  Code for the used protocol. Defined by the selected class.

2.5.6.5 Endpoint Descriptor

The Endpoint Descriptor is returned with the configuration descriptor as well. It is a child of the Interface Descriptor. Field names are shown in Tab. 2.11 and described in the following.

Offset	Field name	Bytes	Description
0	bLength	1	Descriptor length: 7 Byte
1	bDescriptorType	1	0x05 (Endpoint Descriptor ID)
2	bEndpointAddress	1	Address and data direction
3	bmAttributes	1	Transfer type (and isochronous types)
4	wMaxPacketSize	2	Maximum packet size
6	bInterval	1	Polling interval

Table 2.11: Endpoint Descriptor format

- bLength
  Describes the length of the descriptor, which is always 7 Byte.

- bDescriptorType
  Set to 0x05 (Endpoint Descriptor ID)

- bEndpointAddress
  Contains the number and the data direction of the endpoint

  - Bit 7: Direction.
    0: Out
    1: In
  - Bit 6 to bit 4: Reserved, set to 0

  - Bit 3 to bit 0 : Endpoint number

- bmAttributes
  Describes the transfer type of the endpoint: control, isochronous, interrupt or bulk. For isochronous endpoints it also includes the synchronization and usage types. If the transfer mode is not isochronous, the usage type and synchronization type should be set to 0.

  - Bit 7 to bit 6: Reserved, set to 0
  - Bit 5 to bit 4: Usage type in isochronous mode
    0: Data Endpoint
    1: Feedback Endpoint
    2: Implicit feedback Data endpoint
  - Bit 3 to bit 2: Synchronization type in isochronous mode
    0: No synchronization
    1: Asynchronous
    2: Adaptive
    3: Synchronous
  - Bit 1 to bit 0: Transfer type
    0: Control
    1: Isochronous
    2: Bulk
    3: Interrupt

- wMaxPacketSize
  Describes the maximum packet size. The value is defined by the used buffer in the endpoint.

- bInterval
  Poll interval expressed in frames. The host polls all *bInterval* frames when an interrupt is pending on the endpoint. This field is set to 0 for control and bulk endpoints, since there are no periodic polls.
  In the case of isochronous endpoints, this field is set to 1, since they can stream and adjust the amount of data for each frame to their desired data rate.
  For interrupt endpoints, this field is set to any value between 1 and 255. The interrupt is then processed every *bInterval* frames after it occurs.

2.5.7 USB Audio

The previous sections explained the standard USB components and communication principles in more detail. This section adds the most important details from the USB Audio Class specifications [42].
The audio sample transmission is realized via an asynchronous Isochronous Transfer endpoint of the class AudioStreaming (AS). Asynchronous source endpoints supply their data rate implicitly by the number of samples they deliver per microframe. The AudioControl (AC) device interface provides a number of control values which control the behavior of the device. These values are defined in the AC Descriptor. AC Requests are used to read or write these values. An exemplary structure of an USB audio device is shown in Fig. 2.34.

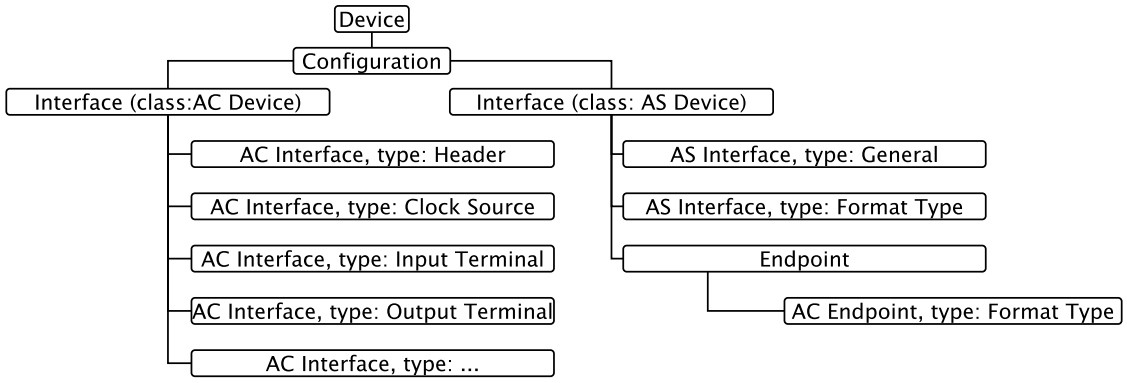


Figure 2.34: USB Audio device structure using the AudioControl (AC) and AudioStreaming (AS) class, Figure from [42]

2.5.7.1 Class-Specific Requests

The Audio Device class supports the standard requests that are described in Sec. 2.5.5.3. Furthermore, it introduces class specific requests that provide getter and setter methods to control the audio and streaming functions of the device. Examples of these methods are channel and master volumes or setting different sampling rates.

AudioControl Requests

The AudioControl (AC) Request is a class specific request. With this request, the audio device is controlled by changing certain control values. There are two request

types: Get/Set Current Value and Get Range. Get/Set Current Value returns or sets the actual control value. Get Range returns a valid range of values allowed for setting a value. All AudioControl Requests are sent over endpoint 0 in the same form as the control requests described in Tab. 2.4. The fields are utilized as follows:

- bmRequestType

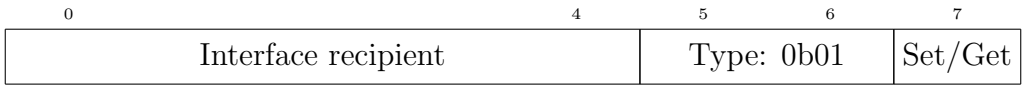


Figure 2.35: bmRequestType field for AC requests, Figure from [42]

  - Recipient, 5 bit
    1: AC interface
    2: AS interface
  - Type, 2 bit
    1: Class specific request (fixed value)
  - Transfer direction, 1 bit
    0: SET
    1: GET

- bRequest
  Is either set to 1 for a Get/Set Current Value request or set to 2 for a Get Range request.

- wValue
  Divided into a high byte and a low byte. The high byte describes the Control Selector (CS) which describes the control value and the low byte is the audio channel number which addresses the logical channel number.

- wIndex
  If it the recipient is an interface, the field is split into a high and a low byte. The high byte describes the Entity ID (e.g. Clock Entity ID), the low byte specifies the interface. If the recipient addresses an endpoint, the low byte specifies the end point and the high byte is set to zero.

- wLength
  Describes the length of the parameter block.

A control request data stage is structured as follows:

- The data sent for a "Get/Set Current Value" request is either 1, 2 or 4 byte long and therefore represents an 8, 16, or 32 bit value.

- A “Get Range” request consists of a number of subranges. The first two bytes describe the length of the subranges. Each subrange consists of three values: dMIN, dMAX and dRES where the number of bytes of each value is the same number of bytes as for the “Get/Set Current Value” request.
  This results in a total number of bytes for each subrange of 3, 6 or 12 Byte.
  dMIN describes the lowest valid value, dMAX the highest and dRES the step size between dMIN and dMAX.

2.5.7.2 AudioControl Interface Descriptors

The standard AudioControl (AC) Interface Descriptors are identical to the Control Interface Descriptors presented in Tab. 2.10. The assigned field values are explained in Tab. 2.12.

Offset	Field name	Bytes	Description
0	bLength	1	Descriptor length. Always 9 Byte
1	bDescriptorType	1	0x24 (AC Interface Descriptor ID)
2	bInterfaceNumber	1	Number of described Interface
3	bAlternateSetting	1	Number of described Alternate Setting
4	bNumEndpoints	1	1 if interrupt endpoint is present, else 0
5	bInterfaceClass	1	0x01 (Audio Interface Class)
6	bInterfaceSubClass	1	Audio Interface subclass: Undefined, Control, Audio streaming, Midi streaming
7	bInterfaceProtocol	1	0x20: IP_VERSION_02_000
8	iInterface	1	Index of String Descriptor

Table 2.12: Standard AudioControl Interface Descriptor

There are class specific Interface Descriptors which are described in the following.

Header

The total length of the class specific AC Interface Descriptor is defined by the number of Clock Entities, Units, Terminals and Power Domains. For that reason, the AC Interface Descriptor begins with a header, that contains the total length of all class specific descriptors in its *wTotalLength* field. The AudioControl Header Interface Descriptor is shown in Tab. 2.13.

Offset	Field	Size	Description
0	bLength	1	Descriptor length: 9 Byte
1	bDescriptorType	1	0x24 (Class Specific Interface)
2	bDescriptorSubtype	1	0x01 (HEADER)
3	bcdADC	2	Audio Device Class Specification
5	bCategory	1	Indicates primary use of this function
6	wTotalLength	2	Number of bytes returned for this descriptor
8	bmControls	1	Bits 0 to 1: Latency control Bits 2 to 7: Reserved, set to 0

Table 2.13: AudioControl Interface Header Descriptor

Clock Source

The Clock Source Descriptor is used to control the audio clock. The value of the frequency control sets the audio sampling rate. The clock validity control is typically read-only and returns a Boolean status. Details can be found in Tab. 2.14.
Requests for the clock selection and clock multiplier control are specified. Since the developed system uses a fixed sampling rate of 48 kHz, these requests are unused and therefore not described in more detail here.

Offset	Field	Size	Description
0	bLength	1	Descriptor length: 8 Byte
1	bDescriptorType	1	0x24 (Class Specific Interface)
2	bDescriptorSubtype	1	0x0B (Clock Source)
3	bClockID	1	Unique identifier of the Clock Source
4	bmAttributes	1	Clock configuration
5	bmControls	1	Clock Control
6	bAssocTerminal	1	Terminal ID associated with this Clock Source
7	iClockSource	2	Index of String Descriptor

Table 2.14: AudioControl Clock Source

The *bmAttributes* field is utilized as follows:

- Bit 0 to 1: Clock Type
  00: External Clock
  01: Internal fixed Clock
  10: Internal variable Clock
  11: Internal programmable Clock
- Bit 2: Clock synchronized to SOF
- Bit 3 to 7: Reserved, set to 0

The *bmControls* field is utilized as follows:

- Bit 0 to 1: Clock Frequency Control
- Bit 2 to 3: Clock Validity Control
- Bit 4 to 7: Reserved, set to 0

Input Terminal

The device developed in this thesis provides only an audio input to the host. Therefore, only the Input Terminal Descriptor (ITD) is explained here. For the Output Terminal Descriptor refer to [42]. The ITD provides information about functional aspects of the Input Terminal. Its structure is shown in Tab. 2.15.

Offset	**Field**	**Size**	**Description**
0	bLength	1	Descriptor length: 17 Byte
1	bDescriptorType	1	0x24 (Class Specific Interface)
2	bDescriptorSubtype	1	0x02 (Input Terminal)
3	bTerminalID	1	Unique Terminal ID
4	wTerminalType	2	Constant characterizing the terminal
6	bAssocTerminal	1	Output terminal ID this input is associated with. If no association set to 0
7	bCSourceID	1	Clock Entity ID this Input Terminal is connected to
8	bNrChannels	1	Number of logical output channels in the channel cluster
9	bmChannelConfig	4	Spatial location of the channels
13	iChannelNames	1	Index of String Descriptor for Channels
14	bmControls	2	Controls
16	iTerminal	1	Index of String Descriptor for Terminal

Table 2.15: Input Terminal Descriptor

The *bmControls* field is utilized as follows:

- Bits 0 to 1: Copy Protect Control
- Bits 2 to 3: Connector Control
- Bits 4 to 5: Overload Control
- Bits 6 to 7: Cluster Control
- Bits 8 to 9: Underflow Control
- Bits 10 to 11: Overflow Control
- Bits 12 to 15: Reserved, set to 0

Feature Unit

The Feature Unit (FU) Descriptor describes the parameters that are processed by the device and can be manipulated by the host. For the device designed in this thesis, these are the functions: Mute and Volume, both of which are available for each individual input and the master channel. The structure is shown in Tab. 2.16.

Offset	**Field**	**Size**	**Description**
0	bLength	1	Descriptor length: 6+(CH+1)*4 Byte
1	bDescriptorType	1	0x24 (Class Specific Interface)
2	bDescriptorSubtype	1	0x06 (Feature Unit)
3	bUnitID	1	Unique Unit ID
4	bSourceID	1	ID of the connected Unit or Terminal
5	bmaControls(0)	4	Controls for master channel 0
5+(1*4)	bmaControls(1)	4	Controls for channel 1
5+(n*4)	bmaControls(n)	4	Controls for channel n
5+(n+1)*4	iFeature	1	Index of String Descriptor

Table 2.16: Feature Unit Descriptor

Two bits each in the *bmaControls(n)* field describe the following states: 0b00: Not present, 0b01: Present; read only, 0b11: Present; host programmable, 0b10: Reserved.

- Bit 0 to 1: Mute Control
- Bit 2 to 3: Volume Control
- Bit 4 to 5: Bass Control
- Bit 6 to 7: Mid Control
- Bit 8 to 9: Treble Control
- Bit 10 to 11: GEQ Control
- Bit 12 to 13: Auto. Gain Control
- Bit 14 to 15: Delay Control
- Bit 16 to 17: Bass Boost Control
- Bit 18 to 19: Loudness Control
- Bit 20 to 21: Input Gain Control
- Bit 22 to 23: Input Gain Pad Ctrl
- Bit 24 to 25: Phase Inverter Control
- Bit 26 to 27: Underflow Control
- Bit 28 to 29: Overflow Control
- Bit 30 to 31: Reserved, set to 0

2.5.7.3 AudioStreaming Interface Descriptors

The Standard AS Interface Descriptors define the audio format sent or received. The format is identical to the Interface Descriptor in Tab. 2.10 and is explained in Tab. 2.17. The class specific AS Interface Descriptors from Tab. 2.18 are used for device specific configuration.

Offset	Field name	Bytes	Description
0	bLength	1	Descriptor length. Always 9 Byte
1	bDescriptorType	1	0x04 (Interface Descriptor ID)
2	bInterfaceNumber	1	Number of described Interface
3	bAlternateSetting	1	Select Alternate Setting
4	bNumEndpoints	1	Endpoints used by this interface 0x00: no data endpoint 0x01: data endpoint 0x02: data and explicit feedback endpoint
5	bInterfaceClass	1	0x01 (Audio Interface Class)
6	bInterfaceSubClass	1	0x02 (Audio Streaming)
7	bInterfaceProtocol	1	0x20: IP_VERSION_02_000
8	iInterface	1	Index of String Descriptor

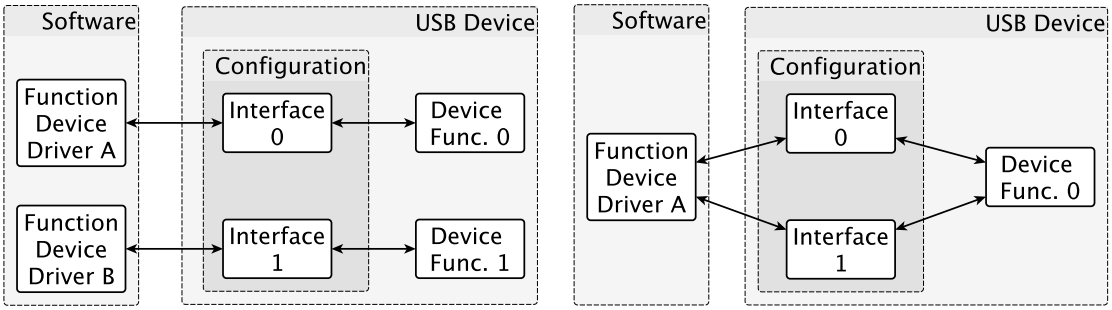
Table 2.17: Standard AudioStreaming Interface Descriptor

Offset	Field name	Bytes	Description
0	bLength	1	Descriptor length. Always 16 Byte
1	bDescriptorType	1	0x24 (AC Interface Descriptor ID)
2	bDescriptorSubtype	1	0x01 (AS General)
3	bTerminalLink	1	ID of connected Terminal
4	bmControls	1	Bit 0 to 1: Active Alternate Setting Bit 2 to 3: Valid Alternate Setting Bit 4 to 7: Reserved. Set to 0
5	bFormatType	1	ID for FormatType & used AS interface
6	bmFormats	4	Audio Formats used for communication
10	bNrChannels	1	Number of channels
11	bmChannelConfig	1	Spatial location of physical channels

Table 2.18: Class specific AudioStreaming Interface Descriptor

2.5.7.4 Interface Association Descriptor

The Interface Association Descriptor (IAD) groups interface descriptors that logically belong to a function of the device. This creates one device driver for each association of interfaces instead of one driver for each interface. The advantage is illustrated in Fig. 2.36.



(a) Without IAD (b) With IAD

Figure 2.36: Comparison of the IAD to the conventional approach, Figure from [41]

Field name	Bytes	Description
bLength	1	Length of this descriptor
bDescriptorType	1	0x0B (Interface Association Descriptor ID)
bFirstInterface	1	Index of the first interface in this group
bInterfaceCount	1	Total number of interfaces in this group
bFunctionClass	1	The class of this group
bFunctionSubClass	1	The subclass of this group
bFunctionProtocol	1	The protocol of this group
iFunction	1	Index of the string descriptor

Table 2.19: Interface Association Descriptor format

Chapter 3

Implementation

This chapter presents the hardware and firmware designed in this thesis. It is therefore divided into a hardware and a firmware part.

The system preprocesses the incoming Pulse-Density Modulation (PDM) data from the microphones into Pulse-Code Modulation (PCM) data suitable for post-processing. The sampling rate of the PCM data stream is 48 kHz with a bit depth of 24 bit.
If several base boards are used simultaneously, the audio signal must be sampled coherently. This requires a uniform sampling clock on all base boards. Inspired by applications in the pro-audio field, a clock master - clock slave system is implemented. For a small number of base boards (usually not more than two), one base board acts as clock master, all other base boards are clock slaves. If a larger number of base boards is used, they are all in clock slave mode to an external clock master. The master clock signal has the same frequency as the used sampling frequency.

3.1 Hardware design

The system is divided into two main parts: the base board and the microphones. The microphones are detached from the base boards to increase flexibility. Each base board can process up to 16 PDM signals from the microphones and send them to a host computer via the Universal Serial Bus (USB). The top level view of the designed system is depicted in Fig. 3.1.
The total current consumption of a complete 16 microphone system is estimated to 1.6 A, assuming the worst case consumptions specified in the respective data sheets. The measured values are given in brackets and are lower than the estimate by a factor of 2.6 which indicates that the components do not operate at their maximum

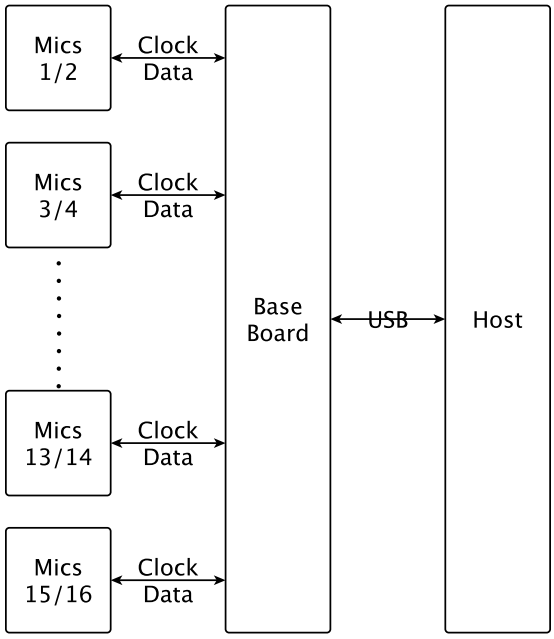


Figure 3.1: Top level view of the system

capability. The estimation is shown in Tab. 3.1.
Details of the current estimation can be found in the respective subsections.
The USB 2.0 is specified for supply currents up to 500 mA [44]. This means that the system cannot be powered by the host, but requires an external power supply.

Component	**Qty**	**Max. current in mA**	**Estimated Sum in mA**
Microphone PCB	8	52.2 (30)	417.6 (240)
Base board PCB	1	1199.5 (380)	1199.5 (380)
		Total:	**1617.1 (620)**

Table 3.1: Current estimation of the system, measured values in brackets

3.1.1 Microphone PCB

The microphone Printed Circuit Board (PCB) consists of two Micro-Electro-Mechanical Systems (MEMS) microphones, a differential to single ended receiver for the clock signal and a single ended to differential transmitter for the data signal. Furthermore,

it contains a Power Supply Unit (PSU) that converts the incoming 5 V to the required 3.3 V and a status LED that indicates the power status. The block diagram of a microphone PCB is shown in Fig. 3.2. The schematic and layout are attached in App. A.1 to App. A.3.

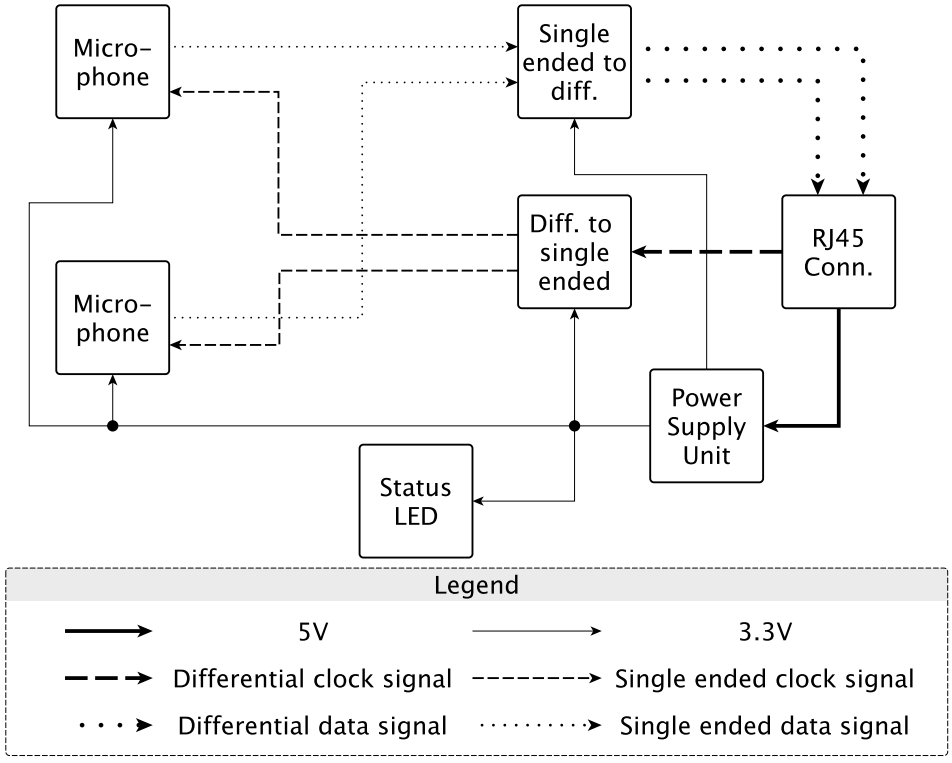


Figure 3.2: Microphone PCB block diagram

Microphone The microphone is the SPH0690LM4H-1 from Knowles [45]. It is an omnidirectional microphone with a sensitivity of −35 dBFS, a Signal-to-Noise Ratio (SNR) of 68 dB(A), an Acoustic Overload Point (AOP) of 130 dBSPL and a defined frequency range of 30 Hz to 24 kHz. They are clocked with 3.072 MHz which corresponds to a 64x oversampling of the desired sampling frequency of 48 kHz. The microphone can either output the data on the rising or falling edge of the clock signal. This allows to operate two microphones on one data line, with one microphone outputting its data on the rising edge of the clock and the other microphone outputting its data on the falling edge. Since the PDM to PCM conversion library from

XMOS only supports data changes on the falling edge of the clock, it is not possible to use one data line for two microphones.
The data output of the microphone is single ended and is converted to a differential signal by the transmitter.

Single ended to differential transmitter The single ended to differential transmitter is the MAX9112 from Maxim Integrated [46]. It converts two single ended inputs into two differential outputs. The differential output is operated in the Low Voltage Differential Signaling (LVDS) format.

Differential to single ended receiver and clock buffer The differential to single ended receiver is the MAX9111 from Maxim Integrated [47]. It converts the differential clock input into a single ended signal.
As specified for the differential transmission, the input needs to be terminated with a 100 Ω resistor at the input of the Integrated Circuit (IC).
In order not to load the receiver output with multiple connections, a 1:3 fanout buffer used. Each output of this buffer supplies a microphone with the clock. The buffer is the PL133-37TC from Micrel [48].

Power Supply Unit The input voltage on the microphone PCB is 5 V which leaves a headroom for fluctuations and losses (e.g. due to cable length). The PSU regulates the incoming 5 V to the required 3.3 V. The conversion is performed by the TPS7A26 from Texas Instruments [49]. It is a Low Drop-out (LDO) linear voltage regulator. A LDO has a low efficiency but needs less external circuitry compared to a step-down converter. It also drives a green LED which indicates that the PCB is powered.

Current estimation The estimated current consumption of the microphone PCB is shown in Tab. 3.2 and is approximately 52 mA. The measured current consumption is 30 mA and shown in brackets. The lower power consumption is due to the fact that the components do not operate at their maximum.

Component	Qty	Max. current in mA	Sum in mA
SPH0690LM4H-1	2	5	10
MAX9111EKA+T	1	6	6
PL133-37TC	1	1.2	1.2
MAX9112EKA+T	1	13	13
ON LED	1	22	22
		Total:	**52.2 (30)**

Table 3.2: Current estimation for the microphone PCB, measured value in brackets

Frequency range limitations The limitation of the frequency range is determined by two factors: spectral and spatial aliasing as well as the frequency range of the microphones which is from 30 Hz to 24 kHz.
The sampling rate defines the frequency where spectral aliasing starts, while the microphone array dimensions define an upper and lower frequency for spatial aliasing. The Nyquist theorem limits the upper frequency of the system to $f_{\max} = f_s/2 = 48\,\text{kHz}/2 = 24\,\text{kHz}$. Frequencies above the Nyquist frequency are affected by spectral aliasing.
Spatial aliasing occurs if the microphone distance is a multiple of the wavelength λ, resulting in very small amplitude differences, so that no clear distinction is possible. Assuming that the microphones are equally spaced, the distance between two adjacent microphones defines the upper frequency limit. It is calculated as follows [50]:

$$f_{\max} = \frac{c}{2 \cdot d} \tag{3.1}$$

With $c = 343\,\frac{\text{m}}{\text{s}}$ and $d = 1.7\,\text{cm}$ this results in $f_{\max} \approx 10\,\text{kHz}$. Frequencies above that are affected by spatial aliasing.
The lower frequency is defined by the Sparrow limit [51] which describes the minimum distance between two sources that still can be distinguished.
It is calculated as follows [51]:

$$r_s = 0.47\frac{\lambda}{\sin(\alpha)} \tag{3.2}$$

With $\sin(\alpha)$ describing the half angle from the position of the source to the edge of the array is described. The validation array setup is described in more detail in Sec. 4.2.1. Selected frequencies for this setup are listed in Tab. 3.3.

Frequency	Min. distance
300 Hz	6.49 m
500 Hz	3.9 m
1 kHz	1.95 m
5 kHz	0.39 m
7 kHz	0.28 m
15 kHz	0.13 m

Table 3.3: Sparrow limit of the validation array

3.1.2 Base board PCB

The base board can be divided into three main sections:

- PSU for the whole system
- Clock generation from the internal or external source, defined by the user
- PDM to PCM conversion and USB communication

The top level view of the base board is depicted in Fig. 3.3. It includes a debug header that has no visible connection to the microcontroller since it is not important for the presented signal flow.
The design is inspired by the XMOS microphone array reference implementation from [31] which operates with seven microphones. The schematics and layout are attached in App. A.4 to App. A.7.

3.1.2.1 Power Supply Unit

The PSU of the base board generates the required voltages of 3.3 V and 1.0 V and passes the incoming 5 V to the RJ45 connector for the microphone PCBs. Before distributing the incoming voltage, it is sent through a ferrite bead. This cleans the input voltage from high frequency noise resulting in a cleaner input voltage for all components. The 3.3 V and 1.0 V generation are based on the same step-down DC-DC converter: the ST1S06A from ST Microelectronics [52]. The output voltage is defined by the external voltage divider of the IC. It is calculated as follows [52]:

$$V_{\text{out}} = V_{\text{FB}} \left(1 + \frac{R_1}{R_2}\right) \tag{3.3}$$

which results in $R_1 = 18\,\text{k}\Omega$ and $R_2 = 5.6\,\text{k}\Omega$ for the generation of 3.3 V. For the generation of 1 V the resistors are calculated to $R_1 = 4.7\,\text{k}\Omega$ and $R_2 = 18\,\text{k}\Omega$.

$$V_{\text{out, 3.3V}} = 0.8\,\text{V} \left(1 + \frac{18\,\text{k}\Omega}{5.6\,\text{k}\Omega}\right) = 3.37\,\text{V} \tag{3.4}$$

$$V_{\text{out, 1.0V}} = 0.8\,\text{V} \left(1 + \frac{4.7\,\text{k}\Omega}{18\,\text{k}\Omega}\right) = 1.00\,\text{V} \tag{3.5}$$

The power supply is considered the analog part of the board. For that reason, the PSU components have their own ground plane which is called AGND. Introducing two separate ground planes minimizes interference between the analog and digital domain. AGND and GND (digital ground) are connected with a $0\,\Omega$ resistor.
All components except the XMOS XUF216 run on 3.3 V. The XUF216 has the requirement that its reset pin is kept low until the supply voltages are stable.

