Andreas Wöhrer, BSc

16 Channel USB 2.0 Sound Card for Digital MEMS Microphones

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Franz Pernkopf

Institute of Signal Processing and Speech Communication

Graz, August 2018

AFFIDAVIT[1]

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Graz, _____ _____

Date Signature

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Microphone arrays are a vital aspect of research into the field of audio signal processing, which includes speech recognition, beamforming and source separation. The more channels a microphone array provides, the more universally usable it becomes. Since there are research projects requiring recordings with up to sixteen audio channels the need for a new sound card arose. The subject of this thesis was the development, building and testing of such a sound card. To keep the whole setup reasonably small Micro-Electro-Mechanic-System (MEMS) microphones were selected. These microphones are growing in popularity as they are very small in size and MEMS microphones with even better signal to noise ratios are becoming increasingly available. A USB 2.0 interface was chosen as the protocol between computer and sound card. This was done for easy usage of the sound card, as the USB 2.0 standard includes audio recording devices and provides plug and play functionality. The new hardware and software was developed based on an evaluation board of the company XMOS. It increases the microphone count from seven to sixteen and provides the possibility to connect the microphones with ribbon cables to the sound card. Different spatial layouts of the microphone array can therefore be easily achieved. In this thesis the functional principle of MEMS microphones and the included analog-digital-converter is also displayed. The USB 2.0 protocol is studied in detail and the xCORE XUF200 processor family from XMOS is introduced. Furthermore, the developed hardware, software and prototype is introduced and documented along with some functional tests of the sound card and microphone array prototype.

Contents

1. Introduction

The idea for this thesis was born at the Institute of Signal Processing and Speech at Graz University of Technology as a lot of their research is about different kinds of audio signals. Some of the fields like channel separation, beamforming or speech recognition are working with more than only one microphone as a microphone array offers the benefit of recording audio data not only over time but also provides spatial information. This spatial information about the recorded sound gives a completely new dimension and is therefore essential for research. As the spatial dimension is a discrete one which increases with raising numbers of microphones the need for a microphone array with more than 8 channels arose and the decision was made to develop a sound card as subject of this thesis. At the core of this sound card is a processor of the xCORE family by the company XMOS. This processor was used based on a demo board with 7 channels. The schematic, layout and firmware of this demo board had to be altered to support 16 microphones. A class of microphones called MEMS was used as they are small in size and lately gained a lot of popularity. This is due to the fact that they are used in lots of small devices and are actively developed to increase performance. After creating a prototype, basic functional testing was done to confirm the correct operation of the sound card.

1.1. Scope of this Thesis

The first step of this thesis is to design and build a USB sound card with 16 MEMS microphone inputs and a headphone jack for a stereo audio output. This was done by redesigning and extending the hardware and software of the XMOS demo kit "xCORE Array Microphone" [1] which has 7 microphones and a stereo audio output.

Therefore the first step was to familiarize oneself with the XCORE XUF200 processor family that has a multi-core architecture. This includes an extension to the C programming language called XC with special keywords and operators for parallelism, inter core communication and clocked IO.
The next step was to extend the schematics to 16 microphones and also remove unused components. From this schematic a printed circuit board (PCB) was designed, ordered, assembled and tested.
The last step was understanding and extending the firmware. The source code of the demo kit and some XMOS libraries were therefore altered to work with 16 microphones.

This includes understanding the USB protocol, as explained in detail in its own chapter.

1.2. Chapter Overview

This theses consists of three theoretical chapters (2. Audio, 3. USB and 4. Processor), two practical chapters (5. Hardware and 6. Firmware) and a chapter compiling the results (7.2. Functional Test).

Chapter 2 explains how the microphones and the ADC inside the microphones work. It also explains the pulse density modulation (PDM) signal produced by the microphones.

In Chapter 3 the USB protocol is presented. It covers USB versions, protocol basics, communication initiation, the concept of classification and describing USB devices with descriptors, USB transfer types, packet types and basic requests. There is also a section on USB audio which covers the basics of the USB audio device class.