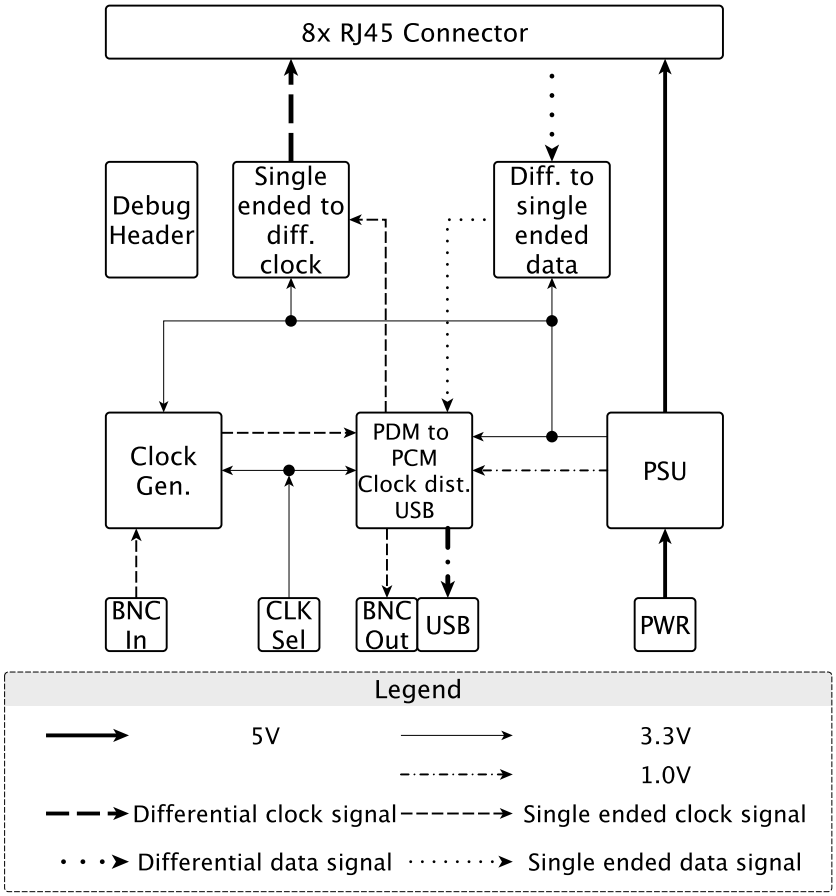


Figure 3.3: Top level view of the base board

Power on delay The XUF216 requires to be kept in reset until at least 1 ms after VDDIOL has reached at least 3.1 V. The power on delay is realized with two ICs: The STM1061 from ST Microelectronics [53] and the MAX6895 from Maxim Integrated [54].
The STM1061 is a voltage detector that monitors the input voltage and forces its output to low as long as the voltage is below 2.8 V. The output signal of this IC is connected to the enable pin of the MAX6895 and the 1 V PSU. As soon as the output voltage of the 3.3 V PSU has surpassed the 2.8 V limit of the voltage monitor, the STM1061 output enables the 1 V PSU and starts the timed delay of the MAX6895. The delay of the MAX6895 is defined by an external capacitor and calculated as

follows [54]:

$$t_{\mathrm{DELAY}} = (C_{\mathrm{CDELAY}} \cdot 4E6) + 40\,\mu\mathrm{s} \\ = (2.7\,\mathrm{nF} \cdot 4E6) + 40\,\mu\mathrm{s} = 10.84\,\mathrm{ms} \tag{3.6}$$

The voltage output of both PSUs are stable after couple of microseconds. The delay time of 10.84 ms fulfills the 1 ms requirement by far. The reset phase is over as soon as the output of the MAX6895 is set to high, allowing the XUF216 to boot. Furthermore, this signal activates the ON LED of the base board.

An estimated current consumption of the base board is shown in Tab. 3.4. The estimation is based on the maximum values specified in the respective data sheets. It is approximately 1.2 A. The measured total current consumption is 380 mA. This is about one third of the estimated current consumption and can be explained by the fact that most of the components are not working at their performance limits.

Component	**Qty**	**Max. current in mA**	**Sum in mA**
XUF216-512-TQ128-C20	1	700	700
PI6C5921512ZDIEX	1	213	213
SN74LVC2G34DBVR	1	0.5	0.5
CS2100CP-CZZ	2	18	36
74LVC1G11GW,125	1	100	100
SN65LVDT386DGG	1	70	70
LMK00804BQWRGTTQ1	1	25	25
ST1S06APUR	2	1.5	3
ASFL1-24.000MHZ-L-T	1	15	15
ASFL1-24.576MHZ-L-T	1	15	15
ON LED	1	22	22
		Total:	**1199.5 (380)**

Table 3.4: Current estimation for the base board, measured value in brackets

3.1.2.2 Reset logic

The base board provides three reset options. The first one is the already described *power on delay*, the second one is provided for the debug header and the third option is a manual reset via a button. All three reset lines are fed into an AND gate [55]. The output of the gate is directly connected to the microcontroller.

3.1.2.3 Clock Generation

The base board can either be driven by the onboard oscillator or an external clock source. The onboard oscillator is referred to as internal clock. Depending on the selected clock source, the base board operates in either a clock master or a clock slave mode. The term clock master describes the case where the internal clock is used and all required clocks are derived from the internal clock. The term clock slave describes the case where an external 48 kHz signal is connected via the BNC input. All other clock signals are derived from that input. This allows to clock multiple base boards by the same clock master, which means that all base boards can sample coherently. Details are illustrated in Fig. 3.4.

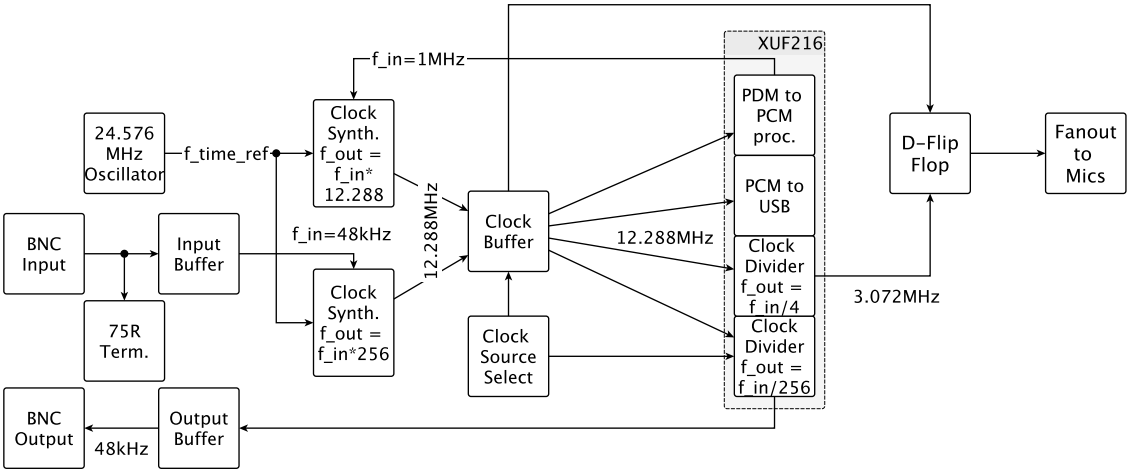


Figure 3.4: Clock Generation part of the base board

Internal clock generation The audio master clock runs at 12.288 MHz. For the internal audio master clock generation, a 1 MHz reference signal is generated in the XUF216 microcontroller and sent to the clock synthesizer. The clock synthesizer is the CS2100CP from Cirrus Logic [56]. It requires two inputs: a frequency reference from which the output is derived and a low jitter timing reference for the internal Phase-Locked Loop (PLL). The clock reference generator is the ASFL1-24.576MHZ-EC-T from Abracon [57].
The clock synthesizer runs in the high multiplication 20.12 format mode, which was inherited from the reference design. The format defines the integer and fractional portion of multiplier. In this case the 20 Most Significant Bit (MSB) define the integer

portion while the 12 Least Significant Bit (LSB) define the fractional portion. The value of the multiplier is calculated as follows [56]:

$$\begin{aligned} &\frac{\text{Output clock frequency}}{\text{Input clock frequency}} \cdot 2^{12} \\ &= \frac{12.288\,\text{MHz}}{1\,\text{MHz}} \cdot 2^{12} \\ &= 50331.648 \\ &\approx 50332 \end{aligned} \tag{3.7}$$

A detailed configuration description of the CS2100 can be found in App. D. The output of the clock synthesizer is connected to the clock buffer which provides two inputs and four outputs. The respective input is selected with an external switch. The clock buffer is the LMK00804B-Q1 from Texas Instruments [58].
The outputs of the clock buffer are sent to the two tiles of the XUF216 which generates two clock signals from the incoming 12.288 MHz signal: the first one is the 3.072 MHz clock which drives all microphones. The second one is the 48 kHz signal that is used as the reference output for an other clock slave.
For the microphone clock and the reference clock generation, the divider ratio is an integer. Integer divisions can be performed with the XUF216 internal clock dividers.

The outgoing 3.072 MHz clock for the microphones is buffered with a D-Flipflop. The flipflop is the 74AHC1G79GW,125 from Nexperia [59]. The clock input of the Flipflop is driven by the 12.288 MHz master clock. This helps to reduce jitter created by the division of the master clock in the XUF216. The output of the flipflop is connected to a two input, twelve output fanout buffer. The fanout buffer is the PI6C5921512 from Pericom [60]. This IC accepts single ended and differential inputs. For the base board created in this thesis, only one single ended input is used. This requires the negative input of the differential pair to be biased at $V_{\text{DD}}/2$ [60]. The other input is tied to ground with 1 kΩ resistors.

When running on the internal clock, the XUF216 also generates the 48 kHz clock that is sent to the BNC output and can be used to synchronize a clock slave. This operation utilizes an additional clock block to divide the incoming 12.288 MHz clock to the required 48 kHz by the factor of 256. The 48 kHz clock is buffered. As usual in pro audio equipment these clock lines have an impedance of 75 Ω. In order to match the output impedance of the clock buffer, the V_{OH} characteristic [61] is used

to determine the required resistor to match the impedance:

$$\begin{aligned} R_{\text{diff}} &= \frac{dU}{dI} \\ &= \frac{2.4\,\text{V} - 2.3\,\text{V}}{24\,\text{mA} - 16\,\text{mA}} \\ &= 12.5\,\Omega \end{aligned} \tag{3.8}$$

The output impedance of the clock buffer is approximately $12.5\,\Omega$ with an additional resistor in series of approximately $R_{\text{TX}} = 60\,\Omega$, the impedance requirements are satisfied. The circuit is shown in Fig. 3.5.

External clock generation When the base board is driven by an external clock, this signal needs to match the sampling frequency of 48 kHz. The incoming signal is sent to the clock synthesizer which is the identical model already described in the previous paragraph. Both synthesizers use the same onboard 24.576 MHz oscillator as timing reference. This time, the incoming clock with 48 kHz needs to be upscaled to 12.288 MHz with factor of 256. The clock synthesizer is running in high resolution mode. This mode expects a 12.20 format with 12 MSB and 20 LSB. The calculation of the factor is as follows [56]:

$$\begin{aligned} &\frac{\text{Output clock frequency}}{\text{Input clock frequency}} \cdot 2^{20} \\ &= \frac{12.288\,\text{MHz}}{48\,\text{kHz}} \cdot 2^{20} \\ &= 268435456 \end{aligned} \tag{3.9}$$

The output of the clock synthesizer feeds the same clock buffer as the internal clock synthesizer. This has already been described in the previous paragraph. When the clock source select switch is set to the external clock, the XUF216 does not derive the 48 kHz clock from the incoming 12.288 MHz and sets the output to zero. This avoids operating errors.
Assuming there could be more than one clock slave connected to the clock master, then the last participant needs to terminate the clock line with $75\,\Omega$. Since $75\,\Omega$ is a relatively small resistance for the clock master, the design aims to reduce the impedance after the rising edge. At this time, the impedance can be neglected. This can be achieved with a capacitor in series with the resistor. Each rising edge of the clock signal charges the capacitance and makes the $75\,\Omega$ resistance visible to the transmitter. Once the capacitance is charged, the current flow through the resistor

can be neglected. The capacitance is calculated with ten times the rise time from the data sheet of [59]: $10 \cdot 1\,\text{ns} = 10\,\text{ns}$. A simplified circuit is shown in Fig. 3.5.

$$\tau = R \cdot C \qquad (3.10)$$
$$C = \frac{10\,\text{ns}}{75\,\Omega} = 133\,\text{pF}$$

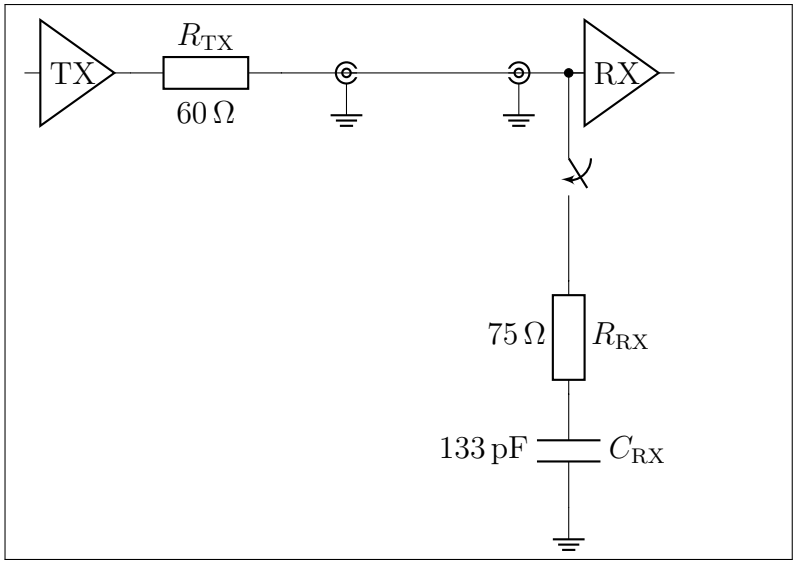


Figure 3.5: Impedance matching for external clock master

3.1.2.4 Data acquisition and USB communication

The XMOS XUF216 is the heart of the base board. Details on the architecture and firmware can be found in Sec. 2.4 and Sec. 3.2, respectively. The two main tasks of the microcontroller are the PDM to PCM decimation and to stream the PCM data to a host computer via the USB.
The functionality is illustrated in Fig. 3.6 and is described in more detail from a hardware point of view in the following.

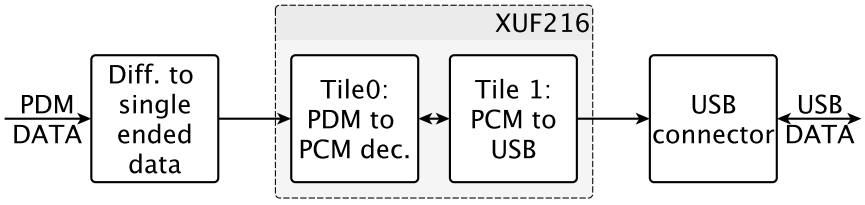


Figure 3.6: Top level view of the main XUF216 tasks

The XUF216 runs on 1 V and utilizes 3.3 V for its General Purpose Input Output

(GPIO) pins. It is clocked by a 24 MHz oscillator, the ASFL1-24.000MHZ-L-T from Abracon [57]. This clock is used to drive the microcontroller and all clocks that are not related to the audio part of the processing.

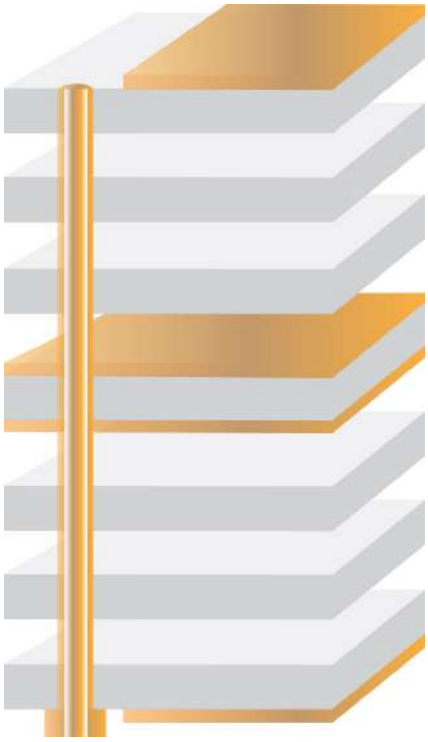
Data acquisition The data from the microphone PCBs is transmitted as a differential signal to the base board and is then converted to single ended, with the SN65LVDT386 from Texas Instruments [62]. The required 100 Ω termination resistors are already included, so there is no need for additional external resistors. The 16 single ended data lines are directly connected to the input ports of the XUF216.

USB communication The XUF216 has a build-in USB Physical Layer (PHY) and can therefore be connected directly to an USB connector which requires the following pins:

- **USB_VBUS** This pin is required if the device is not powered from the USB bus. As soon as a connection is detected, the internal pull-up resistors from the data lines are disconnected. In order to be compliant with the USB specifications, a 2.2 µF capacitor is connected to the VBUS line. The combination of the USB cable induction and the capacitor can cause an overvoltage transient. This transient is attenuated with a passive Low-Pass Filter (LPF) consisting of a 10 kΩ resistor and a 100 nF capacitor which results in a cut-off frequency of $f = \frac{1}{2\pi RC} \approx 160\,\text{Hz}$.
- **USB_DM, USB_DP** These are the differential data lines of the USB bus. They are protected against Electro Static Discharge (ESD) with a diode array. The diode array is the TPD2E001 from Texas Instruments [63].

3.1.2.5 PCB structure

The PCB consists of four layers. Using different layers for different purpose helps to lower the noise floor on the PCB and makes it easier to route signals from point to point without a large amount of vias to switch layers. The PCB structure used as well as the layer utilization is described in Fig. 3.7.



Type	Thickness	Usage
Layer 1	0.18 mm	Data top
Prepreg		
Prepreg	0.116 mm	
Prepreg	0.116 mm	
Layer 2	0.71 mm	Ground
Prepreg		
Layer 3		Power
Prepreg	0.116 mm	
Prepreg	0.116 mm	
Layer 4	0.18 mm	Data bottom
Prepreg		

Figure 3.7: Multilayer structure of the used PCB [64], Courtesy of beta LAYOUT GmbH

Impedance matching In order to minimize reflections, the transmission line should match the input impedance of the receiver. Therefore, the trace impedance needs to be calculated. The impedance for a differential micro strip is calculated in Eq. 3.11 and for a single ended microstrip in Eq. 3.13. All impedance calculations are based on [65].

$$Z_{\text{diff,microstrip,surface}} = 2Z_0 \cdot \left(1 - 0.48 \cdot \exp(-0.96\frac{s}{h'})\right) \tag{3.11}$$

Where:

$$h' = h \tag{3.12}$$

$$Z_0 = \frac{87}{\sqrt{\varepsilon_r + 1.41}} \cdot \ln\left(\frac{5.98 \cdot h}{0.8 \cdot w + t}\right) \tag{3.13}$$

$$t_{\mathrm{pd}} = 1000 \cdot \left(\frac{1.017\sqrt{0.457\varepsilon_r + 0.67}}{12(\mathrm{Inch})}\right) \tag{3.14}$$

$$C_0 = \frac{t_{\mathrm{pd}}}{Z_0} \tag{3.15}$$

$$L_0 = \frac{Z_0^2 C_0}{12} \tag{3.16}$$

With h: distance from signal trace to reference plane, t: height of the signal trace, w: width of the signal trace, s: spacing between differential traces.
Z_0: Impedance of the single microstrip, t_{pd}: intrinsic propagation delay, C_0: intrinsic line capacitance and L_0: intrinsic line inductance.

The following values are used for the impedance matched differential traces:
$h = 400\,\mu\mathrm{m}$, $t = 35\,\mu\mathrm{m}$, $w = 400\,\mu\mathrm{m}$, $s = 200\,\mu\mathrm{m}$ and $\varepsilon_r = 4.3$
This results in $Z_{\mathrm{diff,microstrip,surface}} = 97.6567\,\Omega \approx 100\,\Omega$.

Test Points & Spare pins The PCB has multiple Test Points (TP) for the validation of signals and supply voltages. Most of the TP are copper pads that are integrated in the signal traces while TP that are going to be used for a longer period of time are single pins. A detailed description of the TP can be found App. C.
To solve unforeseen problems easily, four additional GPIOs per tile have been added to pins on the outside of each tile. The pins are named TxSy, where x describes the tile number and y the pin number. The pin assignments can be found in App. B.

Power distribution The three used voltages of 5 V, 3.3 V and 1 V are distributed on layer three. The whole copper plane is fed with 3.3 V. Single traces for 5 V and 1 V are placed individually. The focus for these two traces lays on sufficient trace thickness and distribution from a centralized point close to the voltage generator.

Heat sinks The XUF216 and both clock fan-out buffers require heat sinks. All three ICs have a larger ground pad in which several ground vias were inserted. This helps to dissipate the heat through the bottom of the PCB.

Increasing the PCB resonance frequency The PCB is covered with a net of ground vias from the top to the bottom layer. The vias prevent excitation of low frequency modes within the PCB. These vias have a distance of $d = 10\,\text{mm}$ to each other. This results in the resonance frequency of the fundamental oscillation mode at about $f_{\text{res}} = \frac{c_0}{2\sqrt{\varepsilon_r} \cdot d} \approx 7\,\text{GHz}$. With c_0 as the vacuum velocity of light, and $\varepsilon_r \approx 4.3$, the relative permittivity of the FR4 PCB material.
In this oscillation mode two adjacent wave nodes are located at the via positions. This is the lowest possible frequency at which resonances can occur. Resonances at lower frequencies are suppressed by the via grid structure. Resonances at f_{res} or above will not be excited because of the limited frequency spectrum of the digital signals.

3.1.3 Transmission line length

The setup and hold times are specified by XMOS in [66]. For an XMOS device operating at 500 MHz, the timing requirements are as follows: $t_{\text{su}} = 21\,\text{ns}$ and $t_{\text{h}} = -11\,\text{ns}$. The negative value for the hold time implies that the hold point is located before the rising edge of the clock signal. The detailed clock and data path are illustrated in Fig. 3.8.

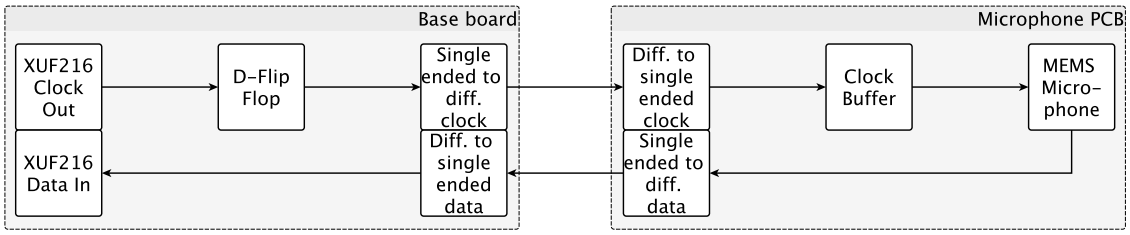


Figure 3.8: Clock and data path of the system

The used microphones specify the following timings [45]:

- Time until data line driven after falling edge
  $t_{\text{dd, min}} = 18\,\text{ns}$, $t_{\text{dd, max}} = 35\,\text{ns}$
- Time until data valid for max. capacitance
  $t_{\text{dv}} = 100\,\text{ns}$
- Hold time
  $t_{\text{h}} = 5\,\text{ns}$

These values do not satisfy the timing requirements of the XMOS specifications. Since it is possible to operate two microphones at the same data line (first microphone outputs data on the rising edge of the clock, second microphone on the falling edge of the clock), the microphones set their outputs to HI-Z after the hold time. This project utilizes only one microphone per data line. A timing measurement on the microphone PCB revealed that the capacity of the trace is sufficient to hold the signal until the next clock edge. The timings are $t_{\mathrm{dd}} = 28\,\mathrm{ns}$ and $t_{\mathrm{dv}} = 35\,\mathrm{ns}$. The timing requirements are illustrated in Fig. 3.9.

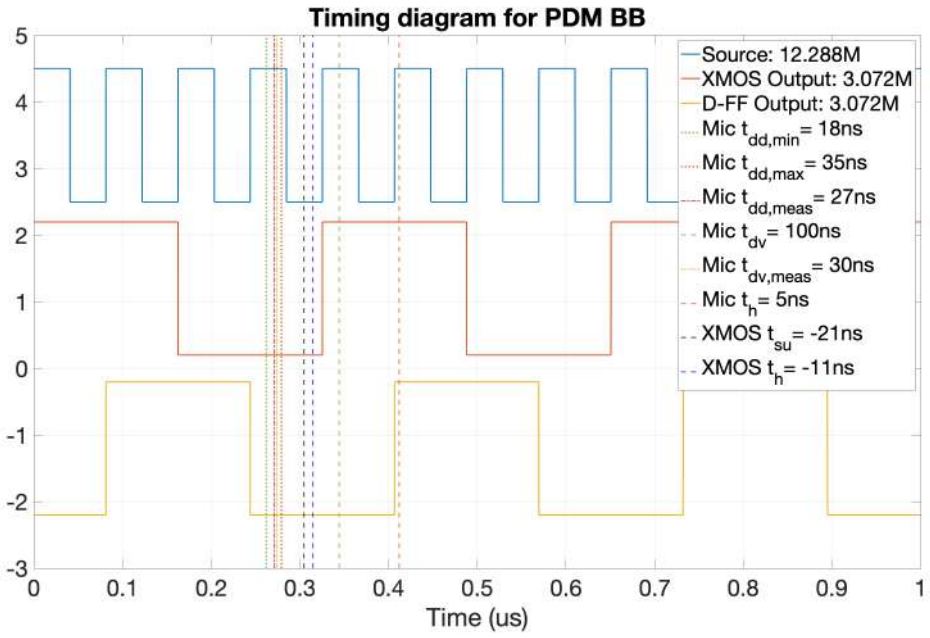


Figure 3.9: Timing diagram for the base board

The timings illustrated in Fig. 3.9 do not take the cable length into account. The maximum cable length is calculated as follows:

$$t_{\mathrm{su,abs}} = T_{3.072\,\mathrm{MHz}} - t_{\mathrm{su,rel}} = 325.5\,\mathrm{ns} - 21\,\mathrm{ns} \qquad = 304.5\,\mathrm{ns} \tag{3.17}$$

$$\begin{aligned} t_{\mathrm{dv,meas,abs}} &= T_{12.288\,\mathrm{MHz}} + T_{3.072\,\mathrm{MHz}}/2 + t_{\mathrm{dv,meas,rel}} \\ &= 81.4\,\mathrm{ns} + 162.8\,\mathrm{ns} + 35\,\mathrm{ns} \qquad = 279.2\,\mathrm{ns} \end{aligned} \tag{3.18}$$

This results in a timing overhead of $t_{\mathrm{su,abs}} - t_{\mathrm{dv,meas,abs}} = 25.3\,\mathrm{ns}$ which needs to be split between the outgoing and incoming signal. With a velocity of $0.2\,\frac{\mathrm{m}}{\mathrm{ns}}$ for signals in copper [67], maximum cable length is therefore $\mathrm{len}_{\mathrm{max}} = 0.2\,\frac{\mathrm{m}}{\mathrm{ns}} \cdot 12.65\,\mathrm{ns} = 2.53\,\mathrm{m}$. Further improvements for this setup are discussed in Sec. 5.

3.2 Firmware design

XMOS provides reference implementations for its USB audio platforms. The signal processing part depends on the application area, while the USB communication requires mandatory parts that are listed in Tab 3.5.

Component	Description
XMOS USB Device Driver (XUD)	Low-level USB I/O
Endpoint 0	Logic for endpoint 0
Endpoint buffer	Buffers endpoint data packets to and from the host
Decouple	Audio packet transfer between the endpoint buffer and the audio components
Audio Driver	Audio data transfer to/from other audio components

Table 3.5: Mandatory components of the XMOS USB subsystem [68]

The used microprocessor in this thesis is the XUF216 [35]. It consists of two tiles with eight cores per tile. For the processing of 16 microphones a total number of seven cores is required [69]. This occupies one tile and leaves the other tile for managing the USB communication. The cores on the USB have to meet the timing requirements for the High-speed transmission grade, which requires them to operate with at least 80 Million Instructions per Second (MIPS) [68]. With a tile frequency of 500 MHz and the assumption of one instruction per cycle, this results in a maximum number of usable cores of: 500 MHz/80 MIPS $\approx$ 6 cores. A detailed overview of the core usage is given in Tab. 3.6 and Tab. 3.7 and is visualized in Fig. 3.10.