Chapter 4 shows the capabilities and illustrates the architecture of the XMOS XUF200 multi core processor series. It also explains the XC programming language which is an extension to C tailored to the needs of the XMOS multi-core processors. Further there is a small introduction to the integrated development environment (IDE) from XMOS called XTime Composer and the programming process with a JTAG programmer.

Chapter 5 covers important parts of the hardware design and presents the most relevant parts of the schematic. The full schematics can be found in Appendix B.

Chapter 6 explains the structure of the firmware, all the tasks running in parallel, the communication between tasks and the used libraries. The use of the USB libraries is also explained in more detail and how the interface sends and receives audio data is defined. The signal processing chain for the audio signals from the MEMS microphones is the most detailed part as it is the most relevant code for this thesis.

2. Audio

2.1. MEMS Microphone

Micro-Electro-Mechanical System (MEMS) are systems which are often embedded in the waver of a microchip and their mechanical movements are restricted to the micrometer range. MEMS microphones have a small conductive, movable membrane which is directly connected to the waver and resides over a fixed, perforated back plate in order to allow air to flow in and out of the chamber between membrane and back plate. This back plate is also conductive and forms a small capacitor in conjunction with the membrane (see Figure 2.1). When the membrane is moving the capacitance of this capacitor changes depending on the size of the gap.

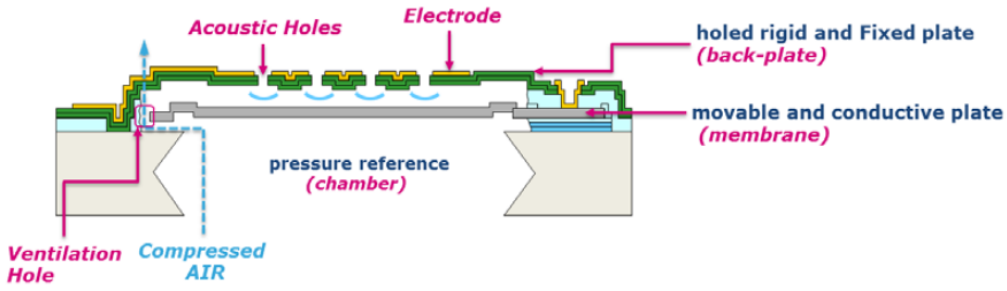


Figure 2.1.: Internal structure of the MEMS microphone membrane and the back plate on a wafer [2].

To utilize this effect a charge pump is used to create a fixed charge on the membrane. Due to the relations

$$C = \epsilon_0 \cdot \epsilon_r \cdot \frac{A}{d} \tag{2.1}$$

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

a change in voltage can be measured and amplified (equations for a parallel plate capacitor).

In analog MEMS microphones this amplified voltage is applied directly to one of the pins of the MEMS microphone. In digital MEMS microphones this amplified voltage is then converted into a digital Pulse Density Modulation (PDM) signal (details are in Chapter 2.3). This is done by an ASIC inside of the MEMS microphone which contains a Sigma-Delta-Modulator (see Chapter 2.2). The block diagram of a digital MEMS microphone is shown in Figure 2.2;

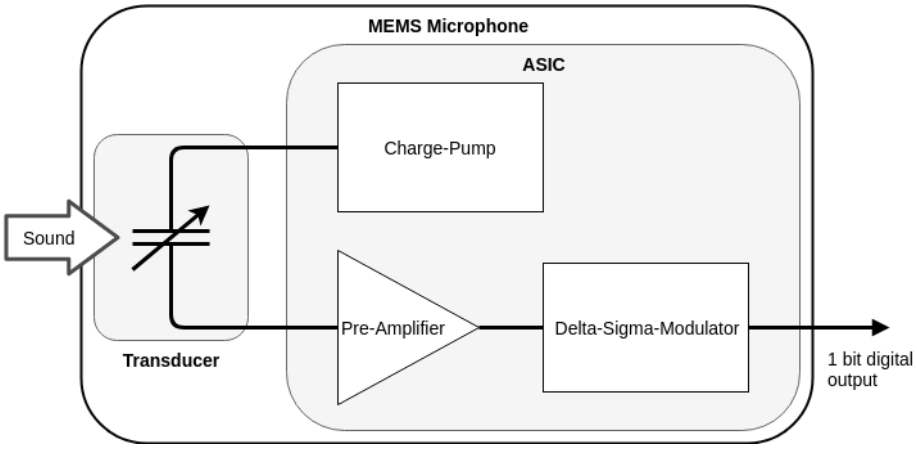


Figure 2.2.: Block diagram of a digital MEMS microphone.