Component	Cores
XUD	1
Endpoint 0	1
Endpoint buffer	1
Decouple	1
Audio Buffer	1
Audio reference clock	1
Total:	**6**

Table 3.6: Core usage on Tile 0

Component	Cores
PDM RX 0 to 7	1
PDM decimator 0 to 3	1
PDM decimator 4 to 7	1
PDM RX 8 to 15	1
PDM decimator 8 to 11	1
PDM decimator 12 to 15	1
PCM Buffer	1
Word clock Generator	1
Total:	**8**

Table 3.7: Core usage on Tile 1

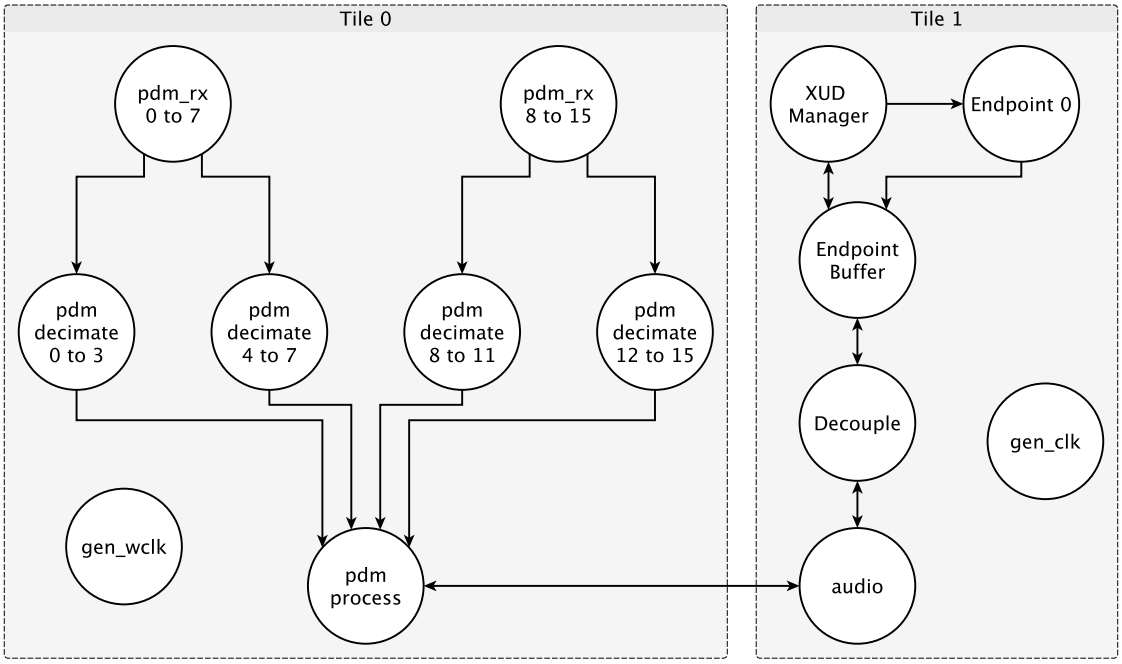


Figure 3.10: Core usage and communication between cores

3.2.1 Project Structure

The implementation for this thesis is based on the reference design from the XMOS Microphone Array Development Kit [31]. The project contains the main code for PDM processing and additional libraries and modules from XMOS for the USB communication as well as the control of the external clock synthesizer. The project structure is shown in Fig. 3.11. A brief description of the used libraries and modules is given in Tab. 3.8. To include a library or a module it needs to be added to *USED_MODULES* define in the Makefile of the application.

The first step was to examine the reference implementation. It turned out that the code base is basically one big project, containing not only the reference code for the array hardware, but for all platforms. It is tailored by *#ifdef* and *#ifndef* preprocessor directives. Depending on the active project, the appropriate components are then used to generate a bit file for the microcontroller.
Before the actual work on the code started, the code base was cleared of unnecessary ifdef directives as well as unused code snippets to keep the code lean.
In a second step, the USB and PDM to PCM components were then examined and

adapted if necessary.

Module	Description
lib_logging [70]	Debugging framework for compile time controlling
lib_mic_array [69]	PDM to PCM decimation
module_dfu [71]	Device Firmware Upgrade code
module_i2c_single_port [72]	Supports a single i2c master where SDA and SCL are shared on a single port
module_usb_device [73]	Common code for USB device applications
module_usb_shared [73]	Common code for USB applications
module_xud [68]	Low level USB device library

Table 3.8: Description of used modules and libraries

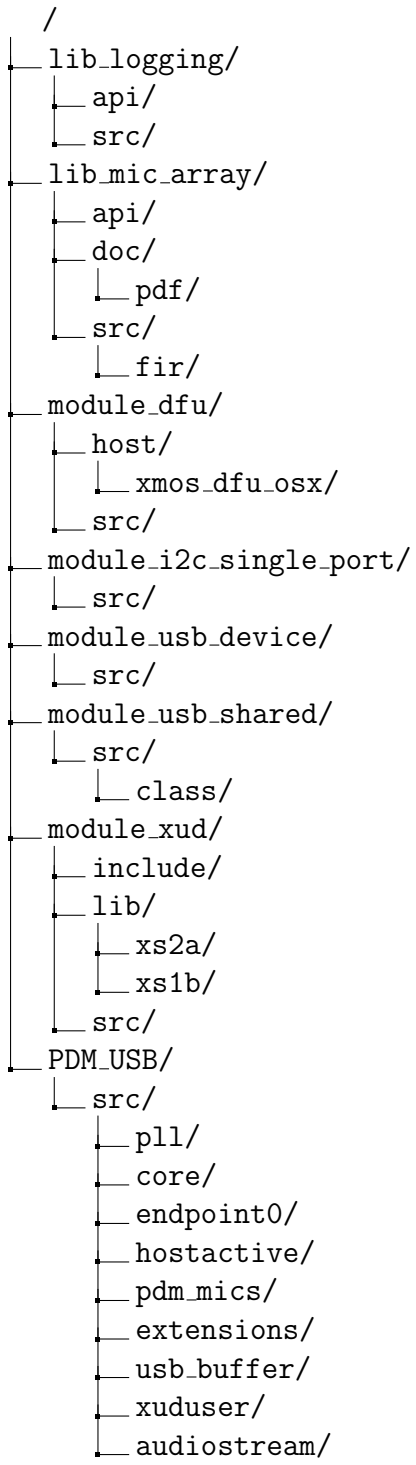


Figure 3.11: Project structure

3.2.2 USB implementation

The XMOS USB audio framework supports the USB Audio Class 1.0 (UAC1) and the USB Audio Class 2.0 (UAC2).
The UAC1 was specified for the USB 1.0 standard which allowed for a maximum bit depth of 24 bit at a maximum sampling rate of 96 kHz. The maximum speed grade of USB 1.0 is Full-speed with a bandwidth of 12 Mbit/s. This is not sufficient for this project since it requires a bandwidth of 18.43 Mbit/s (as calculated in Sec. 2.5). The UAC2 extends the previous specification by increasing the maximum bit depth to 32 bit and allows for sampling rates larger than 96 kHz. With the transmission speed of the High-speed grade, a maximum bandwidth of 480 Mbit/s is available. The hardware developed in this thesis is specified for UAC2 only, as it provides sufficient bandwidth and all major operating systems support UAC2.

Tile 1 of the XUF216 runs the USB communication and audio buffering backend. Furthermore, one core is utilized to generate the audio reference clock. The 12.288 MHz audio master clock is derived from this audio reference clock.
This section describes the implementation details of the USB part and explains some XMOS specific adjustments based on [68].

In the XMOS implementation, not all requests are handled as defined in the USB specification. Some of the standard requests, class requests, and audio requests are processed in a different way. This is implemented with functions differently than specified but leads to the same result. A detailed description of this behavior would require an extensive description of the usual behavior, in order to then show the deviations from this behavior which adds only little value to this thesis. Furthermore, since no changes were made to this part of the implementation, readers are referred to the literature in [68].

Endpoint 0 implementation

Endpoint 0 is mandatory for every USB device. This endpoint controls the management tasks of the USB device. The endpoint 0 code runs on its own core to guarantee fast response to requests. Tasks processed on this core can be divided into enumeration, audio configuration and Device Firmware Upgrade (DFU) requests.

Enumeration The enumeration process describes how the USB host is interrogating the device about its functionality. The device is reset, answers to the requested

descriptors and is then assigned to an address by the host.
The device must support the mandatory standard request that are specified in the "USB_StandardRequests()" function. This includes the Device Descriptor, Configuration Descriptor and the corresponding String Descriptors as well as the GET/SET Interface requests, the GET/SET Configuration requests and the SET Address request. The function stalls unknown requests which are then parsed to device specific requests.

Audio endpoint implementation

The following sections describe details of the different cores shown in Fig. 3.10.

Endpoint Buffer The endpoint buffer handles data that is sent and received by the XUD core and the endpoint 0 core.

Decouple core The decouple core introduces additional buffers for transmitting and receiving audio data from and to the host. These buffers are implemented as First in, First out (FIFO)s. The FIFO data is sent to other parts of the system via channels, given the other part requires it. The decoupling core also determines the packet size for each audio transmission to the host.

Audio Buffering Scheme The system designed in this thesis can be seen as data source to the host. Therefore, only the transmitting scheme is described here. This is done in the following way:

1. The audio core sends its samples to the decouple core which puts them in a FIFO. The samples in the FIFO are split into packets.

2. After sending the previous packet to the XUD core, the buffer core requests the next packet from the decouple core. This is done with a shared memory flag.

3. As soon as the buffer core requests the next packet, the decouple core passes the requested packet to the buffer core. Furthermore, the XUD core is notified by the decouple core that the buffer core is able to send the next packet.

4. When the buffer core has passed the requested packet to the XUD core, it signals the decouple core that the sending process is finished. The read pointer of the decouple audio FIFO is then moved to the next packet in the FIFO.

Decouple/ Audio Core interaction The timing requirements of the audio system must be met in any case. This means that the decoupling core must respond immediately to the audio system's requests to send samples. That is realized with the help of an interrupt handler in the decoupling core.
Sample transfers are requested by sending a word over a channel to the decouple core which activates the "handle_audio_request()" function to transmits samples. The interrupt handler answers to the request by acknowledging the word. Then the sample transfer takes place.

USB Rate Control The audio core provides the USB Data at the correct rate that matches the sampling frequency of the device. A Start of Frame (SOF) is sent by the host every 125 µs. Therefore the frame transmission rate is at 8 kHz. With a sampling frequency of 48 kHz, the average packet contains six samples per channel. Since the device runs in asynchronous mode, this can potentially cause a drift in the audio clock leading to a variation of ± 1 sample per packet. This is within the specified range for a sampling rate of 48 kHz.

Device Firmware Upgrade

The Device Firmware Upgrade (DFU) provides a way to update the firmware on the device via USB instead with the programmer with the debug header. To start a DFU process, the device needs to reset and start in the DFU mode. This is either done by sending a DETACH request and resetting the device within a specified time or by using the custom request XMOS_DFU_RESETDEVICE. This resets the device directly into DFU mode. Details on the requests can be found in [71]
As long as the device is in DFU mode, the active interface accepts the commands that are defined in the DFU device class specification [74]. Furthermore, the interface accepts the custom command XMOS_DFU_REVERTFACTORY which reverts the active boot image to the factory image.

Audio Stream Formats

The firmware is prepared to support up to three different stream formats for the data output of the device. Different stream formats are implemented by using Alternate Settings in the AudioStreaming (AS) interface. In the current firmware, the format is fixed to 24 bit at 48 kHz only.

3.2.3 PDM to PCM processing

Tile 0 of the XUF216 runs the PDM to PCM decimation cores that use the *lib_mic_array* [69]. This section is based on [69]. The library provides a configurable decimation filter to convert the incoming PDM samples into PCM samples. The process is divided into different stages, which allows to split the stages among several cores. The decimation process for 16 PDM inputs is shown in Fig. 3.12. Each gray rectangle with a dashed outline represents one core. This means that six cores are required to process the PDM data and an additional core is required to transfer the samples to the USB backend of the microcontroller.

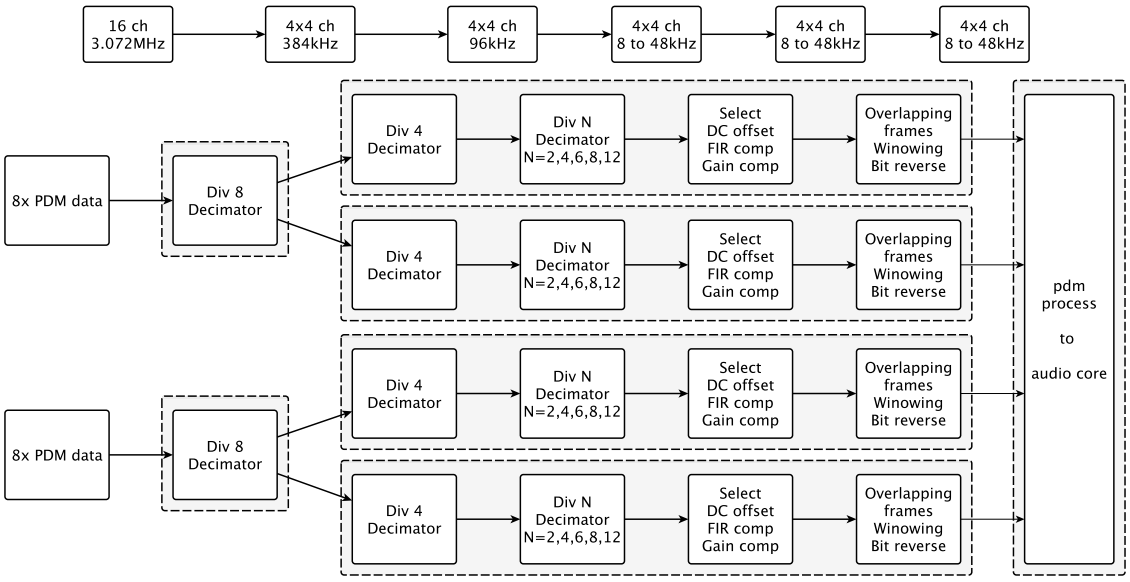


Figure 3.12: Sixteen count PDM interface, Figure from [69]

Hardware characteristics

All microphones are driven by the same clock signal and output their data on eight separate lines which are connected to a single 8 bit port. The library only supports data sampling on the rising edge of the clock.

Accessing the samples

Samples are accessed in the form of frames. A frame is returned from the decimators in either the time domain format or in the FFT ready format. Since this project does not contain any signal processing that requires operations in the frequency domain, only the time domain frames are explained.

Time domain frames Each time domain frame consists of a single two dimensional array. The first array dimension describes the channel number and the second dimension the sample number. The sample number zero describes the oldest sample. Newer samples increase the sample number. The sample counter has a length of 7 bit and can therefore store up to 128 samples before the oldest samples are overwritten.

Decimator stages

The decimation process can be divided into two main stages (compare to Fig. 3.12). The first stage processes the incoming 1 bit PDM high frequency data stream to a lower frequency but higher bit count data stream.
The second stage applies additional decimation and frequency shaping.

First decimation stage The first decimation stage contains a decimator with a fixed divider of 8, which decimates the 16 incoming microphone channels with a frequency of 3.072 MHz to two times four output streams with a frequency of 384 kHz.

Second decimation stage The second decimation stage contains two decimators. The first one has a fixed divider of 4, the second decimator is user defined. Furthermore, there are options for DC offset removal, Finite Impulse Response Filter (FIR) compensation, gain compensation, frame overlapping, windowing and bit reverse.
The first decimator converts the incoming 384 kHz signals to 96 kHz signals. In order to obtain a sampling rate of 48 kHz, the second decimator is set to the factor 2.

DC offset removal This optional parameter can remove the DC offset of the signal with a High-Pass Filter (HPF). It is implemented as a single pole Infinite Impulse Response Filter (IIR) filter and obeys the following relation:

$$Y[n] = Y[n-1] \cdot \alpha + x[n] - x[n-1] \tag{3.19}$$
$$\text{With } \alpha = 1 - 2^{-\text{MIC\_ARRAY\_DC\_OFFSET\_LOG2}}$$
$$\text{MIC\_ARRAY\_DC\_OFFSET\_LOG2} \in [0, 31]$$

The value of α controls the cut-off frequency of the filter with the MIC_ARRAY_DC_OFFSET_LOG2 coefficient. A negative sign is missing for the coefficient in the original specification which is corrected in Eq. 3.19. The highest cut-off frequency is at 5.528 kHz for a coefficient of 0. With increasing coefficients, the cut-off frequency moves towards 0 Hz. The default coefficient of 13 achieves a cut-off frequency of 1 Hz.

Gain compensation The library offers the option for microphone gain compensation. This requires the knowledge of the gain of the quietest microphone which is not present and therefore not applied.

FIR filtering The final filtering stage contains a FIR filter which can either be used with default or custom values. Furthermore, a FIR gain compensation can be applied.
The developed device uses the default coefficients for the FIR filter and does not apply FIR gain compensation.
The frequency response of the whole decimator is shown in Fig. 3.13. The cut-off frequency of the system is at approximately 19.2 kHz.

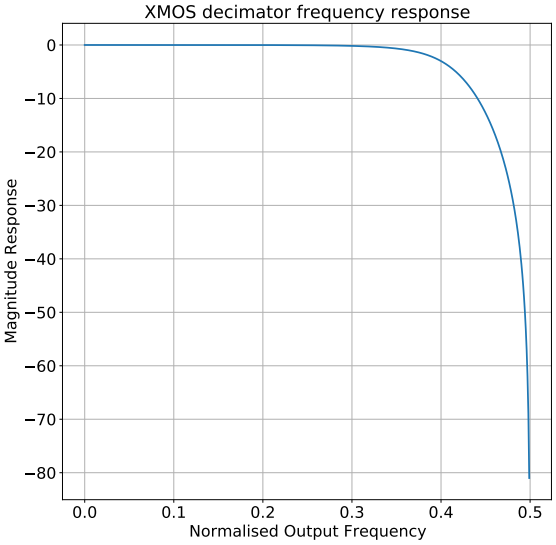


Figure 3.13: Frequency response of the whole decimator, Figure from [69]

Typical memory usage

The memory usage depends on the output sampling rate and the number of channels. Lower output sampling rates require a greater decimation which results in an increased memory requirement. For a sampling rat of 48 kHz, this results in about 18.1 kB for 16 inputs leaving 45.9 kB of the available 64 kB tile memory.

Chapter 4

Results

This section presents the results of the project. It is divided into two main parts: The first part contains clock and signal measurements for system validation. The second part contains acoustic measurements for the validation of functionality of the system.

4.1 Electrical validation

A single base board is used to validate all clock master functionality. Secondly, a cascaded system with two base boards in clock master - slave configuration is used. All frequency measurements were performed with the RTO 1004 from Rohde&Schwarz [75]. The results are described using certain Test Points (TP), although other TP have the same functionality. If the other TP are not mentioned separately, they show the same behavior and are therefore not described again.
The Printed Circuit Boards (PCBs) were manufactured by beta Layout GmbH and assembled and soldered in a reflow oven by the author at the Chair of Communications and Information Theory of the TU Berlin. The result is shown in Fig. 4.1. The described validation was also carried out at the chair.

The design has four hardware design errors, which are briefly described below.

Microphone PCB The used capacitors C9 and C11 introduce a Direct Current (DC) block, which is not needed here. As recommended in the PL133-37 [48] data sheet, the capacitors have been replaced with 30 Ω resistors.
The pin assignment of the RJ45 connector is reversed. This requires rewiring the ground pin and the 5 V pin. The ground pin is correctly assigned with a solder

bridge, the 5 V pin is connected to the appropriate place with a wire. Furthermore, the reversed pin assignment results in an inversion of the audio signal. Fortunately, this can be easily corrected in the processor by multiplying the samples by a value of -1.

Base board The base board uses a Single Pole Single Throw (SPST) switch (S2) for the termination of the external clock input. This type of switch can also be described as a push button that does not toggle between the two positions it offers. Therefore, the switch has been replaced by a manual jumper option.
During the design process, the differential signal pairs were swapped. This was fixed by the wrong pinout of the RJ45 connector on the microphone boards but should be considered in future revisions.

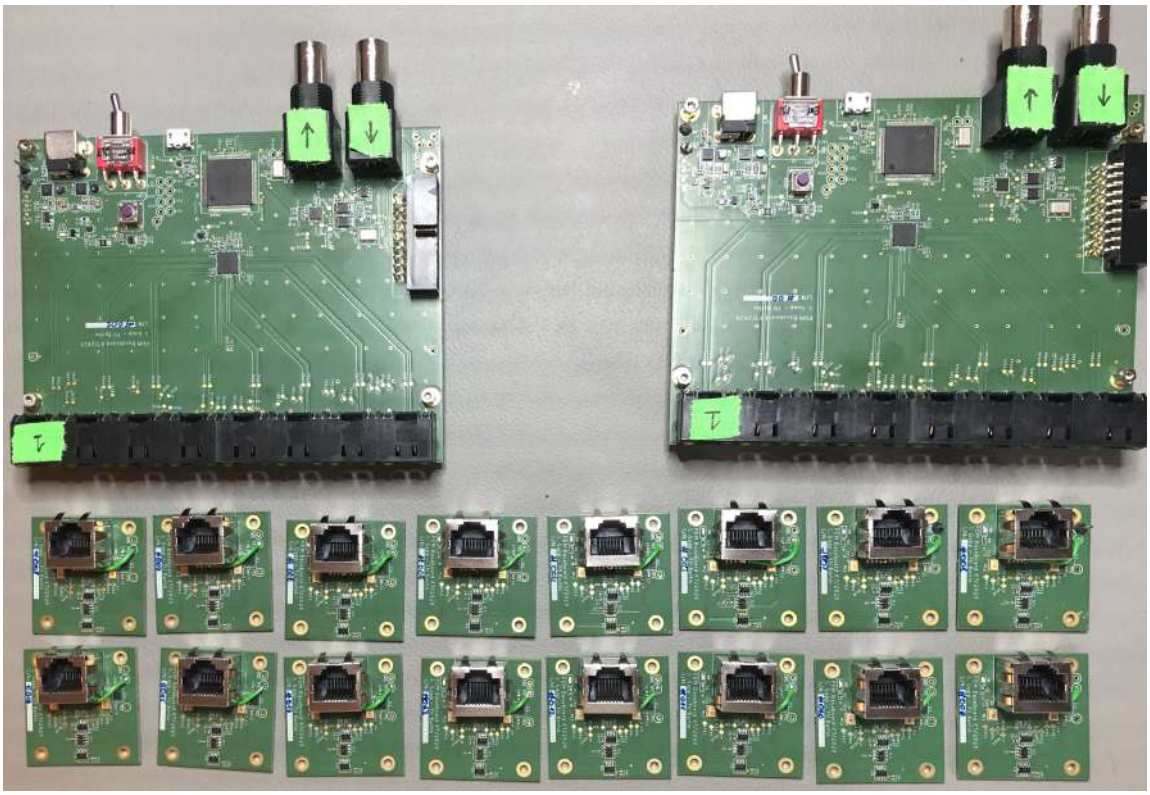


Figure 4.1: Manufactured base boards and microphone PCBs

4.1.1 Base board validation

The measurement setup consists of one base board (Identifier (ID): 001) and a microphone PCB (ID: 001). Measurements on the base board are used to validate correct operation of the system in clock master mode. The setup is shown in Fig. 4.2.

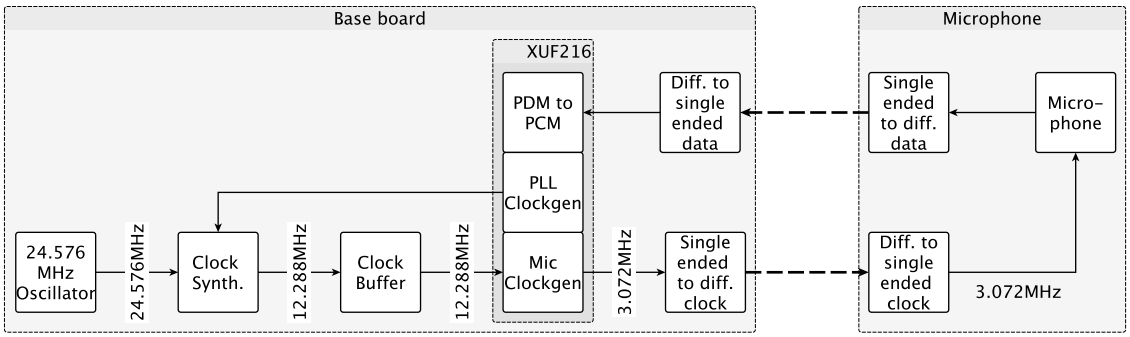


Figure 4.2: Measurement setup for a single base board

The measurement of the audio master clock at TP76 is shown in Fig. 4.3. The frequency requirement of 12.288 MHz is met. The clock shows some overshooting and ripple which is probably caused by an input mismatch of the clock buffer.