MEMS microphones have the advantage that they are very small in size and the analog signal from the back plate has a very short path to the amplifying electronic. This means that they are fairly immune to electromagnetic noise compared to other microphones. Digital MEMS microphones in particular have the advantage that the signal path of the amplified analog is also kept short as it only goes from the amplifier to the Delta-Sigma-Modulator. The digital output signal is also far more insensitive to noise than analog signals. Although digital signals are insensitive to noise up to a certain level they too can be distorted to the point that bit errors occur. This distortions can be caused by parasitic capacitance, inductance or resistance between the microphone and the receiver. While transmitting over long distances impedance mismatches can also be a problem as they can distort the signals by creating reflections which in turn lead to bit errors. These bit error then show up as noise in the audio signal.

2.2. Delta-Sigma-Modulator

A Delta-Sigma-Modulator is a part of the Delta-Sigma Analog Digital Converter (ADC) which takes an analog signal and converts it to a PDM bitstream (see Chapter 2.3). The full ADC also includes a low pass filter at the end but this is not implemented in common digital MEMS microphones. A Delta-Sigma-Modulator can be interpreted as a 1 bit ADC. Since the modulator only has a 1 bit output it normally runs at a much higher frequency than the desired sampling rate. This oversampling then leads to an increase in the number of bits once it is converted to a PCM signal (see Chapter 2.3).

2.2.1. Functional Principle

The way this modulator works is that it has a comparator for the analog to digital conversion. The comparator has a binary output of zero if its input is negative or one if it is positive. The value of this comparator is then fed into a D-Flip-Flop which holds the value for one clock cycle and is also the output of the modulator.
The binary output is converted back to an analog signal and can only have two possible values - plus or minus the full scale voltage of the input signal range. This value is then

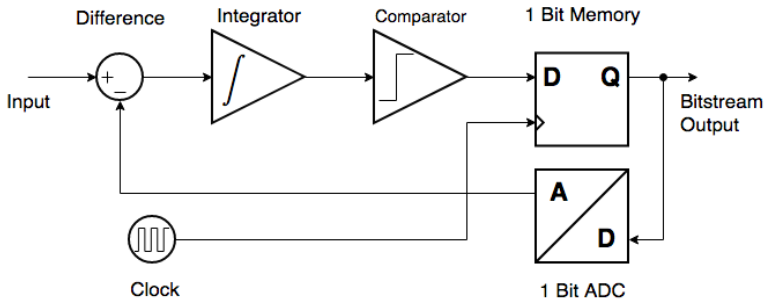


Figure 2.3.: Block diagram of a first order delta-sigma-ADC.

subtracted from the input signal which creates a difference signal. This difference represents the error between output and input and is the Delta part of the modulator.
The error is then integrated and fed into the comparator mentioned above. The slope of the output of the integrator is the value of the error signal. This means that if the error between input and output is large the integrator value reaches the comparator toggling point fast and if the error is small the toggling point of the comparator is reached slowly. This results in short pulses of ones and long pauses of zeros if the input signal is low. If the input signal is high then the pauses of zeros are short and the pulses of ones are long. This is illustrated in Figure 2.4.

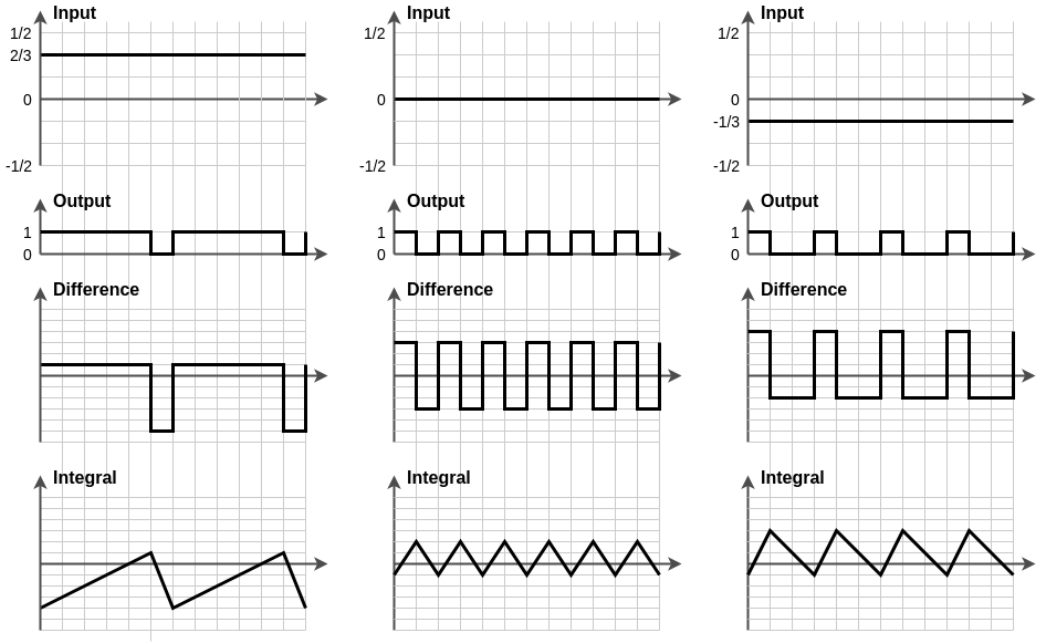


Figure 2.4.: Output examples of a Delta-Sigma-Modulator with the following input values: 83.3% (left), 50% (middle) and 33.3% (right).

In the following the advantage of a Delta-Sigma-Converter, which is its noise shaping capabilities, is discussed.

2.2.2. Quantization Noise

Quantization of an analog signal always produces quantization noise. This noise can be determined.

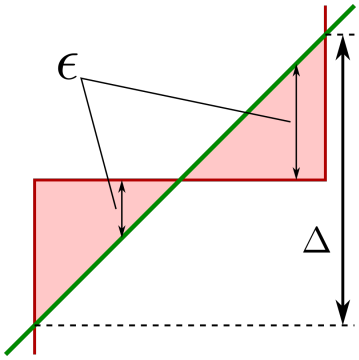


Figure 2.5.: Quantization error ϵ and quantization step Δ.

An ADC has a quantization width Δ which is the interval between the quantization steps. For each quantisized sample the quantization error has an equal probability anywhere between $-\frac{\Delta}{2}$ and $+\frac{\Delta}{2}$ which means a uniform probability function over the whole range of the quantization error. The quantization error is linear for each quantization step as shown in Figure 2.5.

The noise power can be calculated by integrating the squared quantization error:

$$\epsilon_{rms}^2 = \frac{1}{\Delta} \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} \epsilon^2 \, \mathrm{d}\epsilon = \frac{\Delta^2}{12} \tag{2.3}$$

This can also be expressed in terms of full scale range FS and number of bits N:

$$\epsilon_{rms}^2 = \frac{\Delta^2}{12} \approx \frac{FS^2}{3 \cdot 2^{2N}} \tag{2.4}$$

This means that the Noise Power is only affected by the size of the quantization step and not by the sampling rate.

2.2.3. Oversampling

Since it can be assumed that the noise spectrum over the frequency is flat (because the signals measured by a microphone are analog signals and are not highly periodic during normal operation) a relation between the noise floor PSD_{noise} and the sampling frequency f_s can be written down as:

$$PSD_{noise} = \frac{\epsilon_{rms}^2}{f_s} \tag{2.5}$$

This means that the noise floor can be reduced by oversampling due to the fact that the overall noise power, represented by the area under the noise floor, needs to stay the same. Figure 2.7 shows this effect (along with the noise shaping effect described in the next chapter).