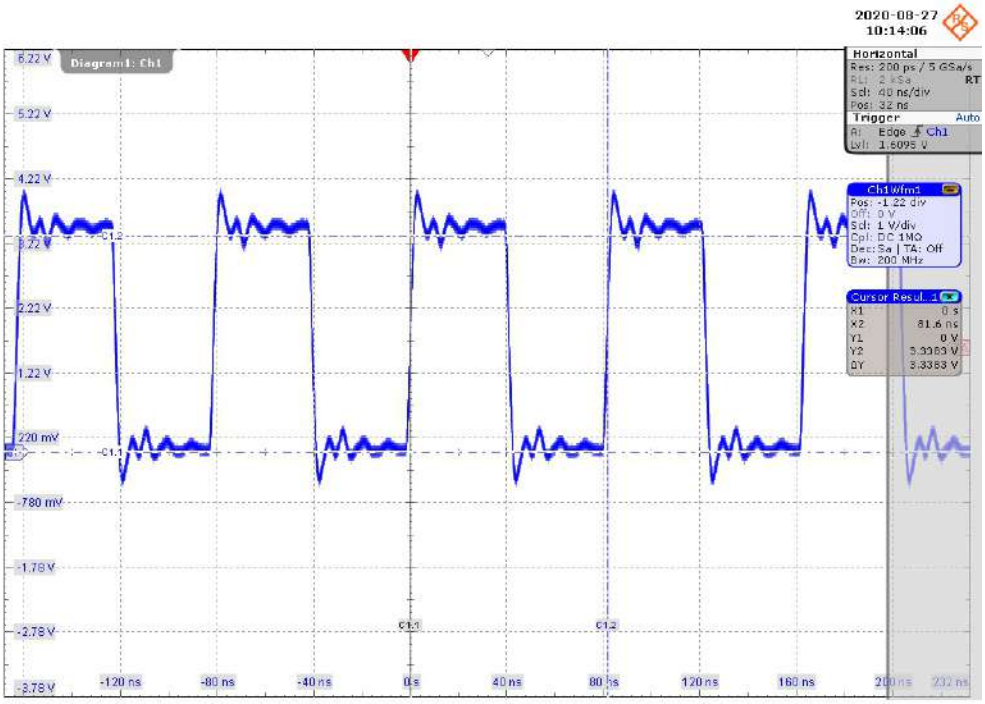
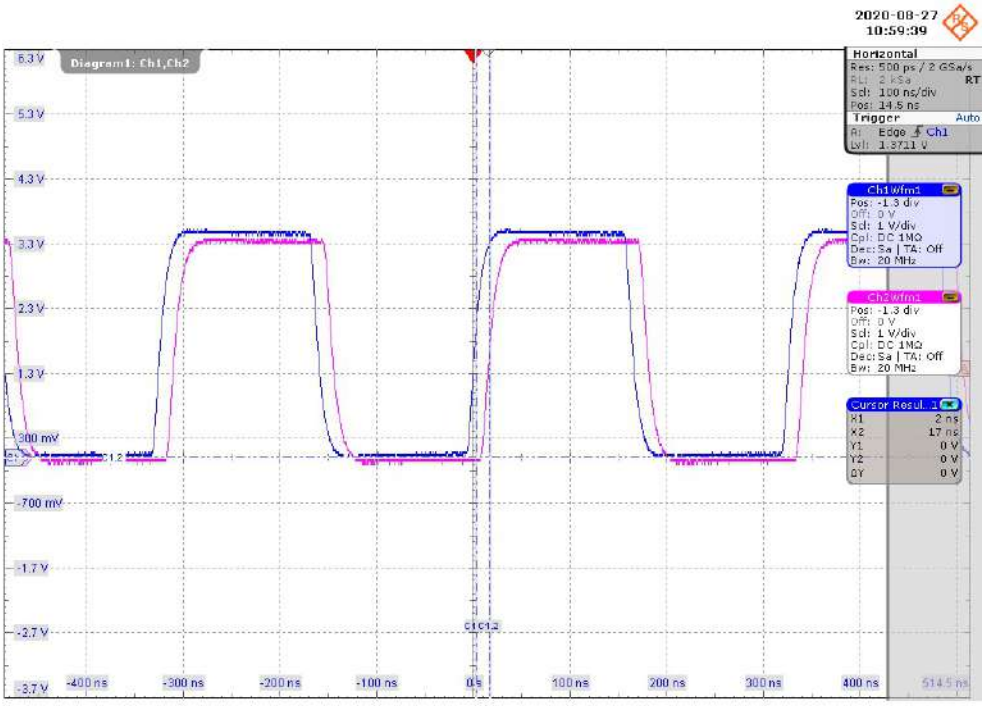
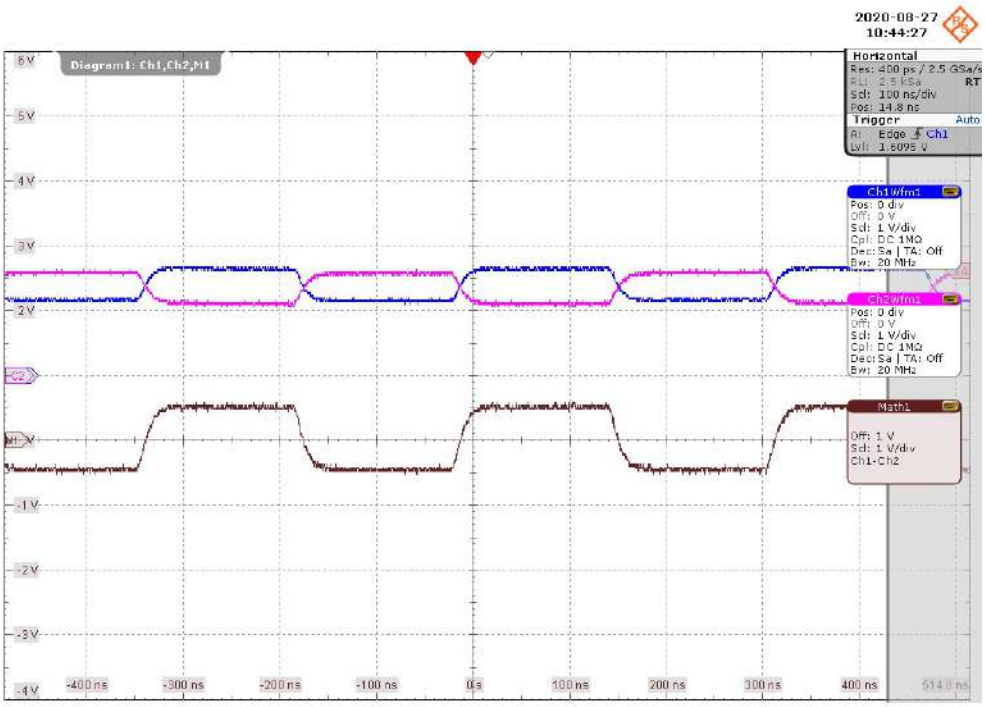


Figure 4.3: Audio master clock on the base board

The XUF216 derives the 3.072 MHz microphone clock from the audio master clock. In order to validate the clock itself and the differential transmission, TP79 on the base board and TP5 on the microphone PCB were measured at the same time. The result is shown in Fig. 4.4(a). The blue line describes the clock on the base board, the pink line the clock on the microphone PCB. The measurement shows a transmission delay of approximately 12 ns between the blue and the pink line. This matches the delay introduced by the cable length of 2 m and additional delay introduced by the converting circuitry.
The differential transmission was measured on the microphone PCB at TP3 and TP4. The result is shown in Fig. 4.4(b). The pink and the blue line show the differential transmission, the brown line describes the voltage difference calculated by the oscilloscope. It can be seen that the differential output has an amplitude of approx. 0.5 V while the calculated clock signal has an amplitude of approx. 1 V.



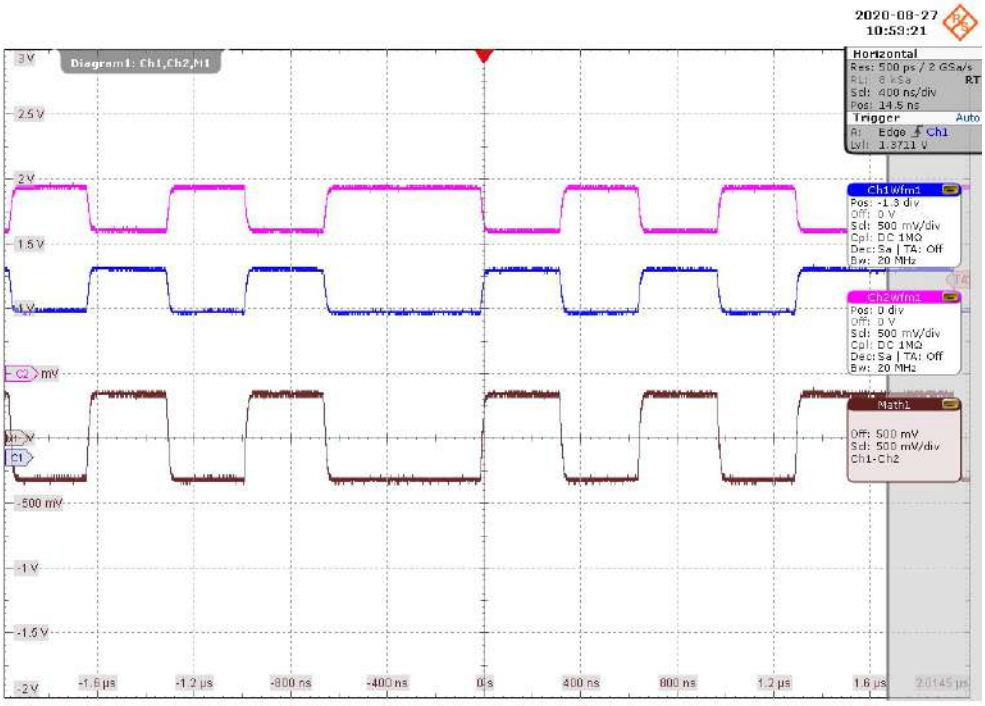
(a) Clock signal blue transmitter, pink receiver



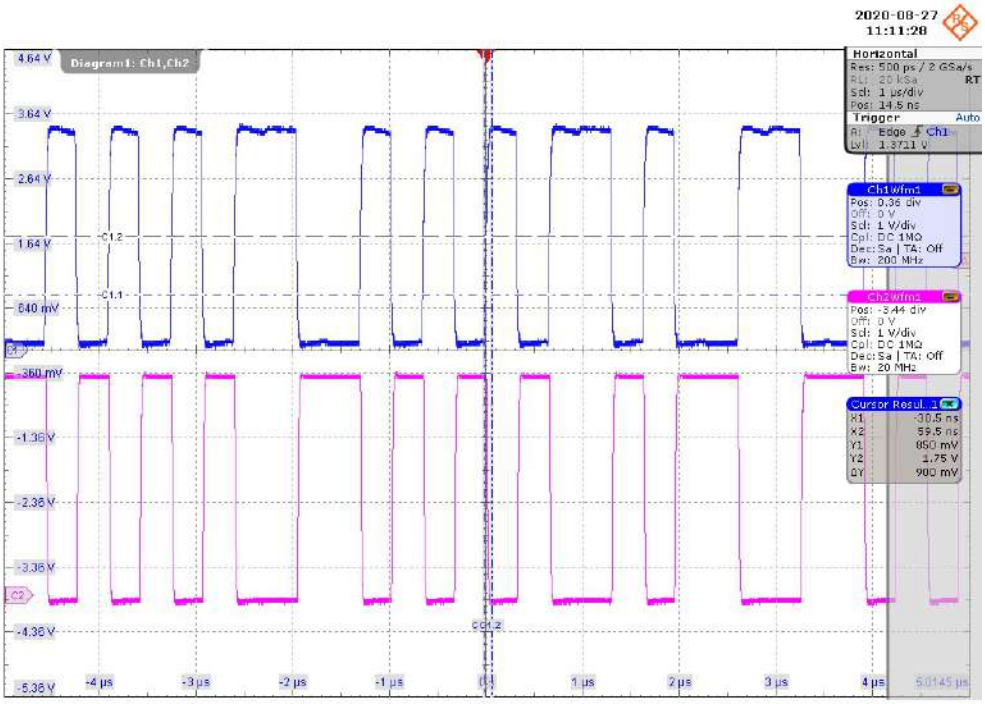
(b) Differential transmission and calculated clock

Figure 4.4: Transmitted and received clock signals

The data output of the microphone PCB is shown in Fig. 4.5(a). It can be seen that the amplitude is also within the Low Voltage Differential Signaling (LVDS) specifications for the differential signal as well as the calculated signal.
When comparing the data output from the microphone and the data input at the XUF216 in Fig. 4.5(b), it is obvious that there is a phase shift of 180°. As already described, this is due to a wiring error on the PCB where the differential inputs on the receiver were incorrectly connected. Fortunately, this error can be corrected by multiplying the sample value in the XUF216 by the factor -1.



(a) Differential and calculated data signal



(b) Recovered data signal sender (blue) and receiver (pink)

Figure 4.5: Data signals on the receiving base board

4.1.2 Cascaded base boards

For the measurements with a cascaded system, a second base board (ID: 002) and microphone PCB (ID: 002) are added to the clock master.
The clock master base board (ID: 002) and the slave base board (ID: 001) are connected via the respective clock output/input with a 75 Ω coaxial cable. The measurement setup for the cascaded system is shown in Fig. 4.6. With these measurements, the system function is validated and jitter measurements between the two base boards are performed. Furthermore, the delay between two receiving microphone PCBs are determined.
For the jitter measurement the 12.288 MHz audio master clocks are measured at TP76 and TP81, respectively. The result is shown in Fig. 4.7. The blue line describes the clock from the master base board, serving as a reference for the jitter measurement.

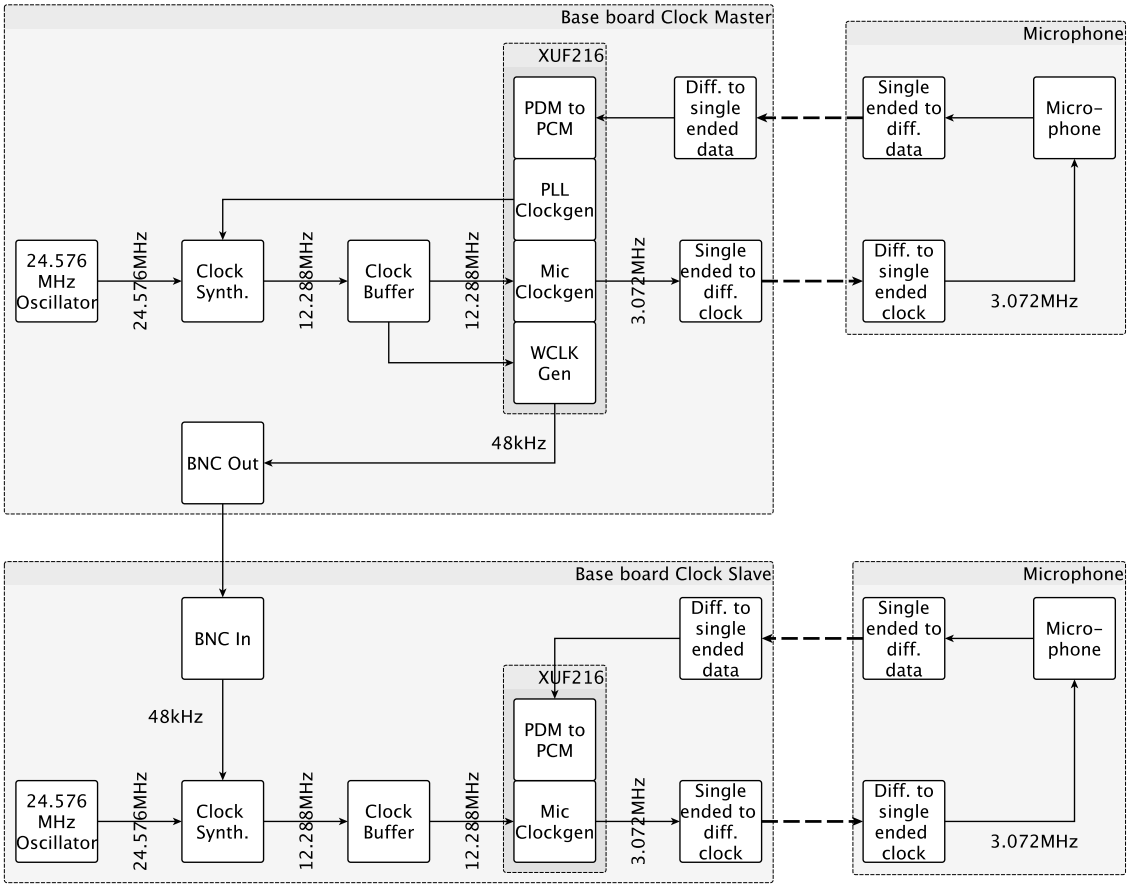


Figure 4.6: Setup for the cascaded base board measurement

The pink line describes the derived audio master clock on the slave base board. The measurement shows a jitter of approximately ±2.5 ns. The periods of the signals of interest have range from 50 ms (for 20 Hz) to 41.67 µs (for 24 kHz). They are very large in relation to the jitter period and therefore the jitter is negligible.
The delay between the base boards is a result of the 48 kHz slave clock generation in the XUF216 on the master base board. It was assumed that this is a static delay which can be compensated in the master microcontroller.
By measuring the phase relation of the 48 kHz input and the 12.288 MHz output at the clock synthesizer, it turned out that there is no defined phase relation between the input and output. The relation depends purely on the turn-on time and is therefore not deterministic. This makes a delay compensation in the master base board impossible. Improvements to this phenomenon are discussed in Sec. 5.

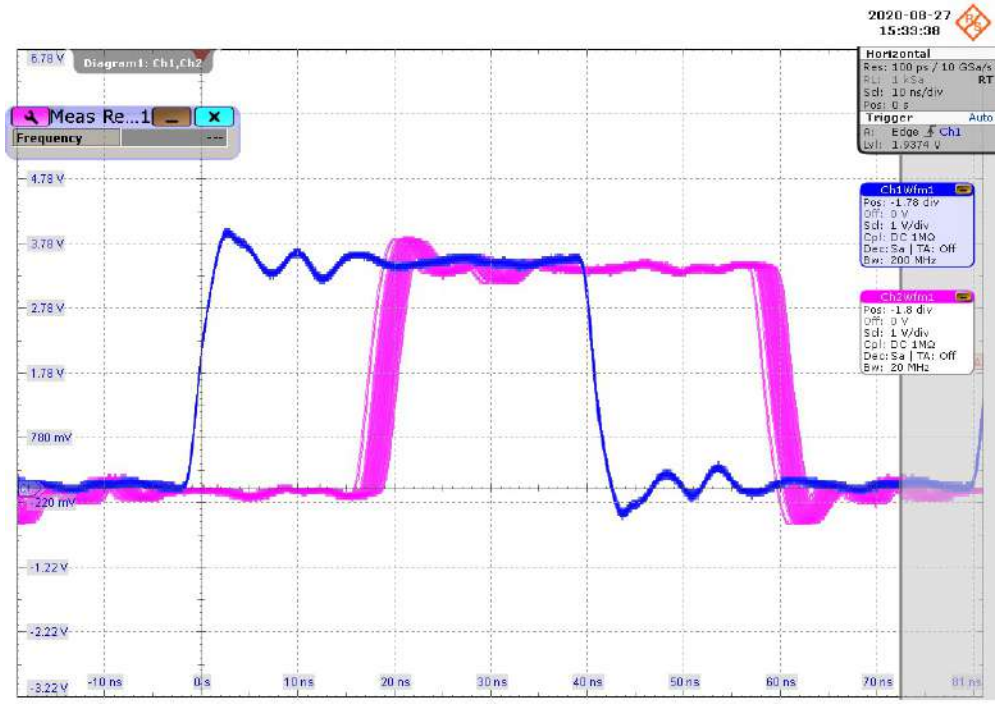


Figure 4.7: Jitter measurement of the 12.288 MHz clocks

4.2 Acoustic validation

To validate the acoustic functionality of the developed system, measurements were performed with the Acoular Framework [76], a linear arrangement of the microphones and one loudspeaker. Impressions from the actual measurement are shown in Fig. 4.11.

4.2.1 Measurement setup

The developed base board is seen as an external sound card by an operating system. Several external sound cards can be combined on the operating system level to one larger device. How the sound cards are combined depends on the operating system. In macOS this is supported natively in the form of an “aggregate device” in the

Audio Midi settings. In Windows a driver from ASIO4ALL [77] can be used and in Linux an "aggregate device" can be created using Jack [78].

For the linear arrangement of the microphone array, all 16 microphone PCBs are attached to an aluminum profile in a horizontal row. The PCBs are spaced 1.7 cm apart, resulting in an array length of 82.7 cm. The microphone spacing is shown in Fig. 4.8 and the arrangement of the microphone array is shown in Fig. 4.9.

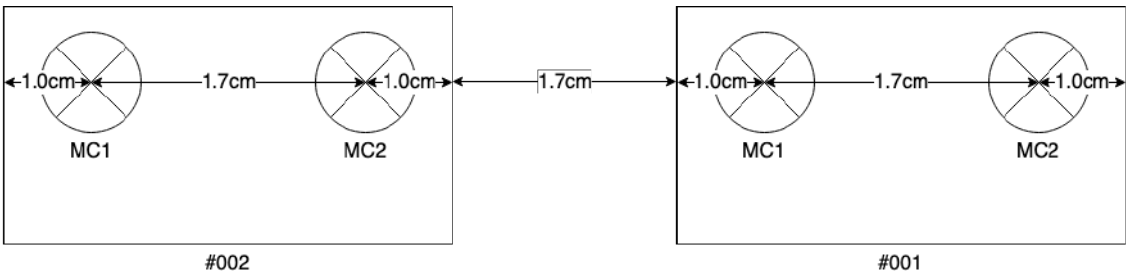


Figure 4.8: Microphone spacing

The microphone positions are labeled on the profile. The order of the serial numbers is listed in Tab. 4.1.

Position:	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01
Serial No:	07	04	08	06	02	16	14	15	013	05	09	01	10	12	03	11

Table 4.1: Microphone PCB positions and corresponding IDs.

Each base board can process eight microphone PCBs with two microphones per PCB. This results in a total number of 32 microphones for the system, of which 16 are processed by each base board..

The band in which the array can operate free of spatial aliasing is defined by the upper and lower frequency. The upper frequency is at $f_{\max} \approx 10\,\text{kHz}$ (see Eq. 3.1). The lower frequency is defined by the Sparrow limit, which describes the minimum distance between two sources that can be distinguished by the array. With the used setup it is possible to distinguish between different sources starting from a source-to-source distance of 6.49 m at 300 Hz and 0.13 m at 15 kHz (see Tab. 3.3 for more details).

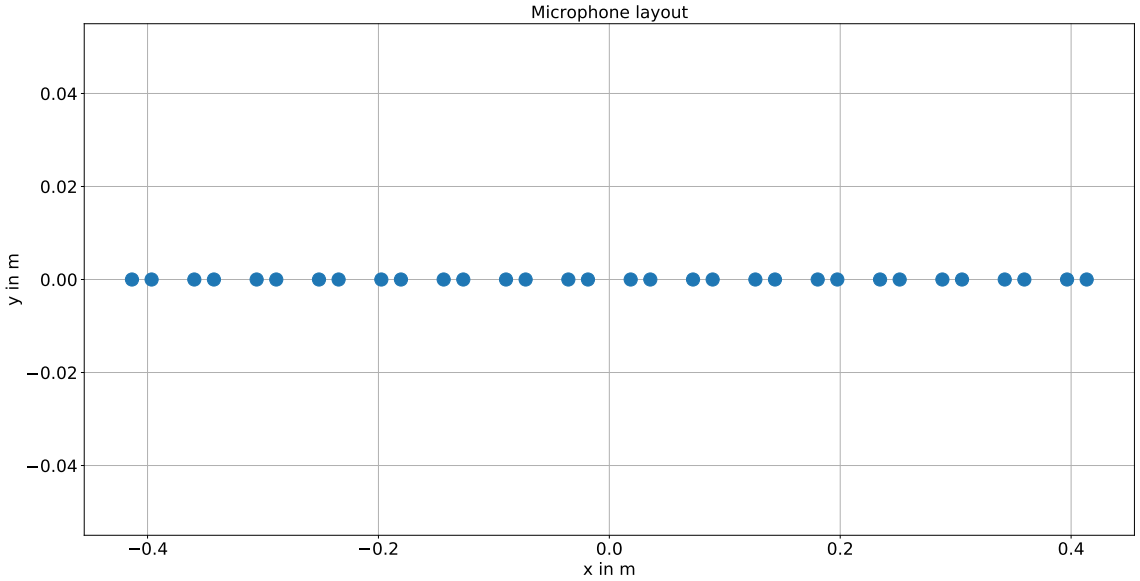


Figure 4.9: Microphone array layout

As a sound source a Mackie CR-4 BT was used [79]. The loudspeaker has frequency range from 70 Hz to 20 kHz with a 3dB deviation. Furthermore, it offers Bluetooth connectivity which is convenient for quick repositioning. It was positioned at a distance of 4.98 m from the microphone array.
The capture system consisted of a MacBook Pro 16" running on macOS 10.15.6 with SpectAcoular [80] and Smaart v.8 [81]. SpectAcoular provides a Graphical User Interface (GUI) for Acoular. Smaart v.8 is an audio analysis software which provides a signal generator and evaluation tools. For this thesis only the signal generator was used to create a white noise stimulus.
The two base boards were connected via an Universal Serial Bus (USB) 2.0 to USB-C adaptor to the MacBook. In the Audio-MIDI-Setup of operating system, an "aggregate device" was created that contains both base boards with drift correction disabled. The loudspeaker was connected via Bluetooth. The Smaart output was set to -12 dB and the MacBook output was set to maximum. A block diagram of the measurement setup is shown in Fig. 4.10. The room temperature was at 21.9 °C. This results in an approximated speed of sound of $c = \sqrt{\frac{\kappa RT}{M_{\text{mol}}}} = \sqrt{\frac{1.4 \cdot 8.314\,\text{Nm}/(\text{mol}\cdot\text{K})}{0.029\,\text{kg/mol}}} = 344.13\,\frac{\text{m}}{\text{s}}$ [82]. The humidity stayed constantly at 61 %.

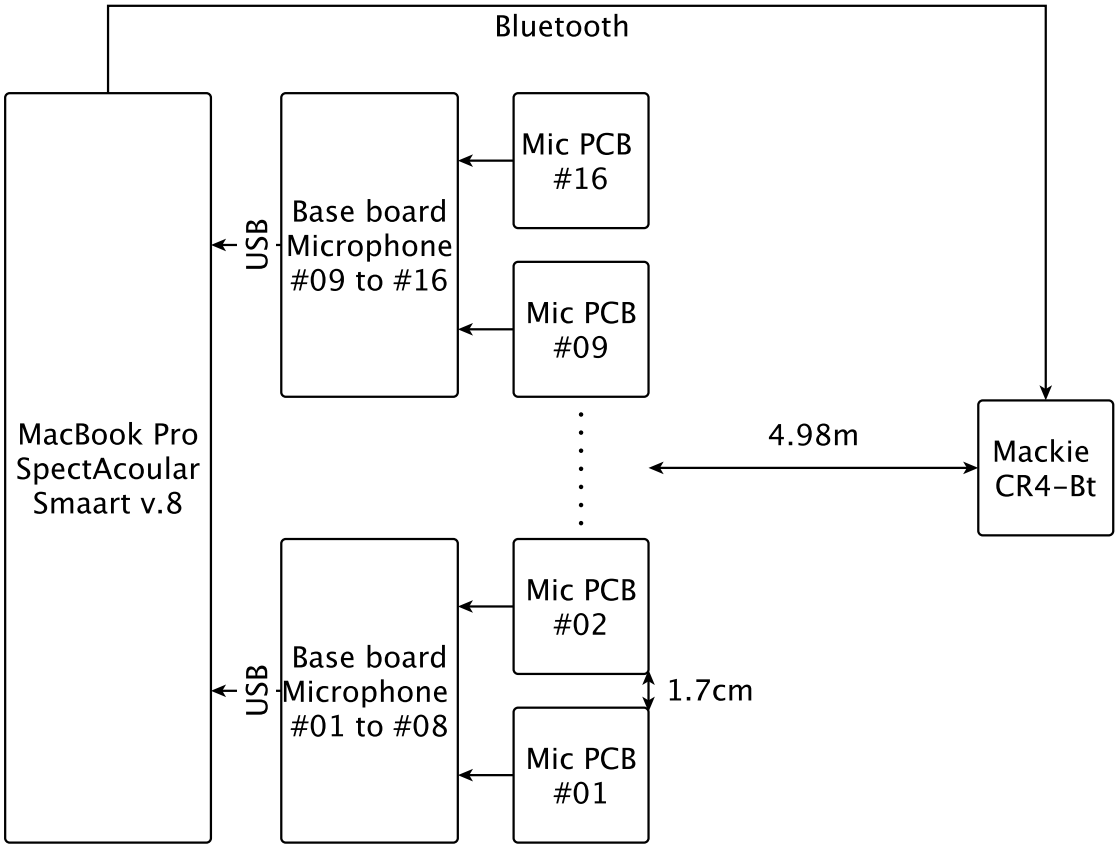
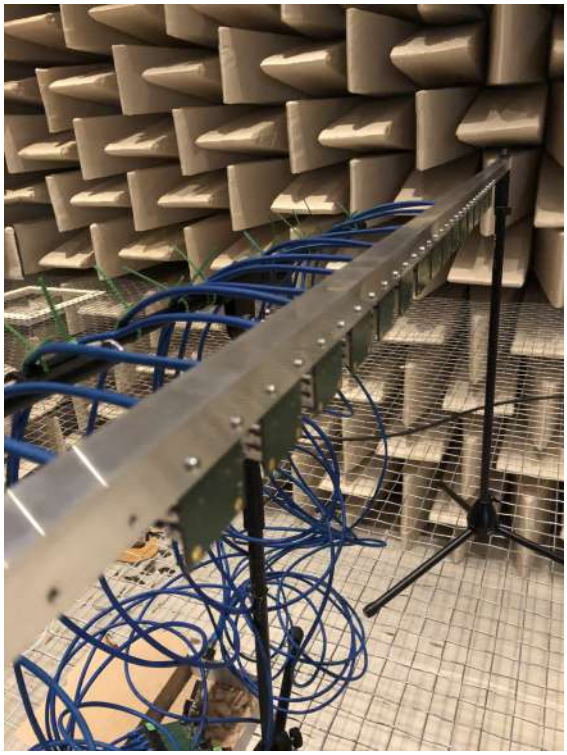
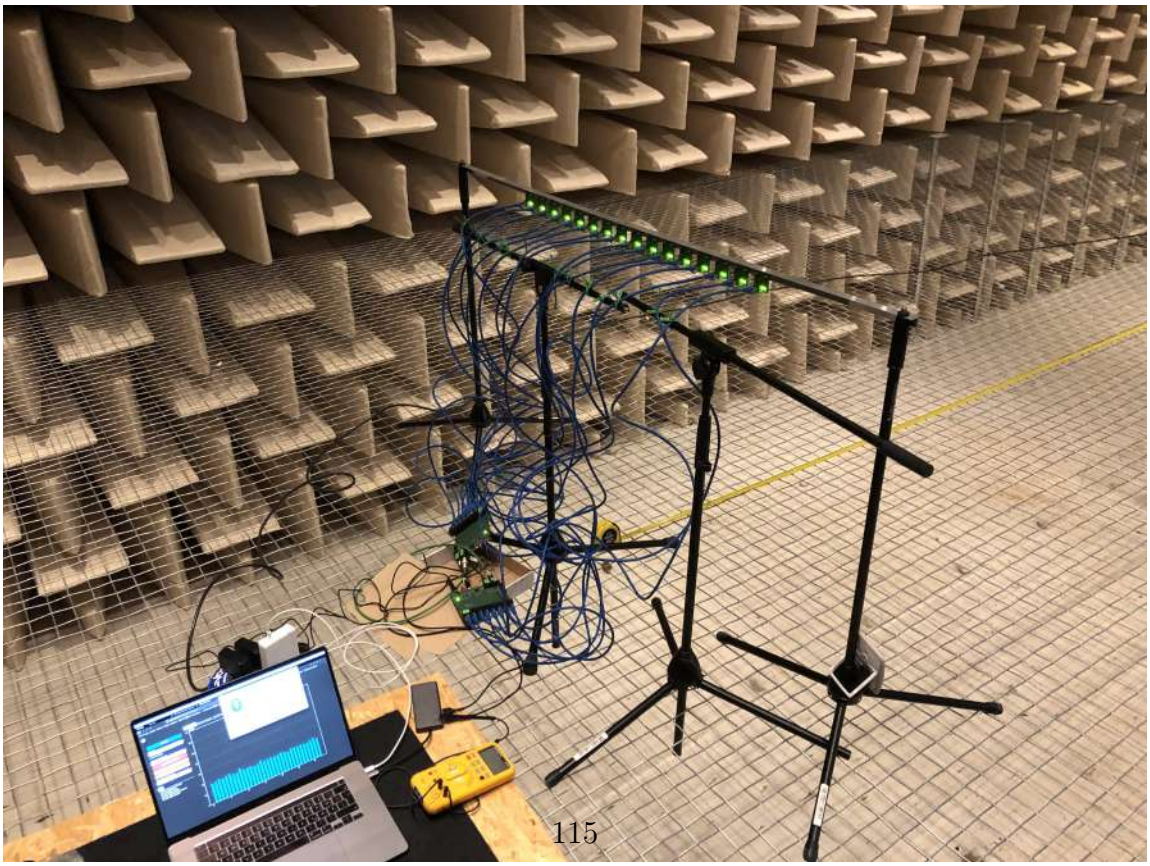


Figure 4.10: Block diagram of the measurement setup

(a) Rear view of the array



(b) Front view of the array



(c) Measurement system

Figure 4.11: Impressions of the measurement

The measurement positions are described in Tab. 4.2 and illustrated in Fig. 4.12. All measurements were performed with white noise as stimulus. Each position was measured over a period of 60 s with three repetitions per clock master. Between each measurement, the base boards' power supply were disconnected and reconnected to track the range of values for non-deterministic phase differences between the clock master and clock slave.