2.2.4. Noise Shaping

Figure 2.6 shows the model of a first order Delta-Sigma-Converter in the Laplace Space.

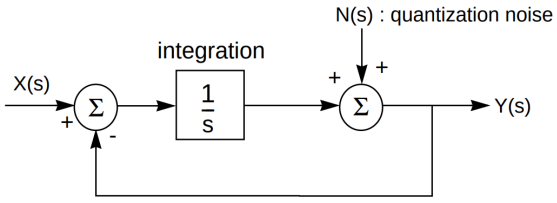


Figure 2.6.: Model of a Delta-Sigma-Converter [3].

The noise is added to the model where the quantization step occurs. From this model the signal transfere function

$$Y(s) = \frac{1}{s}\left[X(s) - Y(s)\right] \tag{2.6}$$

$$\frac{Y(s)}{X(s)} = \frac{1}{s+1} \qquad \text{... low pass} \tag{2.7}$$

and the noise transfer function

$$Y(s) = N(s) - \frac{1}{s}Y(s) \tag{2.8}$$

$$\frac{Y(s)}{N(s)} = \frac{s}{s+1} \qquad \text{... high pass} \tag{2.9}$$

can be derived [3].

This shows that the transfer function for the signal (Equation 2.7) has low pass behavior and the transfer function of the noise (Equation 2.9) follows a high pass behavior. This shapes the noise in a way that most of the noise power is above the nyquist frequency f_B of the original signal [3]. The resulting effect can be seen in Figure 2.7.

Since the signal is oversampled a low pass filter has to be applied to it, as it has the effect of cutting away most of the high pass noise power.

2.3. Pulse Density Modulation (PDM)

Pulse Density Modulation (PDM) is a 1 bit digital data stream which in this application is the encoded version of the analog audio signal. It is the output of the Delta-Sigma-Modulator inside the MEMS microphones. Since a 1 bit signal is not very useful, PDM signals have to be at a much faster sampling rate than the sampling rate of the encoded analog signal. In a

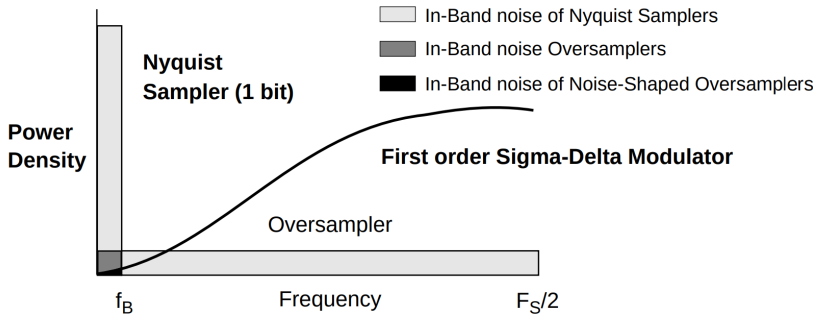


Figure 2.7.: Spectrum of a First-Order Sigma-Delta Noise Shaper [3].

digital PDM signal the amplitude of the analog signal is encoded by the relative density of ones (pulses) in the bit stream. Therefore taking the local mean around a sample gives the approximate amplitude of the encoded signal. An example of an analog signal and the corresponding PDM signal is shown in Figure 2.8

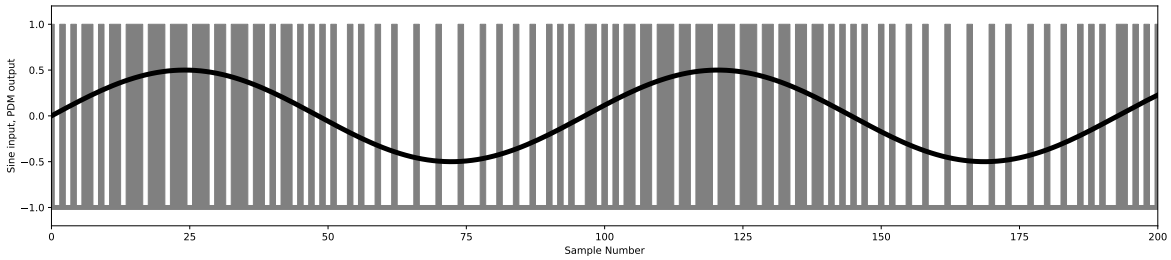


Figure 2.8.: PDM signal of a sine wave.