Name	x pos	y pos	Angle
P1	0 m	4.98 m	90°
P2	1.51 m	4.74 m	72.3°
P3	3.79 m	3.23 m	40.5°
P4	4.98 m	0 m	0°

Table 4.2: Measurement positions

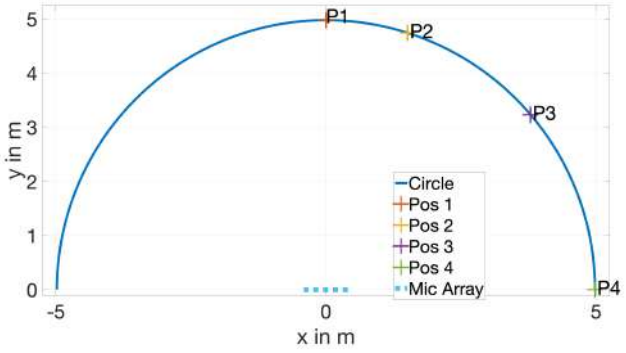


Figure 4.12: Measurement positions

4.2.2 SpectAcoular Integration

SpectAcoular is a GUI for the Acoular framework [80]. It displays the audio data in real-time and provides control and measurement options. It is currently (as of December 2020) in beta status. The system designed in this thesis was integrated into the GUI code base. The changes in the code base are explained in App. E.

4.2.3 Measurements

All measurements were performed in the anechoic chamber of the Department of Engineering Acoustics at the TU Berlin. The evaluation of the measurements was performed by applying a cross-correlation between the microphones. Depending on the angle of incidence, different assumptions can be made.

Oblique incidence of sound

The delays between the microphones increase for an oblique incidence of sound. Due to the decreasing amplitude over distance, it can be expected that the cross-correlation values will also decrease for microphones further away from the sound

source. This behavior can be observed in Fig. 4.13 where the cross-correlation of adjacent microphones for an incidence angle of 40.5° is shown. All pairs have an identical delay of −4 sample, except the pair 17,18 which describes the transition between the two base boards indicating a delay between both. The smaller delay for pair 15,16 is probably caused by the construction of the array.

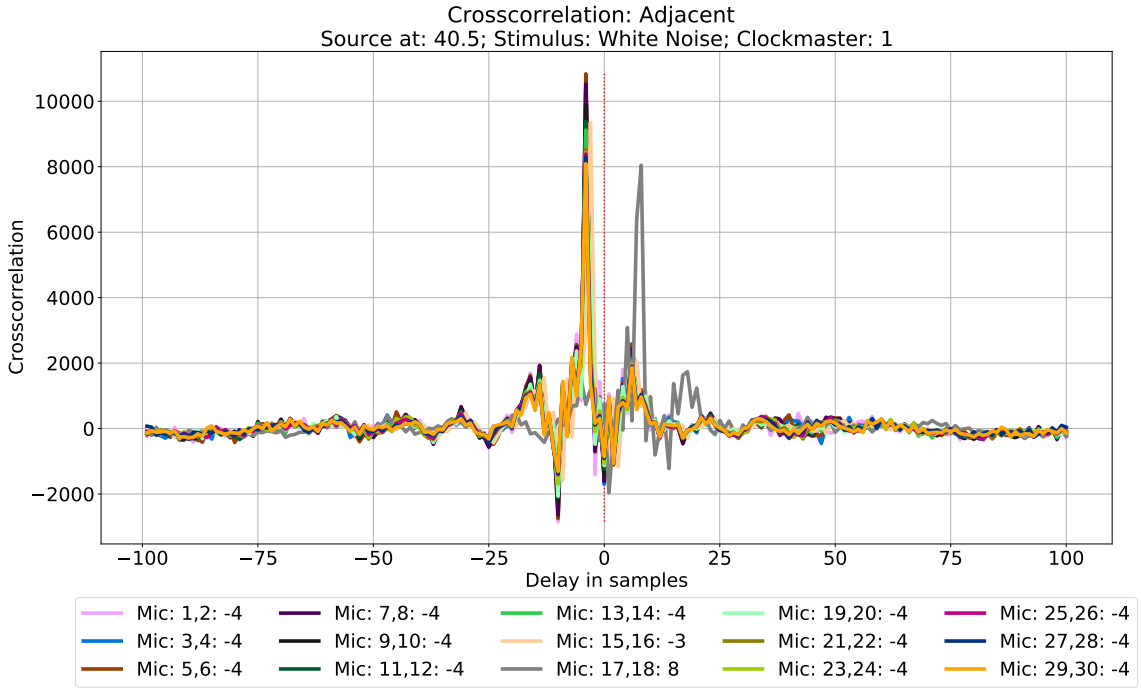


Figure 4.13: Cross-correlation for adjacent microphones at an incidence angle of 40.5°

Straight incidence of sound

Assuming a spheric wave that is emitted from the loudspeaker, the signal arrives after $4.98\,\mathrm{m}/343\,\frac{\mathrm{m}}{\mathrm{s}} = 14.52\,\mathrm{ms}$ at the center microphones and after $4.997\,\mathrm{m}/343\,\frac{\mathrm{m}}{\mathrm{s}} = 14.57\,\mathrm{ms}$ at the outermost microphones. This results in a timing difference of 50 µs. For sampling rate of 48 kHz, this is a difference of approximately 3 sample.
The measurements showed no such sample difference in the cross-correlation of the microphone signals. It can therefore be assumed that the distance between the loudspeaker and the microphones is so large that the sound wave can be approximated not as a spherical wave but as a plane wave.
The processing takes the same time for all microphones and therefore the samples

are expected to arrive at the same time at the host computer.

The normalized and smoothened frequency response of the system is shown in Fig. 4.14. It was generated from the Fourier-transformed measurement data and smoothed with the AKfractOctSmooth function from AKtools [83]. The function applies a 1/n octave smoothing with the following parameters: operation='amp', $fs = 48\,\text{kHz}$ and $band = 3$ which results in a 1/3 octave smoothing.
Smoothing is applied to reduce the high resolution measurement to an envelope. The envelope describes a reduced data set and is therefore easier to interpret. In fractional smoothing, the bandwidth of the smoothing window is a constant percentage of the center frequency [84]. After the smoothing process, the data was normed to a frequency of 1 kHz.

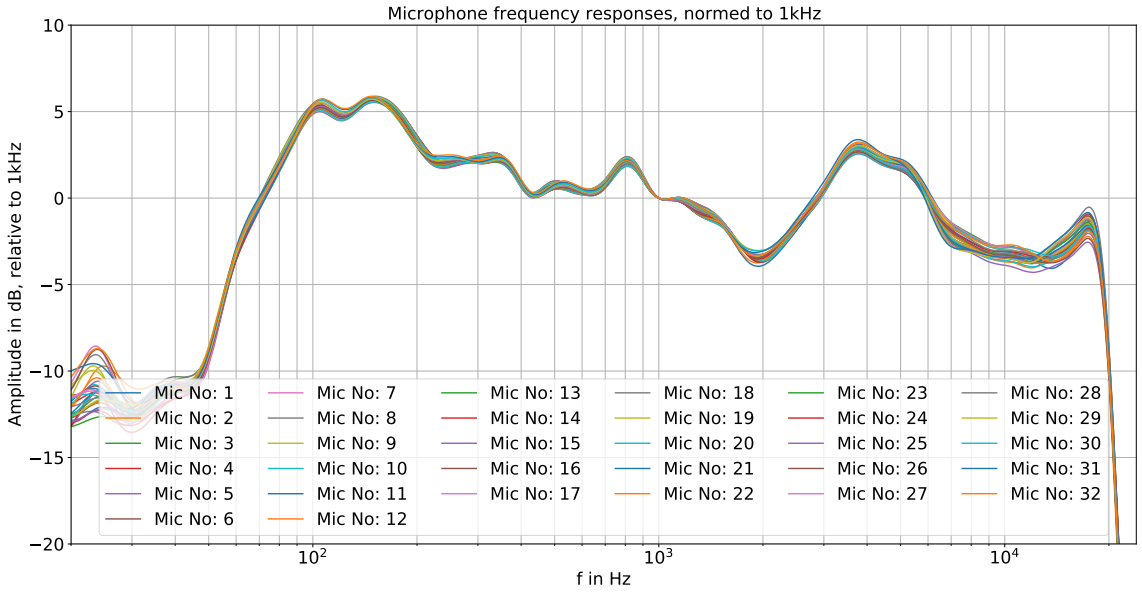


Figure 4.14: Frequency response of the microphones normed at 1 kHz

As it can be seen, the response is rather flat and deviates from the normed point at 1 kHz of roughly ±5 dB. The frequency response of the signal processor is known from Fig. 3.13. It is almost flat (with the typical roll off of a high pass filter at about 19 kHz) and should therefore barely affect the frequency response.
Unfortunately, neither Mackie nor Knowles have published a frequency response for their loudspeaker or microphone, respectively. Therefore the exact causes of the deviations can only be assumed.

According to the data sheet, the −3 dB point of the microphone is at 30 Hz [45], the speaker's at 70 Hz[79]. This can be observed in the diagram where noise can be seen below 30 Hz and a rising edge in the range between 30 Hz and 70 Hz.
The emphasis of the low frequency range (slightly above 100 Hz) is probably caused by the speaker. Mackie emphasizes in the data sheet that the loudspeaker is designed to extend the low frequency range which could explain this behavior.
The break-in and the subsequent boost between 1 kHz and 4 kHz is harder to explain. It could be caused by the construction of the microphone PCBs, the construction of the microphone array or the loudspeaker. Since both manufacturers only specify a ±3 dB deviation in the respective frequency range, it may well be that the combination of the individual components leads to the frequency response shown.
Between the single microphones, the deviation to each other is rather small. Up to 6 kHz it is at roughly ±0.5 dB. Frequencies above 6 kHz show a deviation of approximately ±1 dB. The typical peak in the frequency response of a Micro-Electro-Mechanical Systems (MEMS) microphone at high frequencies can be seen but is much less pronounced.

The delay between the two base boards in the cascaded configuration should be zero. To verify this assumption, the last microphone driven by the clock master and the first microphone driven by the clock slave were examined using cross-correlation. The result is shown in Fig. 4.15. The plot shows each run for each clock master and the calculated delay between the two base boards in samples with respect to the sampling frequency of 48 kHz. The peaks are marked with dots for better visibility. The plot shows that there is a large difference in amplitude between the cascaded configuration and both base boards running on their own, non-synchronized clock. It also shows that there is a non-deterministic delay between the two base boards. This has two reasons:

1. The enumeration process is sequential. As soon as the base boards are switched on, they start to collect samples which are stored in a buffer until they are transferred to the USB host. By interrogating one of the devices earlier than the other, the first device needs to empty its buffer first. This introduces a non-deterministic time delay.

2. As already described in Sec. 4.1, there is no deterministic relationship between the input and the output of the used clock synthesizer. Therefore, the sampling time can vary up to one clock period of the sampling clock. The sampling clock runs at a frequency of 3.072 MHz, the maximum phase shift between the two base boards is 180°. This results in a maximum difference of 162.75 ns or a

spatial shift of 55.8 µm.

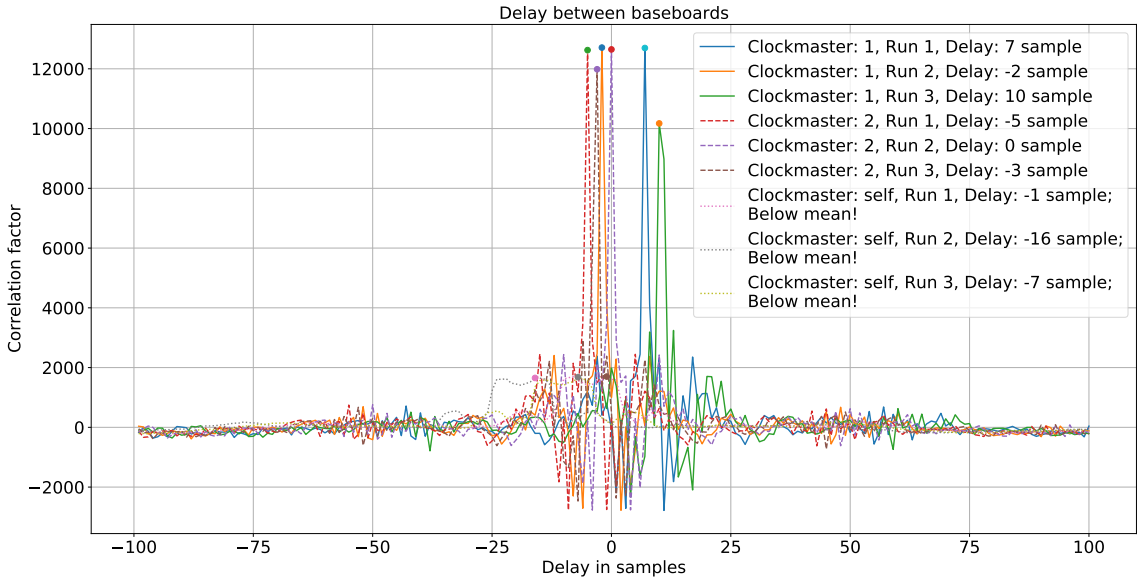


Figure 4.15: Delay in samples between base boards, relative to $f_S = 48\,\text{kHz}$

Three runs per configuration are not suitable to speak of statistical significance of the results, but it does indicate a certain non-deterministic behavior. The runs with the base board with ID 001 acting as clock master show a delay range from -2 sample to 10 sample. Delays with a positive sign indicate that the clock master transmits its samples earlier than the clock slave. A negative value indicates the opposite.
With base board ID 002 as clock master, the delay ranges from -5 sample to -3 sample. One run even made it with zero samples delay.
When both base boards run on their own audio master clock, the delay is barely measurable and it is not clear how meaningful these values are. While the coupled configurations have cross-correlation values between 10000 and 12500, the non-coupled configurations show values slightly above the noise level with cross-correlation values between 1500 and 1800. Therefore, the self-driven configurations are not examined in more detail here. The automatic processing of the measurement data indicates this with a "Below mean" note in the plot.

The beamforming was calculated with a Delay-and-Sum algorithm in the frequency domain. A block size of 128 and a Hanning window were applied for the cross-spectral matrix. A "True Level" steering vector was used [85]. The parameters for the micro-

phone geometry and the focus grid are listed in Tab. 4.3. The plot was created with the "imshow()" function from Python's matplotlib, using a bicubic interpolation.

Parameter	**Description**	**Minimum [m]**	**Maximum [m]**
Microphone geometry	Linear array in x plane	-0.4135	0.4135
Focus grid	Rectangular grid		
	x	-5.5	5.5
	y	-0.25	5.5
	z	0.1	-
	increment	0.01	-

Table 4.3: Used beamformer parameters

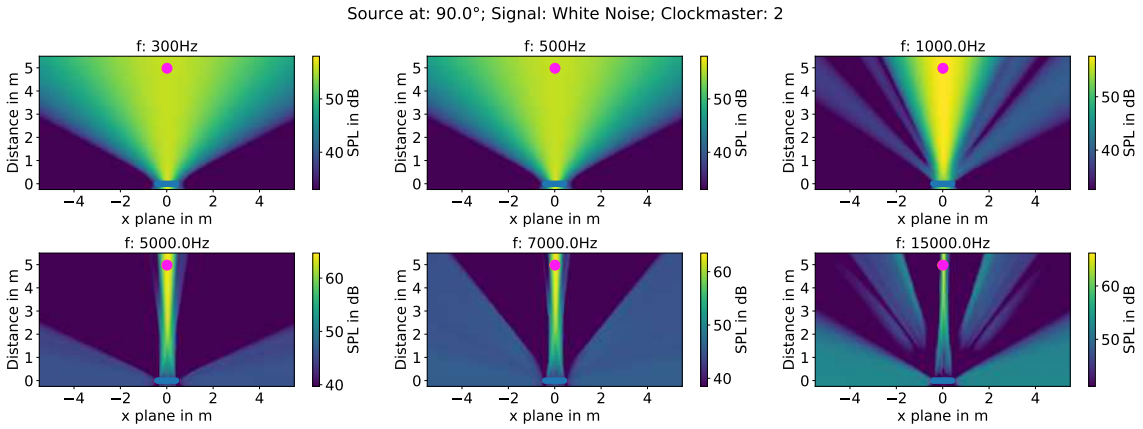
For the estimation of the position of the source, the beamformer data were evaluated at the semicircle describing the 4.98 m distance to the measurement position. Within this data set, the first maximum was searched for and then plotted as a pink dot. The original position is indicated by an orange dot.

In Fig. 4.16 the results of the beamforming are shown.
Fig. 4.16(a) shows the run with base board ID 002 as clock master and a delay of zero samples between both boards.
Fig. 4.16(b) shows the worst case run with base board ID 001 as clock master and ten samples delay between both base boards.
Fig. 4.16(c) shows the simulation results for this measurement setup.

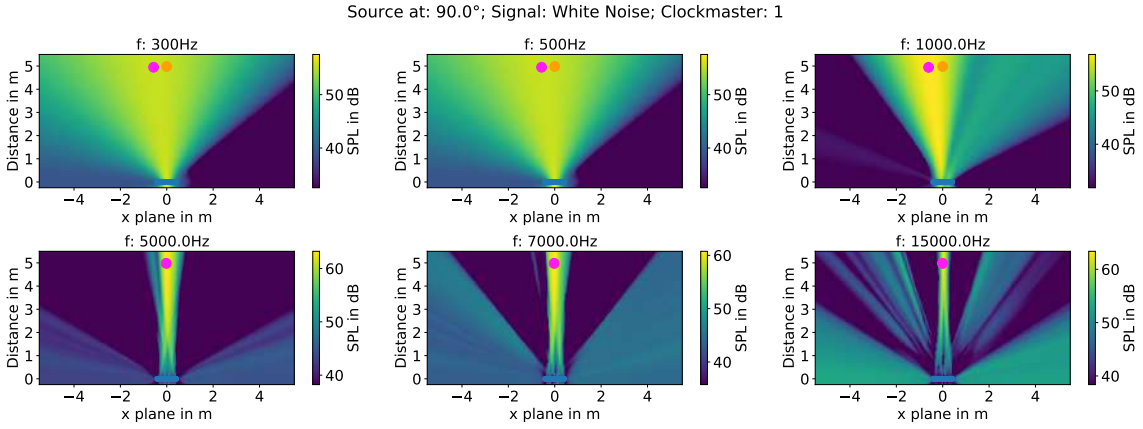
All plots show a clear formation of side lobes from a frequency of 1kHz and above.

Fig. 4.16(a) shows a correct localization of the noise source over all frequencies. For a frequency of 15 kHz, the noise source seems to be slightly off center which could indicate an inaccuracy in positioning the source. The same results can also be observed the simulation in Fig. 4.16(c).

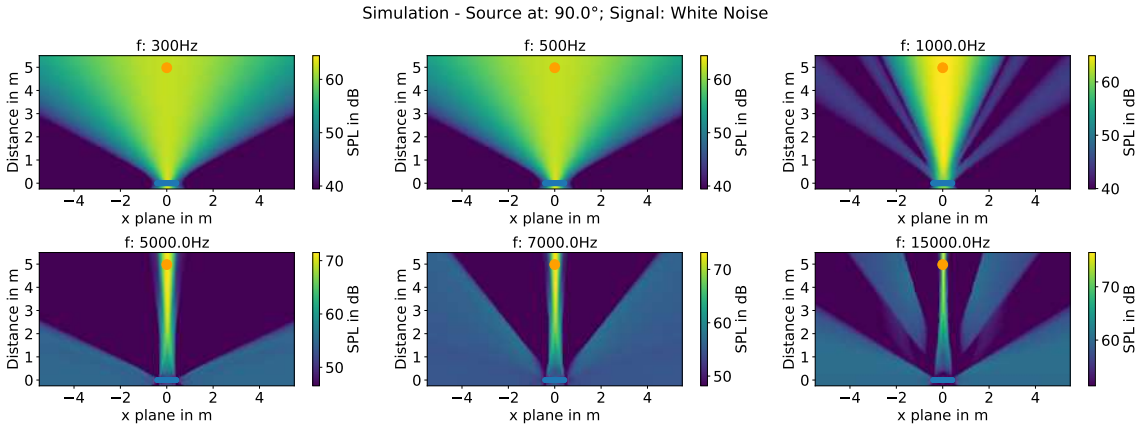
Fig. 4.16(b) shows an incorrect localization of the noise source for lower frequencies which is caused by the delay of ten samples between the base boards. The relationship between the wavelength of the examined frequency in relation to the length of the array is clearly visible. As long as the wavelength is not considerably smaller than half the length of the microphone array, which is preprocessed by one base board, the information of both base boards is relevant. Thus, the error in the



(a) Zero samples delay between base boards



(b) Ten samples delay between base boards



(c) Simulation

Figure 4.16: Beamforming results. Orange dot: actual source position, pink dot: estimated position

location estimation is large. As the frequency increases, the wavelength becomes smaller and so does the location estimation error. The angular difference between the original location and the estimated location is shown in Tab. 4.4.

Frequency	**Wavelength**	**Angular difference**
300 Hz	1.14 m	3.27°
500 Hz	0.686 m	3.27°
1 kHz	0.343 m	3.52°
5 kHz	0.067 m	0.16°
7 kHz	0.049 m	0.16°
15 kHz	0.023 m	0.0°

Table 4.4: Angular difference between the actual and calculated source position for a delay of 10 sample

Chapter 5

Summary and future work

This thesis describes the development, construction, testing and validation of a data acquisition system specialized on Micro-Electro-Mechanical Systems (MEMS) microphone arrays. It provides information on the techniques used in the microphones and the Universal Serial Bus (USB) communication, as well as considerations for the data transfer between microphones and microcontroller.

The system is based on the XMOS XUF216 microcontroller that handles the preprocessing of the microphone data and provides USB communication via the integrated Physical Layer (PHY). XMOS provides a reference design for a small MEMS microphone array which served as a starting point.
The reference design has been extended on the hardware and firmware level in order to be able to preprocess a larger number of microphones, act as clock master or clock slave and detach the microphones from the microcontroller Printed Circuit Board (PCB). Validation of the system was performed by measuring a sound source at various angles of incidence relative to the array.

The developed device is able to preprocess the data of up to 16 MEMS microphones and send this data via USB to a host computer. In order to extend the preprocessing limit of a single microprocessor, the possibility of cascading several microcontrollers was realized. This was achieved by adding additional circuitry for clock generation and clock recovery. The limiting factor of the cascaded system is the available transmission bandwidth on the USB.
The system exceeds the allowed power ratings on the USB. Therefore, an onboard Power Supply Unit (PSU) was designed which provides the required voltages to the base board and the microphone PCBs.

For flexibility, the microphones are separated from the PCB which contains the microcontroller. This results in a base board that contains the clock generation units and data processing unit as well as the PSU.
The separate microphone PCBs contain two microphones each, a data transmission unit, a clock receiving unit and a PSU. The data and clock transmission is differential. This ensures signal integrity and minimizes external influences. The differential traces are impedance matched to the typical value of 100 Ω. The transmission lines between the microphone PCB and the base board use Commercial off-the-shelf (COTS) CAT6 cables. With this system, a maximum cable length of 2.5 m is possible. Cables that are longer than this violate the setup and hold specifications and can therefore lead to undefined behavior.

At the firmware level, the heavily ifdef'd reference design that can be used with all other XMOS reference products, was reduced to the code relevant to this work. It was then extended to handle a total of 16 microphones per microcontroller. This occupies additional cores and requires an upgraded data transfer function between the signal processing tile and the USB tile. The system also identifies whether it is operating as a clock master or clock slave. As clock master, it generates the synchronization clock for the clock slave. As clock slave, it configures the external clock synthesizer accordingly. The power mode has been changed from bus powered to self-powered.

The system used for validation consists of two base boards that can process a total of 32 microphones. It was successfully put into operation.
However, the validation of the functionality of a cascaded system revealed two sources of non-deterministic behavior at the hardware and firmware level:

1. When adding a new device to the USB, it gets interrogated by the host. This enumeration process is performed in a sequential manner. As soon as the base boards are switched on, they start to collect samples which are stored in a buffer until they are transferred to the host. By interrogating one of the devices earlier than the other, the first device already has some samples stored in its buffer that must be transferred first. This introduces a non-deterministic time delay between two base boards.

2. The clock synthesizer has no fixed phase relationship between input and output. In clock slave mode the clock synthesizer is used to generate the 12.288 MHz audio master clock from the incoming 48 kHz signal from the clock master. This leads to non-deterministic behavior, since the sample time between clock

master and clock slave can now differ by up to one period of the sample clock. The sampling frequency of the microphones is 3.072 MHz. A phase shift of 180° leads to a maximum inaccuracy of 162.75 ns. This corresponds to a spatial inaccuracy of $\Delta x = c \cdot \Delta t = 343 \frac{\mathrm{m}}{\mathrm{s}} \cdot 162.75\,\mathrm{ns} = 55.82\,\mu\mathrm{m}$. Such a small value is negligible, because the microphones cannot be positioned so precisely.

This sections answers the research questions from Chap. 1

- Does the modular system provide sufficiently accurate data for post-processing?
- Is it feasible to physically detach the microphones from the Digital Signal Processor (DSP)?
- Is the selected transmission protocol suitable for this task?

as follows:

As described above, the system has a non-deterministic behavior that leads to inaccuracies in the audio sample time when multiple base boards are cascaded. Therefore, the data provided by a cascaded system is not accurate enough for the usual microphone array applications such as beamforming or arrival direction estimation.
However, a single base board will provide consistent data that is accurate enough, but it will reduce the number of microphones that can be used by half.

It is possible to physically separate the microphones from the DSP. The maximum distance is limited by the sampling rate of the system. For a sampling rate of 48 kHz, the maximum distance is 2.5 m. Increasing the sampling rate would decrease the maximum cable length. Furthermore, it is important to transmit the data from and to the microphones in a differential manner. This ensures a reliable data transmission.

USB was selected as the transmission protocol because the used microcontroller provides an integrated PHY and a reference implementation. It was shown that a cascaded system can be taken into operation on a single USB. However, the maximum cable length specified for USB is at approximately 5 m which can limit the range of action the between base boards and a host computer.

Future implementations could consider the following proposals:

- The available number of cores determines the processing capabilities of a base board. Sixteen MEMS microphones occupy six cores on an XMOS microcon-

troller for the conversion of Pulse-Density Modulation (PDM) signals to Pulse-Code Modulation (PCM) signals. The XUF216 was selected for its availability and price. However, there is one model (XUF232) which offers more cores and thus more computing power. This allows to increase the number of microphones per base board. Unfortunately, this model was not available at the beginning of the design process and was therefore not selected.

- The cable length can be increased by adding an additional D-Flipflop in the data path on the microphone PCB. The D-Flipflop is clocked by the microphone clock and holds the information for one period of it. This ensures that the data line is set to the correct value for one microphone clock period and therefore ensures the correct data sampling at the microcontroller.

- By using an externally driven port, it should be possible to run two microphones per data line and read the incoming data with a custom module before passing it to the decimation library. This would cut the required input ports in half.

- In clock master mode, the XMOS microcontroller generates a clock signal from which the audio master clock is derived. The reference design implements the generation of this clock in software rather than utilizing a clock block. Future implementations can improve clock accuracy by using a clock block for this task. It can be configured at the start of the processor with a combinable task. This reduces the number of cores occupied.
  A further increase in precision can be achieved by using the high accuracy mode in the clock synthesizers.

- The clock synthesizer used introduces a phase incoherency between its input and output. It can be replaced by a Phase-Locked Loop (PLL) that provides a control option for the phase between input and output (e.g. 74HC4046).

- In order to synchronize the sampling process on multiple base boards in a clock master-slave configuration, each XMOS microcontroller should start sampling the incoming PDM data as soon as it can detect a Start of Frame (SOF) packet on the USB. The microcontroller then starts sampling and buffers six samples per channel. With the next SOF, the buffer is flushed to the USB while new samples are collected. When a new device is added, it also starts sampling after the detection of the first SOF and flushes its buffer subsequently. Each buffering period collects $16\,\text{microphones} \cdot 6\,\text{sample} \cdot 24\,\text{bit} = 288\,\text{Byte}$. This amount of data can be transferred within one transfer period. By linking the

start of data sampling to the SOF which is broadcast to all devices on the bus, coherent data sampling and transmission is ensured.

- To increase flexibility, it might be worth considering other data transfer protocols. For example, network based transmission protocols for real-time audio applications like Audio over IP (AoIP) or Time Sensitive Networks (TSN).
  An AoIP protocol that is currently experiencing increasing popularity and adoption is AES67 [86]. AES67 is an open source layer 3 protocol suite.
  A TSN solution is Audio Video Bridging (AVB) which is specified in IEEE 802.1BA [87]. AVB operates on layer 2 and has some major differences in the data transmission compared to AES67. Both standards have in common that they use the Precision Timing Protocol (PTP)v2 for their clock generation and synthesis. PTP allows for timing precision in the sub-nanosecond range.
  A system based on AoIP or TSN leads to a different implementation of the system. Smaller microprocessors preprocess a smaller number of microphones and then send the data to a host computer via a suitable medium (either Ethernet or WLAN). This also solves the current limitation on cable length.
  AVB reference designs are available from XMOS[88], modular AVB hardware is for example available from [89].

- The feasibility of WLAN transmissions in the context of AoIP or TSN should be examined.

Bibliography

[1] Yves Grenier. A microphone array for car environments. *Speech Communication*, 12(1):25 – 39, 1993.

[2] Ivan Marković and Ivan Petrović. Speaker localization and tracking with a microphone array on a mobile robot using von mises distribution and particle filtering. *Robotics and Autonomous Systems*, 58(11):1185 – 1196, 2010.

[3] D. Pavlidi, A. Griffin, M. Puigt, and A. Mouchtaris. Real-time multiple sound source localization and counting using a circular microphone array. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(10):2193–2206, 2013.

[4] Carsten Spehr, Daniel Ernst, and Thomas Ahlefeldt. Cabin noise measurements with microphone arrays and sound intensity probes. In *Inter-Noise 2019*, number 1605 in Conference Proceedings online, pages 1–10, Juni 2019. I-INCE Classification of Subject Number: 72.

[5] Merino-Martinez et al. A review of acoustic imaging methods using phased microphone arrays. *CEAS Aeronautical Journal*, 03 2019.

[6] Andrew McPherson. Bela: An embedded platform for low-latency feedback control of sound. *The Journal of the Acoustical Society of America*, 141:3618–3618, 05 2017.

[7] UMA16 USB Mic array, 2017. `https://www.minidsp.com/products/usb-audio-interface/uma-16-microphone-array`, Retrieved on 20.11.2020.

[8] ReSpeaker Core v2.0, 2018. `http://wiki.seeedstudio.com/ReSpeaker_Core_v2.0/`, Retrieved on 20.11.2020.

[9] CAE Software und Systems GmbH, 2020. `https://www.cae-systems.de/produkte.html`, Retrieved on 20.11.2020.

[10] R. Scheibler, J. Azcarreta, R. Beuchat, and C. Ferry. Pyramic: Full stack open microphone array architecture and dataset. In *2018 16th International Workshop on Acoustic Signal Enhancement (IWAENC)*, pages 226–230, 2018.

[11] Florian Perrodin, Janosch Nikolic, Joel Busset, and Roland Siegwart. Design and calibration of large microphone arrays for robotic applications. *IEEE International Conference on Intelligent Robots and Systems*, pages 4596–4601, 2012.

[12] Eugene Weinstein, Kenneth Steele, Anant Agarwal, and James Glass. LOUD : A 1020-Node Modular Microphone Array and Beamformer for Intelligent Computing Spaces. *Glass*, 2007.

[13] Knowles. FREQUENCY RESPONSE AND LATENCY OF MEMS MICROPHONES: THEORY AND PRACTICE, 2017.

[14] STMicroelectronics. Tutorial for MEMS microphones, feb 2017.

[15] M. A. Shah, Ibrar Ali Shah, Duck-Gyu Lee, and S. Hur. Design approaches of mems microphones for enhanced performance. *J. Sensors*, 2019:9294528:1–9294528:26, 2019. Licensed under CC BY 4.0 `https://creativecommons.org/licenses/by/4.0/`.

[16] D.T. Blackstock. *Fundamentals of Physical Acoustics*. A Wiley-Interscience publication. Wiley, 2000.

[17] John Eargle. *The Microphone Book*. Taylor & Francis, third edition, 2011.

[18] M. Ortmanns and F. Gerfers. *Continuous-Time Sigma-Delta A/D Conversion*. Springer Berlin Heidelberg New York, 2006.

[19] Richard Schreier. Delta Sigma Toolbox, 2020. `https://www.mathworks.com/matlabcentral/fileexchange/19-delta-sigma-toolbox`, MATLAB Central File Exchange. Retrieved December 11, 2020.

[20] Eugene B. Hogenhauer. An economical class of digital filters for decimation and interpolation. *IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING*, ASSP-29(2):155–1622, 1981.

[21] Texas Instruments. LVDS Application and Data Handbook. *Texas Instruments Incorporated*, 2002. SLLD009.

[22] Telecommunications Industry Association/ElectronicIndustries Association. *TIA/EIA-644-A*, 1996.

[23] D.E. White. Chapter 6 - external set-up and hold times. In D.E. White, editor, *Logic Design for Array-Based Circuits*, pages 189 – 207. Academic Press, Boston, 1992.

[24] miniDSP. MCHStreamer Kit. `https://www.minidsp.com/products/usb-audio-interface/mchstreamer`, Retrieved on 02.12.2020.

[25] Lattice. iCE40 UltraPlus 8:1 Mic Aggregation Demo, 2018. FPGA-UG-02035-1.2.

[26] Vadim Kotelnikov. CIC-filter core, 2011. `https://opencores.org/projects/cic_core`, Retrieved on 02.12.2020.

[27] Rudolf Usselmann. USB 2.0 Function Core. `https://opencores.org/projects/usb`, Retrieved on 02.12.2020.

[28] ST Microelectronics. PDM audio software decoding on STM32 microcontrollers. AN3998, Doc ID 022391 Rev 1.

[29] Cypress. CE219431 - PSoC 6 MCU PDM-to-PCM Example. Document No. 002-19431 Rev.*A.

[30] Analog Devices. *ADSP-21489 Datasheet*, 2020. Rev. H.

[31] XMOS. *xCORE Microphone Array Hardware Manual*, 2018. Document Number: XM009730A.

[32] Andreas Woehrer. 16 Channel USB 2.0 Sound Card for Digital MEMS Microphones. Master's thesis, Graz University of Technology, 2018.

[33] Dominic Létourneau Vincent-Philippe Rhéaume and Cédric Godin. 16SoundsUSB - 16 Synchronized Inputs USB (UAC2) Sound Card. `https://github.com/introlab/16SoundsUSB`, Retrieved on 02.12.2020.

[34] XMOS. XCORE-200 SERIES. Document No: 082020-1.

[35] XMOS. *XUF216-512-TQ128 Datasheet*, 2018. Document Number: X006990.

[36] XMOS. *XMOS Programming Guide*. Document Number: XM004440A.

[37] S.P. Harbison and G.L. Steele. *C, a Reference Manual*. Prentice-Hall, 2002.

[38] B. Stroustrup. *The C++ Programming Language: The C++ Programm Lang_p4*. Pearson Education, 2013.

[39] Eclipse Foundation. Eclipse IDE. `https://www.eclipse.org/`, Retrieved on 02.12.2020.

[40] XMOS. *Getting Started with the xTIMEcomposer Studio*, 2016. XM008204.

[41] USB Implementers Forum, Inc. *Universal Serial Bus Specification*, apr 2000.

[42] USB Implementers Forum, Inc. *Universal Serial Bus Device Class Definition for Audio Devices*, may 2006.

[43] USB Implementers Forum, Inc. *UNIVERSAL SERIAL BUS DEVICE CLASS DEFINITION FOR AUDIO DATA FORMATS*, sep 2016.

[44] USB Implementers Forum, Inc. *Universal Serial Bus Power Delivery Specification*, 2020.

[45] Knowles. *SPH0690LM4H-1 Everest Datasheet*, 2019. Rev. A.

[46] Maxim Integrated. *MAX9110/MAX9112 Datasheet*, 2019. Rev. 1 9/19.

[47] Maxim Integrated. *MAX9111/MAX9113 Datasheet*, 2019. Rev. 7 6/19.

[48] Micrel. *PL133-37 Datasheet*, 2015. Rev 02/24/15.

[49] Texas Instruments. *TPS7A26 Datasheet*, 2018. SBVS290B–DECEMBER 2018–REVISED OCTOBER 2019.

[50] M. Möser. *Messtechnik der Akustik*. Springer Berlin Heidelberg, 2010.

[51] Robert P. Dougherty, Rakesh C. Ramachandran, and Ganesh Raman. Deconvolution of sources in aeroacoustic images from phased microphone arrays using linear programming. *International Journal of Aeroacoustics*, 12(7-8):699–717, 2013.

[52] ST Microelectronics. *ST1S06AP Datasheet*, 2009. Doc ID 12236 Rev 9.

[53] ST Microelectronics. *STM1061 Datasheet*, 2006. Rev 4.

[54] Maxim Integrated. *MAX6895 Datasheet*, 2007. Rev 14.

[55] Nexperia. *74LVC1G11 Single 3-input AND gate*, jul 2017. Rev. 10.

[56] Cirrus Logic. *CS2100-CP Datasheet*, 2015. DS840F3.

[57] ABRACON. *ASFL1 Datasheet*, 2010. Revised: 04.13.11.

[58] Texas Instruments. *LMK00804B-Q1 Datasheet*, 2019. SNAS784B - MARCH 2019 - REVISED AUGUST2019.

[59] Nexperia. *74AHC1G79GW,125 Datasheet*, 1999. 74AHC AHCT1G79 v.6.

[60] Pericom. *PI6C5921512 Datasheet*, 2018. Document Number DS40090 Rev 1-2.

[61] Texas Instruments. *SN74LVC2G34 Dual Buffer Gate*, 2001. Revised October 2015.

[62] Texas Instruments. *SN65LVDT386DGG Datasheet*, 1999. SLLS394I - SEPTEMBER 1999-REVISED DECEMBER 2014.

[63] Texas Instruments. *TPD2E001 Datasheet*, 2006. SLLS684I - JULY2006 - REVISED MARCH 2016.

[64] Beta Layout ML4 structure. `https://de.beta-layout.com/images/spec/multilayer_4_lagig_B-01.jpg`, Retrieved on 02.09.2020, Courtesy of beta LAYOUT GmbH.

[65] IPC. *IPC-2251 Design Guide for the Packaging of High Speed Electronic Circuits*, 2003.

[66] XMOS. I/O timings for xCORE200, 2017. XM010258A.

[67] H.W. Johnson and M. Graham. *High-speed Digital Design: A Handbook of Black Magic*. Prentice Hall Modern Semiconductor Design. Prentice Hall, 1993.

[68] XMOS. USB Audio Design Guide, 2016. XM0088546.1.

[69] XMOS. Microphone array library 3.0.1, 2017. XM010267.

[70] XMOS. Debug printing library, 2015. XM006383.

[71] XMOS. DFU loader for XMOS USB AUDIO devices, 2014. XM000524A.

[72] XMOS. I2C Master (Using Single Multibit Port) Component, 2014. XM002026A.

[73] XMOS. USB Library, 2015. XM006387.

[74] USB Implementers Forum, Inc. *Universal Serial Bus Device Class Specification forDevice Firmware Upgrade*, aug 2004. Version 1.1.

[75] Rohde & Schwarz. *R&S RTO Digital Oscilloscope User Manual*, 2017. 1332.9725.02 - 11.

[76] Ennes Sarradj and Gert Herold. Acoular–Open-Source-Software zur Anwendung von Mikrofonarrayverfahren. In *Fortschritte der Akustik, 42. Deutsche Jahrestagung für Akustik, 14.-17. März 2016 in Aachen*. DAGA, 2016.

[77] Steinberg Media Technologies GmbH. ASIO4ALL Website. `http://www.asio4all.org/`, Retrieved on 02.12.2020.

[78] Paul Davis and Stéphane Letz. How can I use multiple soundcards with JACK? `https://jackaudio.org/faq/multiple_devices.html`, Retrieved on 02.12.2020.

[79] Mackie. *CR4BT, CR5BT Manual*, sep 2015. Part No. SW1135 Rev. A 09/15.

[80] SpectAcoular! An extension for Acoular. `https://anaconda.org/acoular/spectacoular`, Retrieved on 02.12.2020.

[81] Rational Acoustics. Smaart v.8. `https://www.rationalacoustics.com/smaart/smaart-v8/`, Retrieved on 20.11.2020.

[82] Stefan Weinzierl. *Handbuch der Audiotechnik*. Springer-Verlag Berlin Heidelberg, 2008.

[83] Fabian Brinkmann and Stefan Weinzierl. AKtools—An Open Software Toolbox for Signal Acquisition, Processing, and Inspection in Acoustics. In *Audio Engineering Society Convention 142*, May 2017.

[84] Joseph G. Tylka, Braxton B. Boren, and Edgar Y. Choueiri. A generalized method for fractional-octave smoothing of transfer functions that preserves log-frequency symmetry. *J. Audio Eng. Soc*, 65(3):239–245, 2017.

[85] Ennes Sarradj. Three-dimensional acoustic source mapping with different beamforming steering vector formulations. *Advances in Acoustics and Vibration*, 2012, 06 2012.

[86] Audio Engineering Society, Inc. *AES67-2018*, apr 2018.

[87] Institute of Electrical and Electronics Engineers. *IEEE 802.1BA-2011*, 2011.

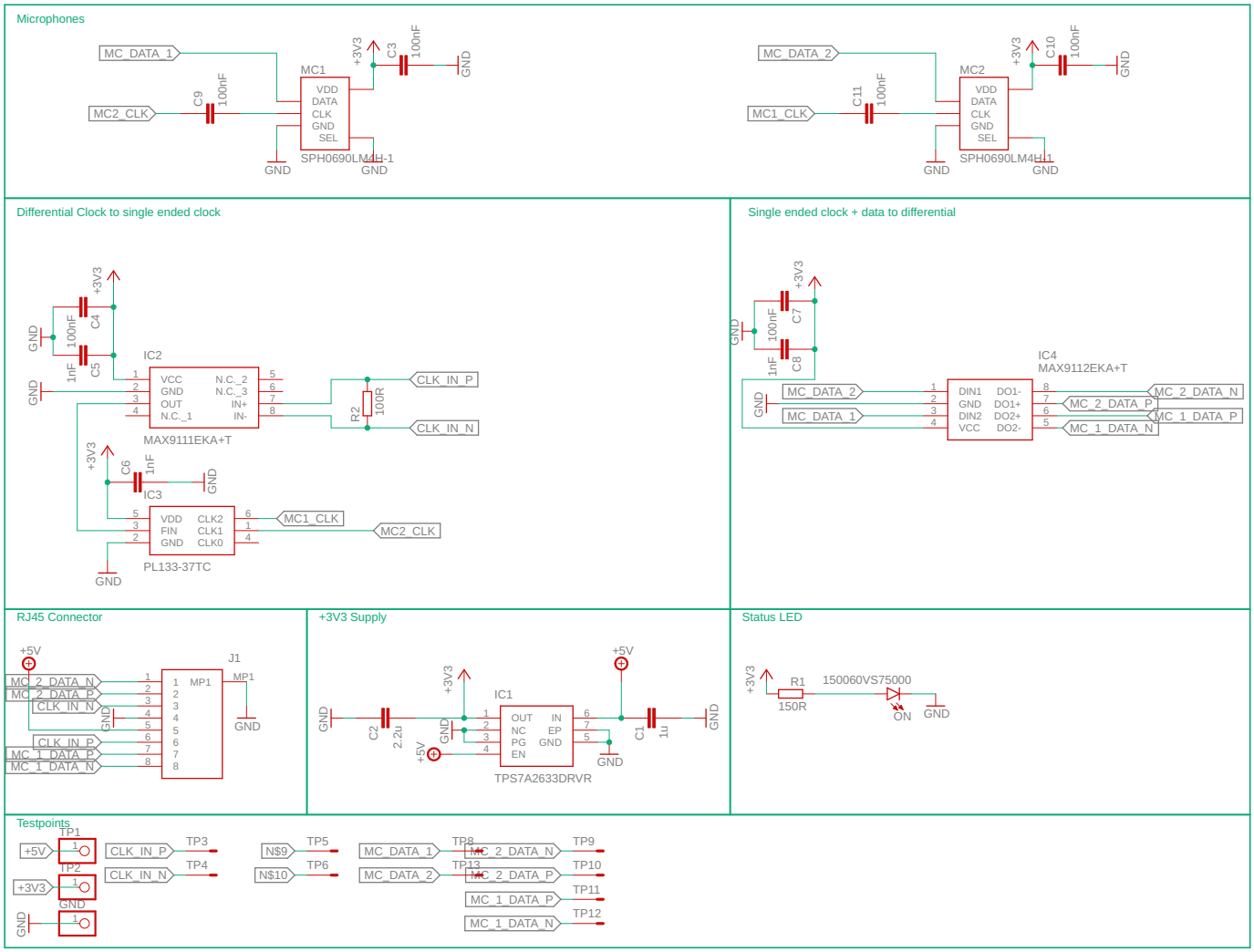
[88] XMOS. AVB Software Stack, nov 2020. `https://github.com/xcore/sw_avb`, Retrieved on 20.11.2020.

[89] Fabian Braun. AVB Audio networking made accessible, nov 2020. `https://www.joyned.online/`, Retrieved on 20.11.2020.

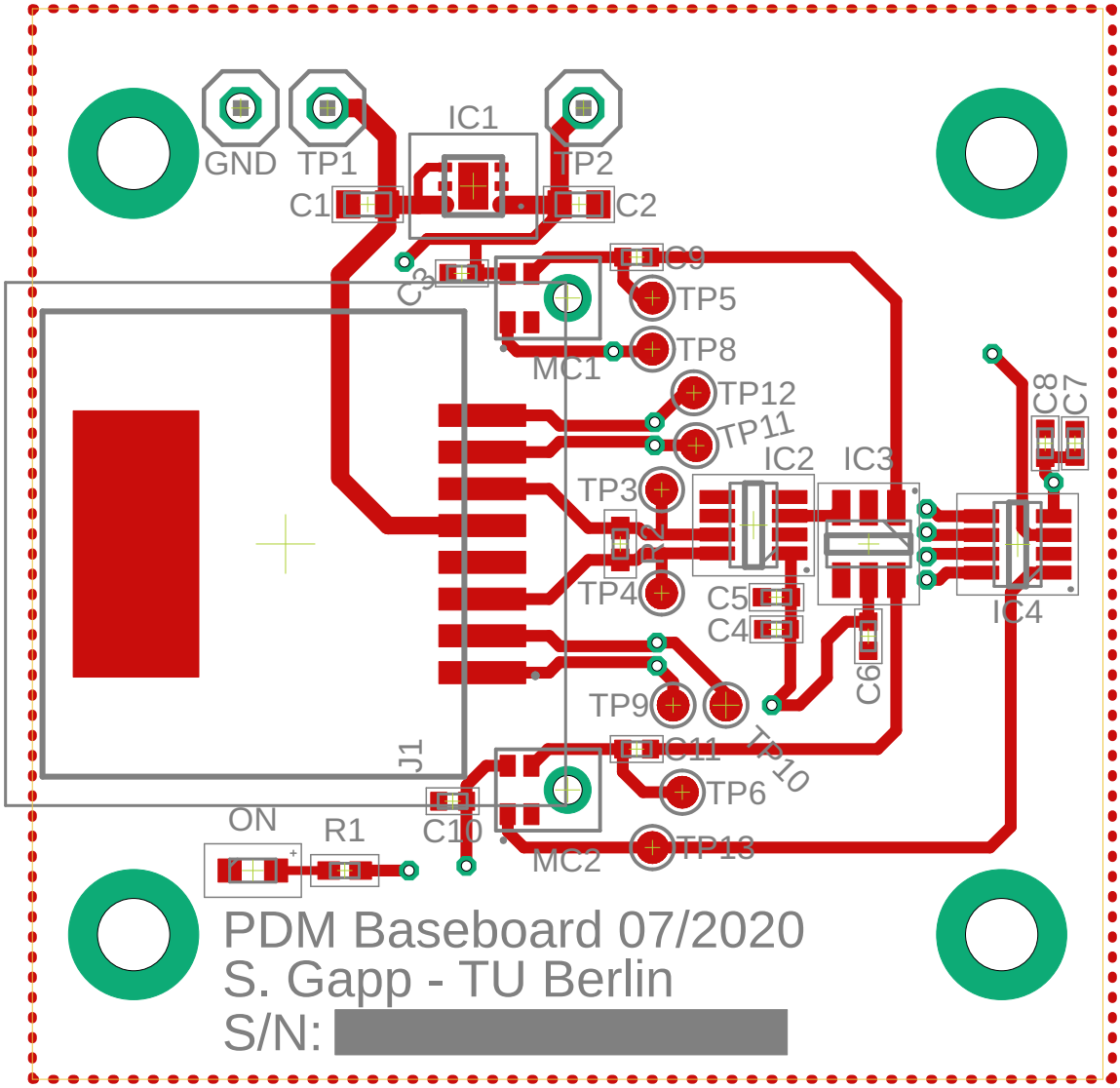
Chapter A

PCB Files

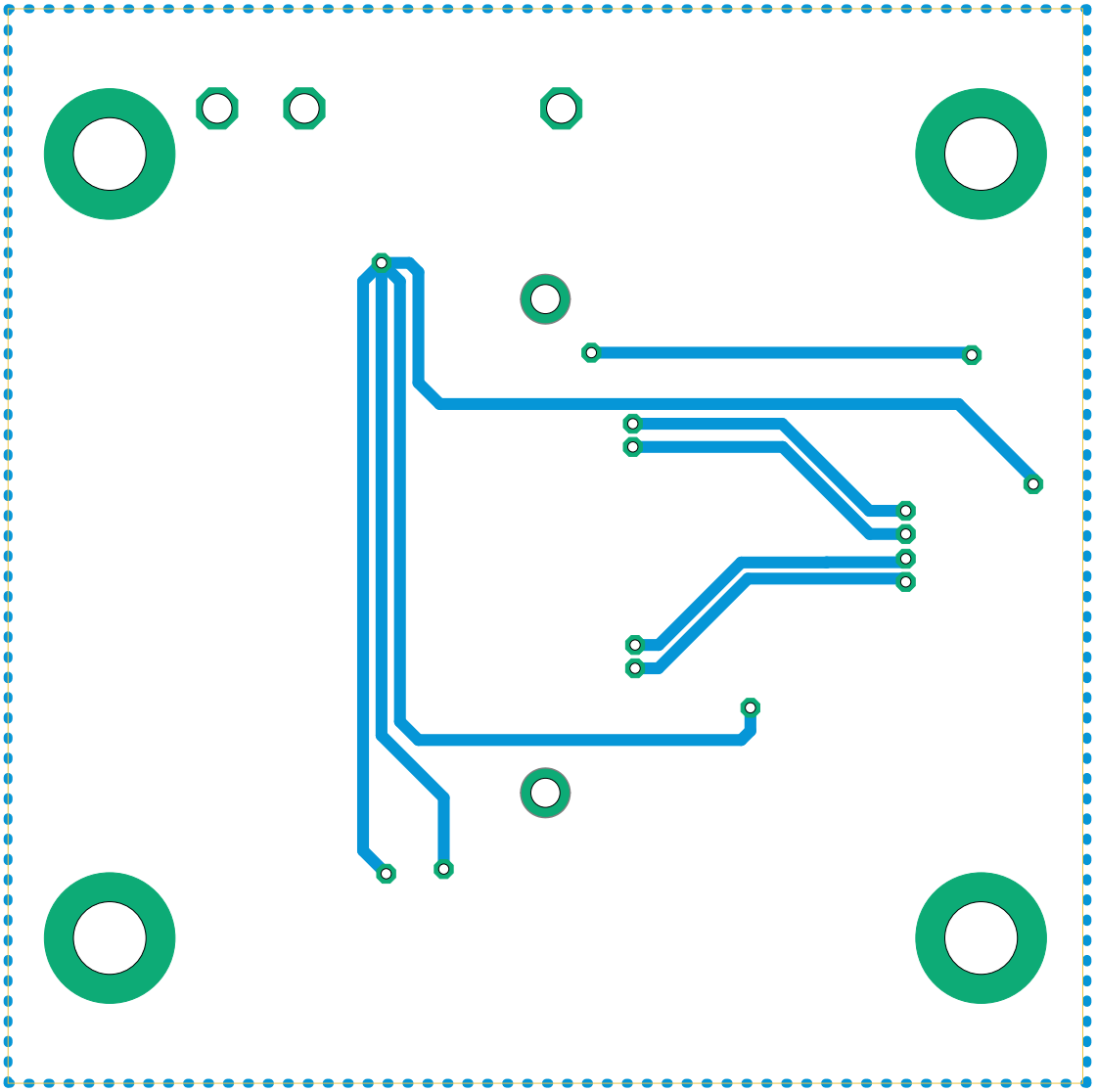
A.1 Microphone PCB schematics



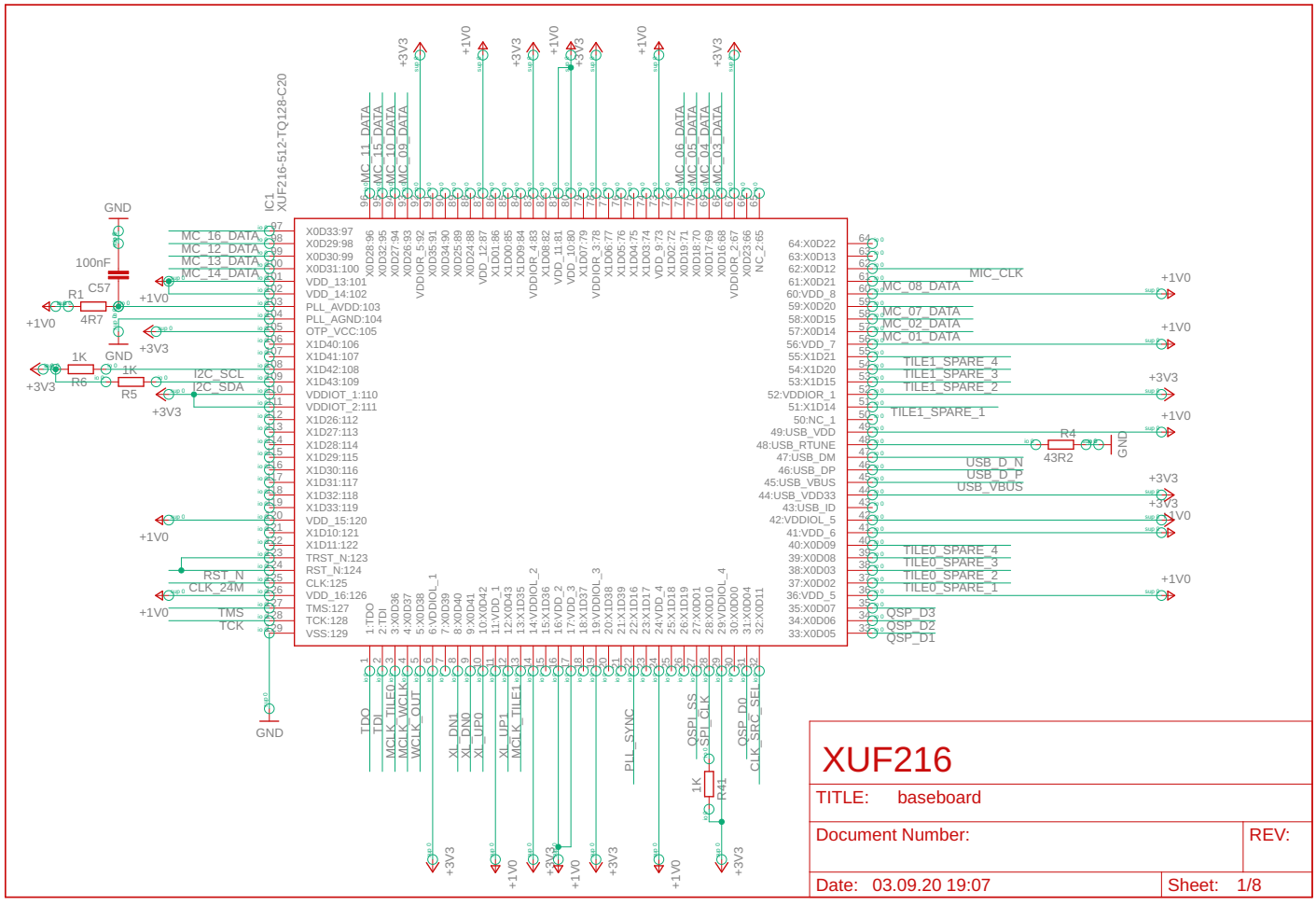
A.2 Microphone PCB Layout Top

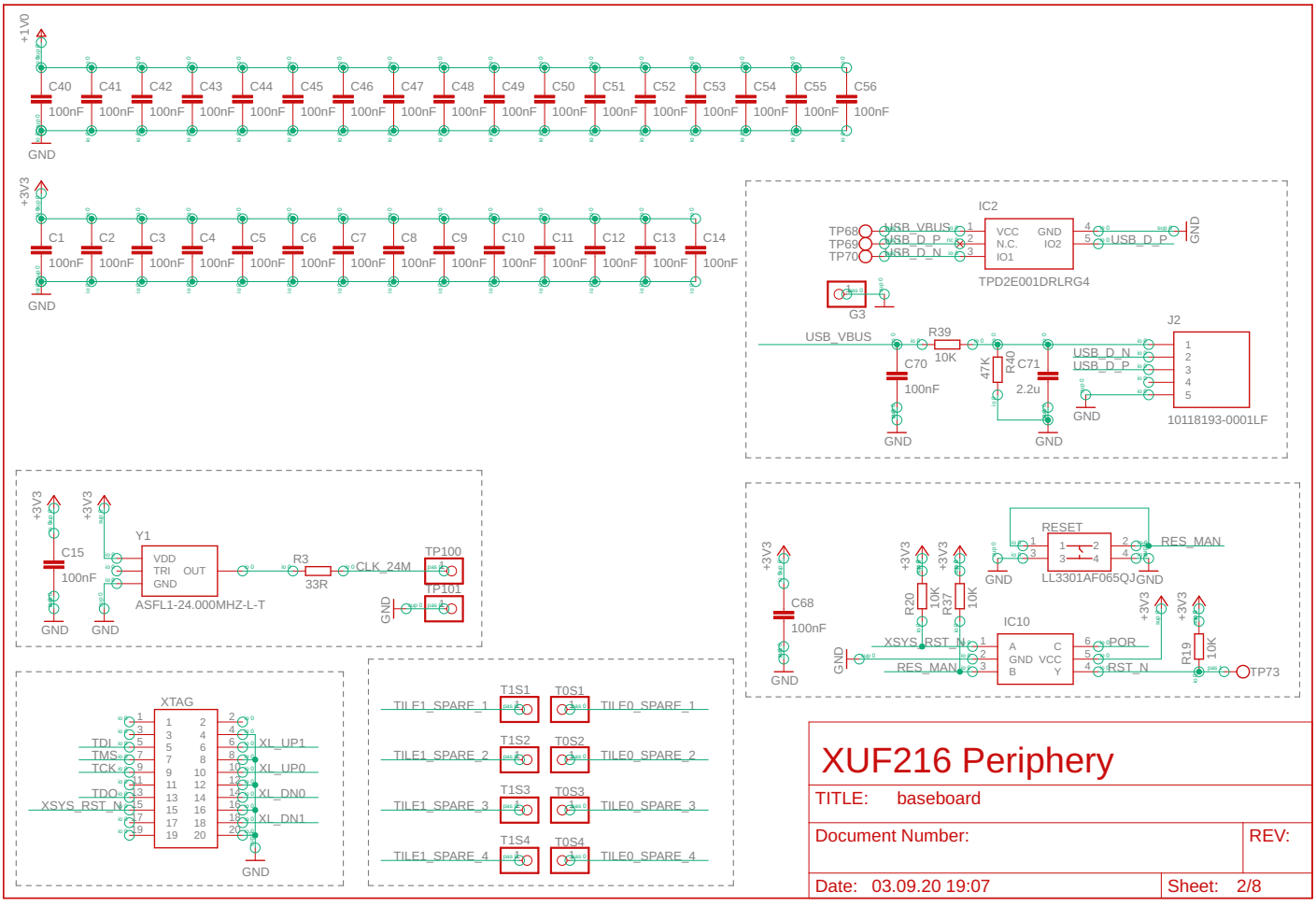


A.3 Microphone PCB Layout Bottom



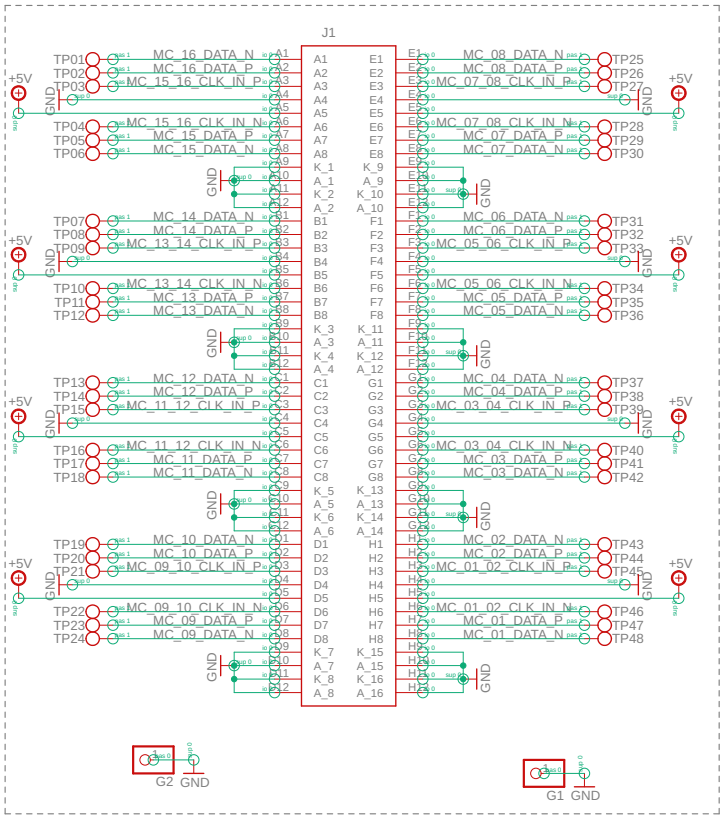
A.4 Base board schematics





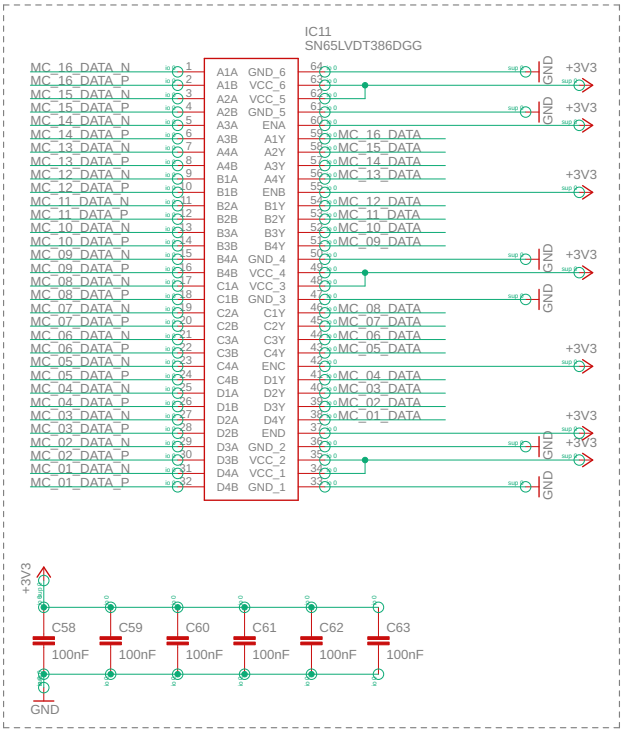
XUF216 Periphery

TITLE: baseboard	
Document Number:	REV:
Date: 03.09.20 19:07	Sheet: 2/8



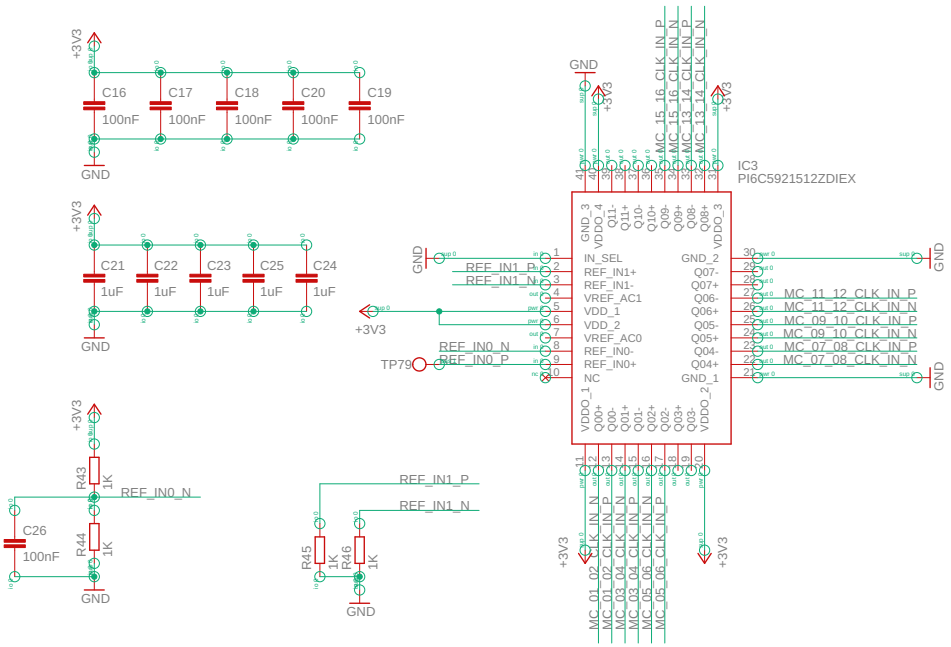
RJ45 Connector

TITLE: baseboard		
Document Number:		REV:
Date: 03.09.20 19:07	Sheet: 3/8	



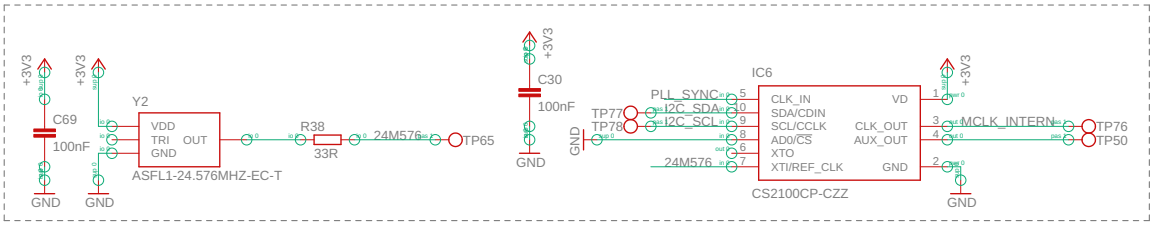
Differential to single ended data

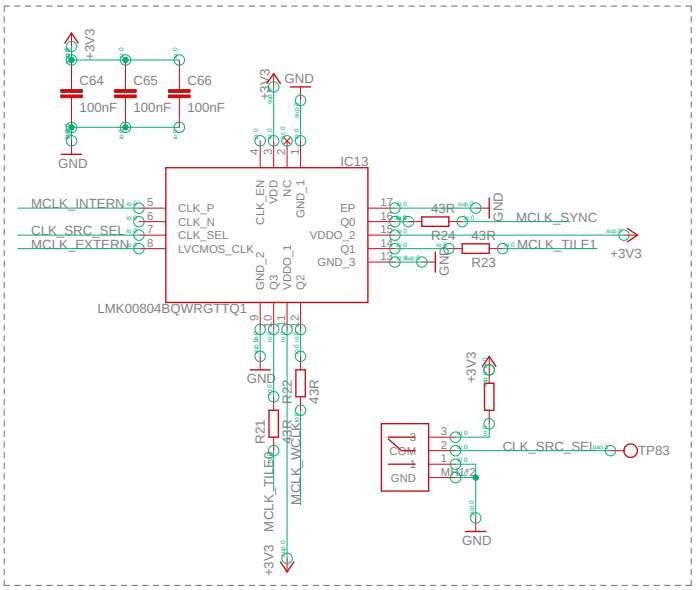
TITLE: baseboard	
Document Number:	REV:
Date: 03.09.20 19:07	Sheet: 4/8



Single ended to differential clock

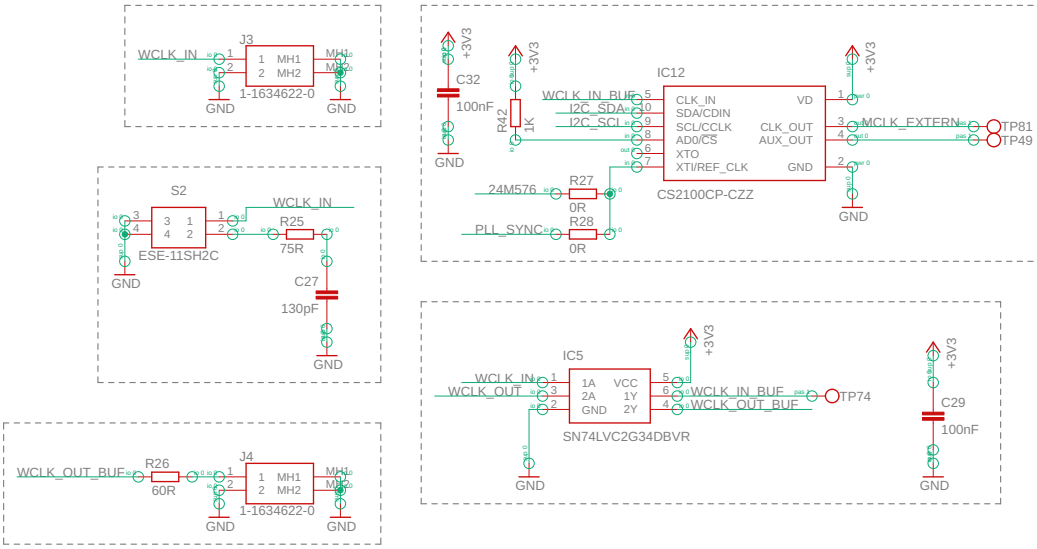
TITLE: baseboard	
Document Number:	REV:
Date: 03.09.20 19:07	Sheet: 5/8





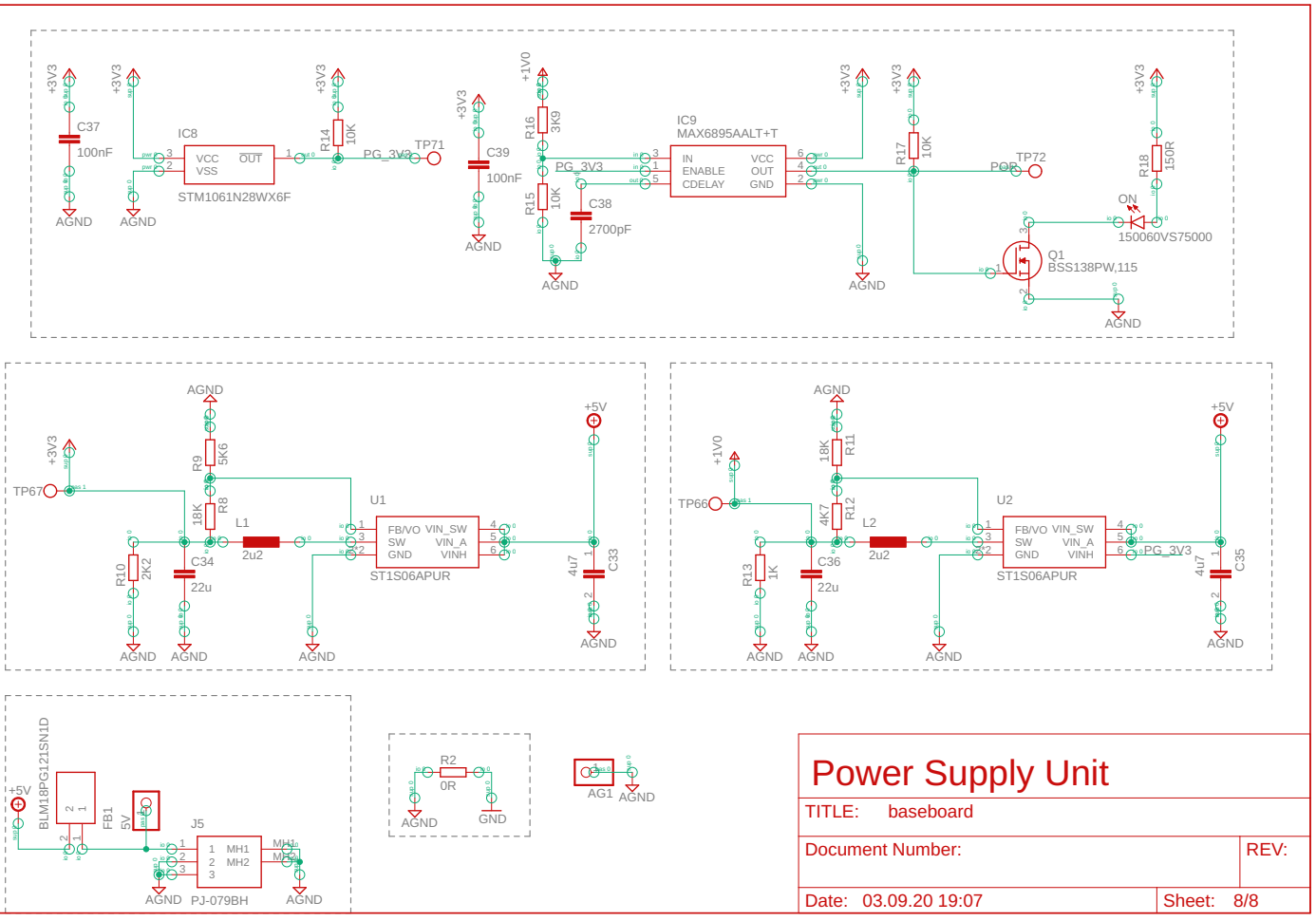
Internal clocking

TITLE: baseboard	
Document Number:	REV:
Date: 03.09.20 19:07	Sheet: 6/8



External clocking

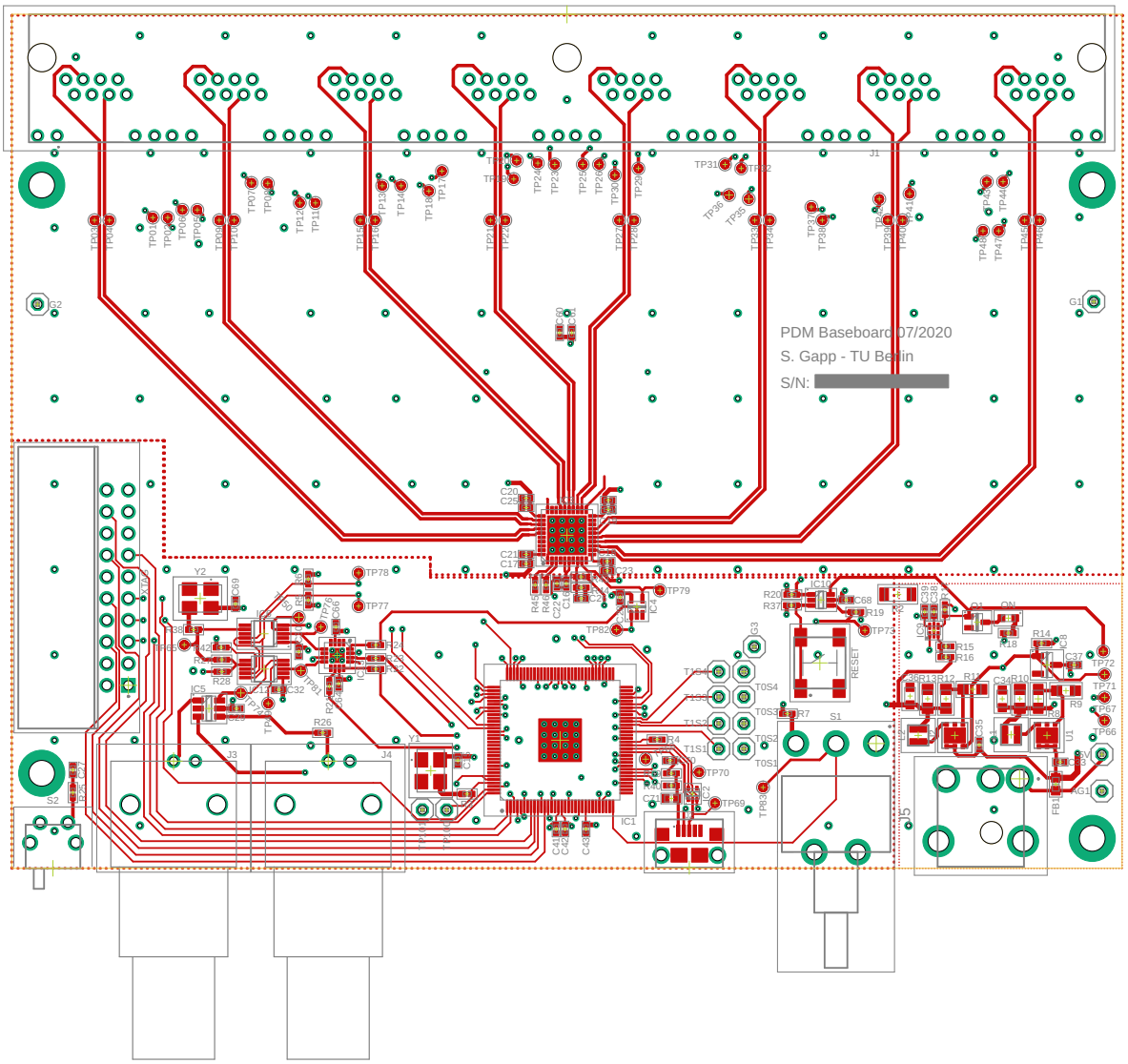
TITLE: baseboard	
Document Number:	REV:
Date: 03.09.20 19:07	Sheet: 7/8



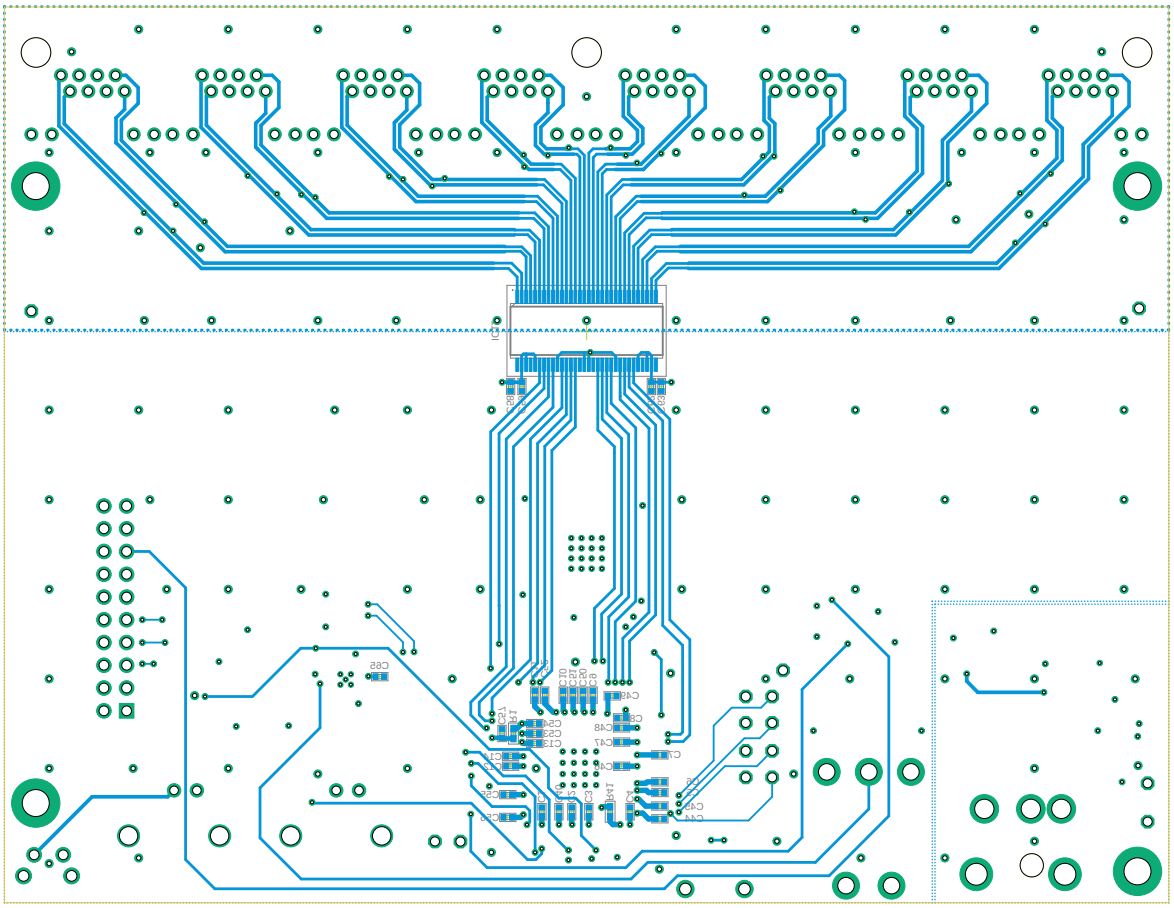
Power Supply Unit

TITLE: baseboard	
Document Number:	REV:
Date: 03.09.20 19:07	Sheet: 8/8

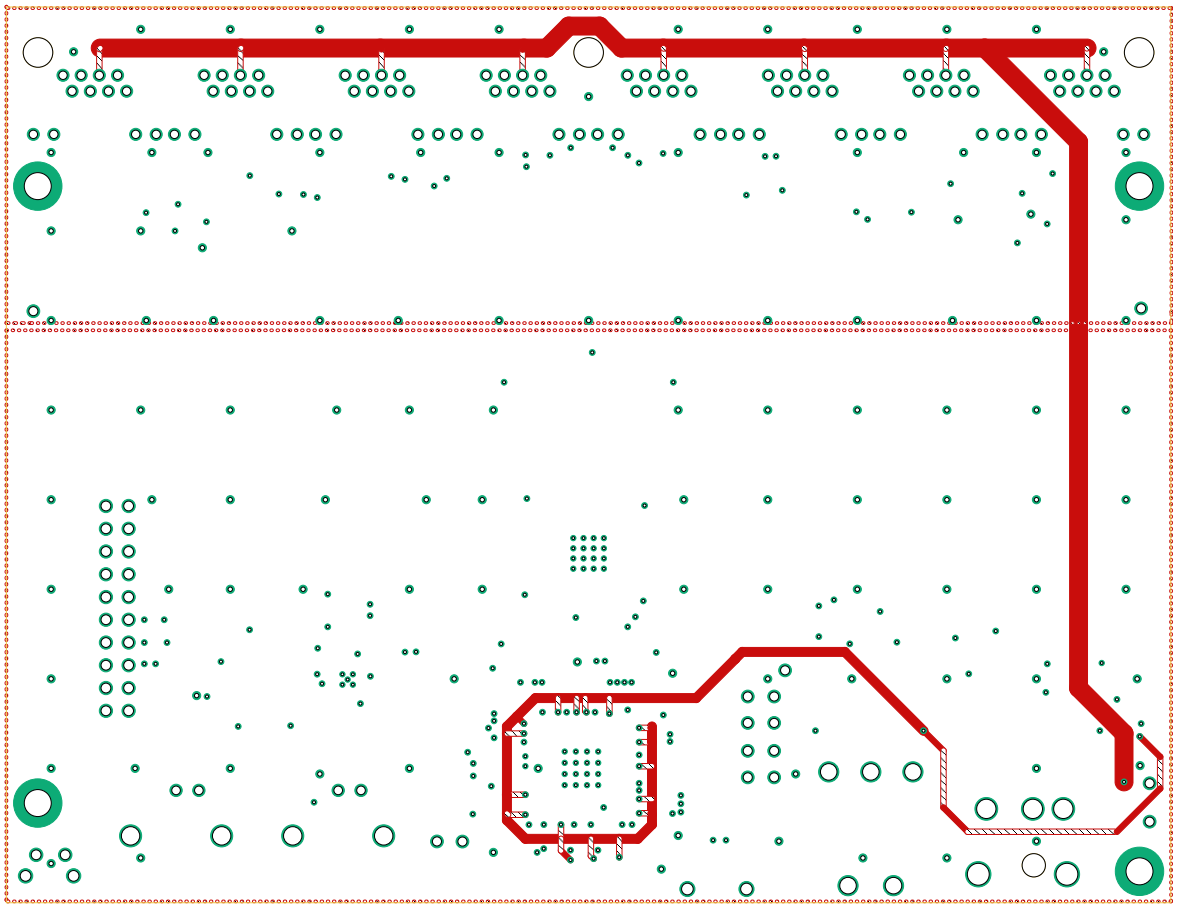
A.5 Base board Layout Top



A.6 Base board Layout Bottom



A.7 Base board Layout Power



Chapter B

Port Assignments

	Port						Reserved Pins		Features			Pin name	Package Pin		
	1b	*4b*	*8b*	*16b*	*32b*	*Link*	*QSPI*	*RGMII*	*I/O rail*	*BANK*	*Drive mA*		*TQ128*	*PCB Naming*	*Code Naming*
Tile 0	P1A0								IOL	0	4	X0D00	30		
	P1B0					xlink3_tx2	SS		IOL	0	4	X0D01	27	QSPI_SS	PORT_SQI_CS
		P4A0	P8A0	P16A0	P32A20				IOL	0	4	X0D02	37	TILE0_SPARE1	
		P4A1	P8A1	P16A1	P32A21				IOL	0	4	X0D03	38	TILE0_SPARE2	
		P4B0	P8A2	P16A2	P32A22		IO0		IOL	0	4	X0D04	31	QSPI_D0	PORT_SQI_SIO
		P4B1	P8A3	P16A3	P32A23		IO1		IOL	0	4	X0D05	33	QSPI_D1	*PORT_SQI_SIO*
		P4B2	P8A4	P16A4	P32A24		IO2		IOL	0	4	X0D06	34	QSPI_D2	*PORT_SQI_SIO*
		P4B3	P8A5	P16A5	P32A25		IO3		IOL	0	4	X0D07	35	QSPI_D3	*PORT_SQI_SIO*
		P4A2	P8A6	P16A6	P32A26				IOL	0	4	X0D08	39	TILE0_SPARE3	
		P4A3	P8A7	P16A7	P32A27				IOL	0	4	X0D09	40	TILE0_SPARE4	
	P1C0					xlink3_tx3	QSCLK		IOL	0	4	X0D10	28	SPI_CLK	PORT_SQI_SCLK
	P1D0								IOL	0	4	X0D11	32	CLK_SRC_SEL	PORT_CLK_SRC_SEL
	P1E0								IOR	1	4	X0D12	62	MIC_CLK	PORT_PDM_CLK
	P1F0								IOR	1	4	X0D13	63		
		P4C0	P8B0	P16A8	P32A28				IOR	1	4	X0D14	57	MC_01_DATA	PORT_PDM_DATA_0_to_7
		P4C1	P8B1	P16A9	P32A29				IOR	1	4	X0D15	58	MC_02_DATA	*PORT_PDM_DATA_0_to_7*
		P4D0	P8B2	P16A10		xlink4_rx4			IOR	1	4	X0D16	68	MC_03_DATA	*PORT_PDM_DATA_0_to_7*
		P4D1	P8B3	P16A11		xlink4_rx3			IOR	1	4	X0D17	69	MC_04_DATA	*PORT_PDM_DATA_0_to_7*
		P4D2	P8B4	P16A12		xlink4_rx2			IOR	1	4	X0D18	70	MC_05_DATA	*PORT_PDM_DATA_0_to_7*
		P4D3	P8B5	P16A13		xlink4_rx1			IOR	1	4	X0D19	71	MC_06_DATA	*PORT_PDM_DATA_0_to_7*
		P4C2	P8B6	P16A14	P32A30				IOR	1	4	X0D20	59	MC_07_DATA	*PORT_PDM_DATA_0_to_7*
		P4C3	P8B7	P16A15	P32A31				IOR	1	4	X0D21	61	MC_08_DATA	*PORT_PDM_DATA_0_to_7*
	P1G0								IOR	1	4	X0D22	64		
	P1H0								IOR	1	4	X0D23	66		
	P1I0					xlink7_rx0			IOR	2	4	X0D24	88		
	P1J0					xlink7_tx0			IOR	2	4	X0D25	89		
		P4E0	P8C0	P16B0		xlink7_tx3			IOR	2	4	X0D26	93	MC_09_DATA	PORT_PDM_DATA_8_to_15
		P4E1	P8C1	P16B1		xlink7_tx4			IOR	2	4	X0D27	94	MC_10_DATA	*PORT_PDM_DATA_8_to_15*
		P4F0	P8C2	P16B2					IOR	2	4	X0D28	96	MC_11_DATA	*PORT_PDM_DATA_8_to_15*
		P4F1	P8C3	P16B3					IOR	2	4	X0D29	98	MC_12_DATA	*PORT_PDM_DATA_8_to_15*
		P4F2	P8C4	P16B4					IOR	2	4	X0D30	99	MC_13_DATA	*PORT_PDM_DATA_8_to_15*
		P4F3	P8C5	P16B5					IOR	2	4	X0D31	100	MC_14_DATA	*PORT_PDM_DATA_8_to_15*
		P4E2	P8C6	P16B6					IOR	2	4	X0D32	95	MC_15_DATA	*PORT_PDM_DATA_8_to_15*
		P4E3	P8C7	P16B7					IOR	2	4	X0D33	97	MC_16_DATA	*PORT_PDM_DATA_8_to_15*
	P1K0					xlink7_tx1			IOR	2	4	X0D34	90		
	P1L0					xlink7_tx2			IOR	2	4	X0D35	91		
	P1M0		P8D0	P16B8					IOL	3	4	X0D36	3	MCLK_TILE0	PORT_PDM_MCLK
	P1N0		P8D1	P16B9		xlink0_rx4			IOL	3	4	X0D37	4	MCLK_WCLK	PORT_WCLK_MCLK
	P1O0		P8D2	P16B10		xlink0_rx3			IOL	3	4	X0D38	5	WCLK_OUT	PORT_WCLK_OUT
	P1P0		P8D3	P16B11		xlink0_rx2			IOL	3	4	X0D39	7		
			P8D4	P16B12		xlink0_rx1			IOL	3	4	X0D40	8	XL_DN1	
			P8D5	P16B13		xlink0_rx0			IOL	3	4	X0D41	9	XL_DN0	
			P8D6	P16B14		xlink0_tx0			IOL	3	4	X0D42	10	XL_UP0	
			P8D7	P16B15		xlink0_tx1			IOL	3	4	X0D43	12	XL_UP1	

	Port						Reserved Pins		Features			Pin name	Package Pin		
	1b	*4b*	*8b*	*16b*	*32b*	*Link*	*QSPI*	*RGMII*	*I/O rail*	*BANK*	*Drive mA*		*TQ128*	*PCB Naming*	*Code Naming*
Tile 1	P1A0					xlink7_rx2			IOR	0	4	X1D00	85		
	P1B0					xlink7_rx1			IOR	0	4	X1D01	86		
		P4A0	P8A0	P16A0	P32A20	xlink4_rx0			IOR	0	4	X1D02	72		
		P4A1	P8A1	P16A1	P32A21	xlink4_tx0			IOR	0	4	X1D03	74		
		P4B0	P8A2	P16A2	P32A22	xlink4_tx1			IOR	0	4	X1D04	75		
		P4B1	P8A3	P16A3	P32A23	xlink4_tx2			IOR	0	4	X1D05	76		
		P4B2	P8A4	P16A4	P32A24	xlink4_tx3			IOR	0	4	X1D06	77		
		P4B3	P8A5	P16A5	P32A25	xlink4_tx4			IOR	0	4	X1D07	79		
		P4A2	P8A6	P16A6	P32A26	xlink7_rx4			IOR	0	4	X1D08	82		
		P4A3	P8A7	P16A7	P32A27	xlink7_rx3			IOR	0	4	X1D09	84		
	P1C0								IOT	0	4	X1D10	121		
	P1D0								IOT	0	4	X1D11	122		
		P4C0	P8B0	P16A8	P32A28				IOR	1	4	X1D14	51	TILE1_SPARE1	
		P4C1	P8B1	P16A9	P32A29				IOR	1	4	X1D15	53	TILE1_SPARE2	
		P4D0	P8B2	P16A10		xlink3_rx1			IOL	1	4	X1D16	22	PLL_SYNC	PORT_PLL_REF
		P4D1	P8B3	P16A11		xlink3_rx0			IOL	1	4	X1D17	23		
		P4D2	P8B4	P16A12		xlink3_tx0			IOL	1	4	X1D18	25		
		P4D3	P8B5	P16A13		xlink3_tx1			IOL	1	4	X1D19	26		
		P4C2	P8B6	P16A14	P32A30				IOR	1	4	X1D20	54	TILE1_SPARE3	
		P4C3	P8B7	P16A15	P32A31				IOR	1	4	X1D21	55	TILE1_SPARE4	
		P4E0	P8C0	P16B0				tx_clk	IOT	2	8	X1D26	112		PORT_MCLK_COUNT
		P4E1	P8C1	P16B1				tx_ctl	IOT	2	8	X1D27	113		
		P4F0	P8C2	P16B2				rx_clk	IOT	2	4	X1D28	114		
		P4F1	P8C3	P16B3				rx_ctl	IOT	2	4	X1D29	115		
		P4F2	P8C4	P16B4				rx0	IOT	2	4	X1D30	116		
		P4F3	P8C5	P16B5				rx1	IOT	2	4	X1D31	117		
		P4E2	P8C6	P16B6				rx2	IOT	2	4	X1D32	118		
		P4E3	P8C7	P16B7				rx3	IOT	2	4	X1D33	119		
	P1L0					xlink0_tx3			IOL	2	4	X1D35	13	MCLK_TILE1	PORT_MCLK_IN
	P1M0		P8D0	P16B8		xlink0_tx4			IOL	3	4	X1D36	15		
	P1N0		P8D1	P16B9		xlink3_rx4			IOL	3	4	X1D37	18		
	P1O0		P8D2	P16B10		xlink3_rx3			IOL	3	4	X1D38	20		
	P1P0		P8D3	P16B11		xlink3_rx2			IOL	3	4	X1D39	21		
			P8D4	P16B12				tx3	IOT	3	8	X1D40	106		
			P8D5	P16B13				tx2	IOT	3	8	X1D41	107		
			P8D6	P16B14				tx1	IOT	3	8	X1D42	108	I2C_SCL	
			P8D7	P16B15				tx0	IOT	3	8	X1D43	109	I2C_SDA	

Chapter C

Test Points

Name	Description	Output	Name	Description	Output
T0S1	Spare Output for Tile0		TP44	MC_02_DATA_P	PDM Data
T0S2	Spare Output for Tile0		TP45	MC_01_02_CLK_IN_P	3.072MHz
T0S3	Spare Output for Tile0		TP46	MC_01_02_CLK_IN_N	3.072MHz
T0S4	Spare Output for Tile0		TP47	MC_01_DATA_P	PDM Data
T1S1	Spare Output for Tile1		TP48	MC_01_DATA_N	PDM Data
T1S2	Spare Output for Tile1				
T1S3	Spare Output for Tile1		TP49	AUX OUT CS2100	
T1S4	Spare Output for Tile1		TP50	AUX OUT CS2100	
			TP65	25M576 Crystal out	24.576MHz
TP01	MC_16_DATA_N	PDM Data	TP66	1V Output	1V
TP02	MC_16_DATA_P	PDM Data	TP67	3V3 Output	3.3V
TP03	MC_15_16_CLK_IN_P	3.072MHz	TP68	USB_VBUS	5V
TP04	MC_15_16_CLK_IN_N	3.072MHz	TP69	USB_D_P	USB Data
TP05	MC_15_DATA_P	PDM Data	TP70	USB_D_N	USB Data
TP06	MC_15_DATA_N	PDM Data	TP71	PG_3V3	3.3V
TP07	MC_14_DATA_N	PDM Data	TP72	POR	3.3V
TP08	MC_14_DATA_P	PDM Data	TP73	RST_N	3.3V
TP09	MC_13_14_CLK_IN_P	3.072MHz	TP74	WCLK_IN_BUF	48kHz
TP10	MC_13_14_CLK_IN_N	3.072MHz	TP76	MCLK_INTERN	12.288MHz
TP11	MC_13_DATA_P	PDM Data	TP77	I2C_SDA	I2C Data
TP12	MC_13_DATA_N	PDM Data	TP78	I2C_SCL	I2C Clock
TP13	MC_12_DATA_N	PDM Data	TP79	REF_IN0_P	3.072MHz
TP14	MC_12_DATA_P	PDM Data	TP81	MCLK_EXTERN	12.288MHz
TP15	MC_11_12_CLK_IN_P	3.072MHz	TP82	MIC_CLK	3.072MHz
TP16	MC_11_12_CLK_IN_N	3.072MHz	TP83	CLK_SRC_SEL	0V/3.3V
TP17	MC_11_DATA_P	PDM Data	TP100	CLK_24M	24MHz
TP18	MC_11_DATA_N	PDM Data	TP101	GND	GND
TP19	MC_10_DATA_N	PDM Data			
TP20	MC_10_DATA_P	PDM Data	5V	5V	5V
TP21	MC_09_10_CLK_IN_P	3.072MHz	AG1	Analog Ground	GND
TP22	MC_09_10_CLK_IN_N	3.072MHz			
TP23	MC_09_DATA_P	PDM Data			
TP24	MC_09_DATA_N	PDM Data			
TP25	MC_08_DATA_P	PDM Data			
TP26	MC_08_DATA_N	PDM Data			
TP27	MC_07_08_CLK_IN_P	3.072MHz			
TP28	MC_07_08_CLK_IN_N	3.072MHz			
TP29	MC_07_DATA_P	PDM Data			
TP30	MC_07_DATA_N	PDM Data			
TP31	MC_06_DATA_N	PDM Data			
TP32	MC_06_DATA_P	PDM Data			
TP33	MC_05_06_CLK_IN_P	3.072MHz			
TP34	MC_05_06_CLK_IN_N	3.072MHz			
TP35	MC_04_DATA_P	PDM Data			
TP36	MC_05_DATA_N	PDM Data			
TP37	MC_04_DATA_N	PDM Data			
TP38	MC_04_DATA_P	PDM Data			
TP39	MC_03_04_CLK_IN_P	3.072MHz			
TP40	MC_03_04_CLK_IN_N	3.072MHz			
TP41	MC_03_DATA_P	PDM Data			
TP42	MC_03_DATA_N	PDM Data			
TP43	MC_02_DATA_N	PDM Data			

Chapter D

CS2100 configuration

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Address	***0x01***		***Device ID***					
Description	Device ID					Revision		
Default Val	*0*	*0*	*0*	*0*	*0*	*X*	*X*	*X*
INT CLK	r	r	r	r	r	r	r	r
EXT CLK	r	r	r	r	r	r	r	r
Address	***0x02***		***Device Ctrl***					
Description	Unlock	Reserved					AuxOutDis	ClkOutDis
Default Val	*X*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	0	0	0	0	0	0	0	0
EXT CLK	0	0	0	0	0	0	0	0
Address	***0x03***		***Device Config 1***					
Description	RmodSel			Reserved	Reserved	AuxOutSrc		EnDevCfg1
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	0	0	0	0	0	1	0	1
EXT CLK	0	0	0	0	0	1	0	1
Address	***0x05***		***Global Config***					
Description	Reserved				Freeze	Reserved		EnDevCfg2
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	0	0	0	0	0	0	0	1
EXT CLK	0	0	0	0	0	0	0	1
Address	***0x06***		***R_UD[31:24]***					
Description	31	30	29	28	27	26	25	24
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	*0*	*0*	*0*	*0*	0	0	0	0
EXT CLK								
Address	***0x07***		***R_UD[23:16]***					
Description	23	22	21	20	19	18	17	16
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	*1*	*1*	*0*	*0*	0	1	0	0
EXT CLK								
Address	***0x08***		***R_UD[15:8]***					
Description	15	14	13	12	11	10	9	8
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	1	0	0	1	1	0	1	1
EXT CLK								
Address	***0x09***		***R_UD[7:0]***					
Description	7	6	5	4	3	2	1	0
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	1	0	1	0	0	1	0	1
EXT CLK								
Address	***0x16***		***Funct Cfg 1***					
Description	ClkSkipEn	AuxLockCfg	Reserved	RefClkDiv		Reserved		
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	0	0	0	0	1	0	0	0
EXT CLK	0	0	0	1	0	0	0	0
Address	***0x17***		***Funct Cfg 2***					
Description	Reserved			ClkOutUnl	LFRatioCfg	Reserved		
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
EXT CLK	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*
Address	***0x1E***		***Funct Cfg 3***					
Description	Reserved	ClkIn_BW			Reserved			
Default Val	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
INT CLK	0	0	0	0	0	0	0	0
EXT CLK	0	0	0	0	0	0	0	0

Chapter E

SpectAcoular Code

Add "pdm_bb" to the choices line in *Spectacoular/apps/Measurement_App/main.py*:

```
55 parser.add_argument(
56   '--device',
57   type=str,
58   default="phantom",
59   choices=["uma16","tornado","typhoon", "pdm_bb", "phantom
       "],
60   help='Connected device.')
```

Extend the DEVICE conditional

```
94 if DEVICE == 'uma16':
95     mg_file = 'minidsp_uma16.xml'
96     inputSignalGen = get_interface(DEVICE)
97     ch_names = [str(_) for _ in range(inputSignalGen.
           numchannels)]
98     grid = RectGrid( x_min=-0.15, x_max=0.15, y_min=-0.15,
           y_max=0.15, z=0.2, increment=0.01)
99 elif DEVICE == 'pdm_bb':
100    mg_file = 'pdm_bb.xml'
101    inputSignalGen = get_interface(DEVICE)
102    print(inputSignalGen)
103    ch_names = [str(_) for _ in range(inputSignalGen.
           numchannels)]
104    grid = RectGrid( x_min=-0.4, x_max=0.4, y_min=-0.2,
           y_max=0.2, z=0.3, increment=0.01)
```

The xml file describing the array layout is shown in App. E.3.
Extend the "get_interfaces()" function for "pdm_bb" in *Spectacoular/apps/Measurement_App/interfaces.py*:

```
23 def get_interface(device,syncorder=[]):
24     if device == 'uma16':
25         from acuma16 import UMA16SamplesGenerator
26         InputSignalGen = UMA16SamplesGenerator()
27         return InputSignalGen
```

```
28      elif device == 'pdm_bb':
29          from pdm_bb import PDMBBSamplesGenerator
30          InputSignalGen = PDMBBSamplesGenerator()
31          return InputSignalGen
```

The created "pdm_bb" class is shown in App. E.1. The class requires a sensitivity value to scale the analog output correctly. The value is applied in line 52 and is based on a visual estimation.

E.1 Class pdm_bb

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Wed Sep 09 09:53:20 2020
5
6  @author: gapp
7  """
8
9  from traits.api import Trait, Long, Bool, List,Int
10 import numpy as np
11 from acoular import SamplesGenerator
12 import sounddevice as sd
13
14 class PDMBBSamplesGenerator(SamplesGenerator):
15
16     """
17     Class for signal processing from PDM Baseboard device
18
19     Generate an output via the generator :meth:`result`.
20     """
21
22     def __init__(self):
23 #         HasTraits.__init__(self)
24         pdm_bb = [(i,dev) for i,dev in enumerate(sd.
               query_devices()) if 'PDM_Test' in dev['name']]
25         if not pdm_bb:
26             raise IOError("PDM_Baseboards_not_found!")
```

```
27            else:
28                dev_index,pdm_bb = pdm_bb[0]
29                print("Found {}".format(pdm_bb['name']))
30                self.numchannels = pdm_bb['max_input_channels']
31                self.sample_freq = pdm_bb['default_samplerate']
32                self.device = dev_index
33
34        #: Device Index
35        device = Int()
36
37        #: Number of input channels.
38        numchannels = Trait(
39            desc="number of analog input channels that collects
                data")
40
41        #: Number of samples to collect; defaults to -1.
42        # If is set to -1 device collects till user breaks
              streaming by setting Trait: collectsamples = False
43        numsamples = Long(-1,
44            desc="number of samples to collect")
45
46        #: Indicates if samples are collected, helper trait to
              break result loop
47        collectsamples = Bool(True,
48            desc="Indicates if samples are collected")
49
50        #: sensitivity value for sensors at analog inputs
51        # if single value is defined, it's applied to all analog
              input channels
52        sensval = List(32*[0.004],
53            desc="sensitivity value for sensors at analog inputs
                ")
54
55        #: Sampling frequency of the signal, changes with
              sinusdevices
56        sample_freq = Trait(
57            desc="sampling frequency")
58
```

```
59      def result(self, num):
60          """
61          Python generator that yields the output block-wise.
               Use at least a
62          block-size of one ring cache block.
63
64          Parameters
65          ----------
66          num : integer
67              This parameter defines the size of the blocks to
                   be yielded
68              (i.e. the number of samples per block).
69
70          Returns
71          -------
72          Samples in blocks of shape (num, :attr:`numchannels
               `).
73              The last block may be shorter than num.
74          """
75
76          # if one sensitivity value for all channels is given
77          if len(self.sensval) == 1: self.sensval = self.
               sensval*self.numchannels
78          assert self.numchannels == len(self.sensval),"{} 
               sensitivity values \
79  ........does not fit to {} channels".format(len(self.sensval
    ),self.numchannels)
80          sens = np.array(self.sensval)
81
82          stream_obj = sd.InputStream(
83                  device=self.device, blocksize=num,channels=
                       self.numchannels,
84                  samplerate=self.sample_freq)
85
86          with stream_obj as stream:
87              if self.numsamples == -1:
88                  while self.collectsamples: # yield data as
                       long as collectsamples is True
```

```
89                      data, overflow = stream.read(num)
90                      yield data[:num]/sens
91                  else:
92                      pass
93
94              elif self.numsamples > 0: # amount of samples to
                    collect is specified by user
95                  samples_count=0 # numsamples counter
96                  while samples_count < self.numsamples:
97                          anz = min(num, self.numsamples-
                                samples_count)
98                          data, overflow = stream.read(num)
99                          yield data[:anz]/sens
100                         samples_count += anz
101         return
```

Listing E.1: pdm_bb class

E.2 Microphone grid

```
1  import numpy as np
2  from create_xml import *
3  import matplotlib.pyplot as plt
4
5  no_of_mics = 32
6  # All distances in mm
7  mic_distance = 17
8  pcb_distance = 17
9  pcb_width = 37
10
11 print("Parameters for a microphone line array in x dimension
       ")
12 print("++++++++++++++++++++++++++++++++++++++++++++++++++++++
       ")
13 print("No of microphones: " + str(no_of_mics))
14 print("Distance between microphones in mm: " + str(
       mic_distance))
```

```
15 print("PCB width in mm: " + str(pcb_width))
16 print("Distance between PCBs in mm: " +str(pcb_distance))
17 print("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   ")
18 print("Calculating parameters.")
19
20 mic_distance_edge = pcb_width/2 - mic_distance/2
21
22 # Calculate microphone distances
23 mic_distance_adj = []
24
25 mic_distance_adj.append(pcb_distance/2 + mic_distance_edge)
26 mic_distance_adj.append(mic_distance_adj[0] + mic_distance)
27
28 for i in range(1, int(no_of_mics/2)-1):
29     if i%2:
30         mic_distance_adj.append(mic_distance_adj[i] +
              mic_distance_edge + pcb_distance +
              mic_distance_edge)
31     else:
32         mic_distance_adj.append(mic_distance_adj[i] +
              mic_distance)
33
34 # Convert to numpy
35 mic_distance_adj = np.array(mic_distance_adj)
36 # Complete Vector
37 result = np.around((np.append(mic_distance_adj[::-1],
     mic_distance_adj*(-1)))*10**-3, 9)
38 y = np.zeros(len(result))
39
40 print("Creating XML file.")
41 if(create_xml(result, 0, 0, "pdm_bb")<0):
42     print("Error in xml file creation!")
43 else:
44     print("XML file created successfully!")
45
46 plt.rcParams.update({'font.size': 36})
47 plt.plot(result, y, 'o', markersize=14)
```

```
48 plt.grid(True, which="both")
49 plt.title("Microphone layout")
50 plt.xlabel("x in m")
51 plt.ylim()
52 plt.show()
```

Listing E.2: calc_mic_distance

```
1  def create_xml(mic_pos_x, mic_pos_y, mic_pos_z,
       MicArray_name):
2
3      point_strings = []
4      header = "<?xml version=\"1.0\" encoding=\"utf-8\"?><
           MicArray name=\"" + MicArray_name +"\">\n"
5      footer = "</MicArray>"
6      filename = MicArray_name + ".xml"
7
8      ## Assume that only x is a vector
9      for i in range(len(mic_pos_x)):
10         point_strings.append("<pos Name=\"Point " +str(i+1)
               + "\" x=\"" + str(mic_pos_x[i]) + "\" y=\"" + str(
               mic_pos_y) + "\" z=\"" + str(mic_pos_z) + "\"/>\n"
                )
11
12     # Save to xml file
13     with open(filename, "w") as f:
14         f.write(header)
15         for i in range(len(point_strings)):
16             f.write(point_strings[i])
17         f.write(footer)
18
19     return 0
```

Listing E.3: create mic distance xml

E.3 pdm_bb.xml

```
1  <?xml version="1.0" encoding="utf-8"?><MicArray name="pdm_bb
       ">
2  <pos Name="Point 1" x="0.4135" y="0" z="0"/>
3  <pos Name="Point 2" x="0.3965" y="0" z="0"/>
4  <pos Name="Point 3" x="0.3595" y="0" z="0"/>
5  <pos Name="Point 4" x="0.3425" y="0" z="0"/>
6  <pos Name="Point 5" x="0.3055" y="0" z="0"/>
7  <pos Name="Point 6" x="0.2885" y="0" z="0"/>
8  <pos Name="Point 7" x="0.2515" y="0" z="0"/>
9  <pos Name="Point 8" x="0.2345" y="0" z="0"/>
10 <pos Name="Point 9" x="0.1975" y="0" z="0"/>
11 <pos Name="Point 10" x="0.1805" y="0" z="0"/>
12 <pos Name="Point 11" x="0.1435" y="0" z="0"/>
13 <pos Name="Point 12" x="0.1265" y="0" z="0"/>
14 <pos Name="Point 13" x="0.0895" y="0" z="0"/>
15 <pos Name="Point 14" x="0.0725" y="0" z="0"/>
16 <pos Name="Point 15" x="0.0355" y="0" z="0"/>
17 <pos Name="Point 16" x="0.0185" y="0" z="0"/>
18 <pos Name="Point 17" x="-0.0185" y="0" z="0"/>
19 <pos Name="Point 18" x="-0.0355" y="0" z="0"/>
20 <pos Name="Point 19" x="-0.0725" y="0" z="0"/>
21 <pos Name="Point 20" x="-0.0895" y="0" z="0"/>
22 <pos Name="Point 21" x="-0.1265" y="0" z="0"/>
23 <pos Name="Point 22" x="-0.1435" y="0" z="0"/>
24 <pos Name="Point 23" x="-0.1805" y="0" z="0"/>
25 <pos Name="Point 24" x="-0.1975" y="0" z="0"/>
26 <pos Name="Point 25" x="-0.2345" y="0" z="0"/>
27 <pos Name="Point 26" x="-0.2515" y="0" z="0"/>
28 <pos Name="Point 27" x="-0.2885" y="0" z="0"/>
29 <pos Name="Point 28" x="-0.3055" y="0" z="0"/>
30 <pos Name="Point 29" x="-0.3425" y="0" z="0"/>
31 <pos Name="Point 30" x="-0.3595" y="0" z="0"/>
32 <pos Name="Point 31" x="-0.3965" y="0" z="0"/>
33 <pos Name="Point 32" x="-0.4135" y="0" z="0"/>
34 </MicArray>
```

Listing E.4: XML output for mic positions