2.3.1. PDM to PCM Conversion

Pulse Coded Modulation (PCM) is the standard encoding of signals in most signal processing applications where each sample is encoded as a digital number representing the amplitude of the signal. To understand the conversion of the PDM signal into a pulse coded modulation (PCM) signal the spectrum of a PDM signal is useful. Figure 2.9 shows this spectrum of an encoded sine wave. The PDM signal is oversampled with a factor of 32. The actual sampling rate in relation to the normalized oversampled frequency is therefore about 0.03. As seen in Figure 2.9 the spectrum below the normalized frequency 0.03 has a very low noise floor and includes the encoded sine wave. The spectrum above 0.03 on the other hand is very noisy. As this part of the spectrum needs to be removed before reducing the sampling rate to the original sampling rate in order to avoid aliasing, a low pass filter needs to be applied.

Therefore converting a PDM to a PCM signal is the combination of a low pass filter and a sample rate reduction.

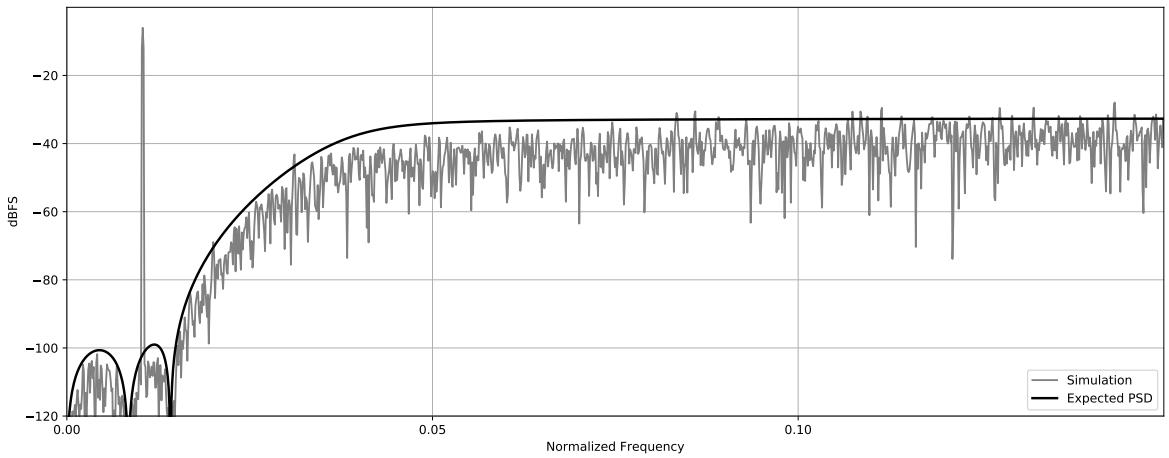


Figure 2.9.: Simulated PDM spectrum of a sine wave.

3. USB

This chapter contains information about the USB protocol as used by the sound card. Most of this information is extracted from the *Universal Serial Bus Specification Revision 2.0* [4], the *Universal Serial Bus Device Class Definition for Audio Devices Release 2.0* [5] and the *Universal Serial Bus Device Class Definition for Audio Data Formats Release 2.0* [6]. This information is split into four parts:

1. The different USB versions and resulting speeds can be found in Chapter **3.1. USB Versions & Speed Modes**.
2. Chapter **3.2. USB Addresses & Endpoints** and Chapter **3.4. USB Communication** introduces the addressing mechanisms, USB packets and USB transfers and explains how a USB connection is established.
3. The virtual structure of a USB device is explained theoretically in Chapter **3.3. USB Device Classes, Configurations & Interfaces**. The device specific structure and its parameters is stored in so called *Descriptors* inside every USB device. The different types of *Descriptors* are introduced in Chapter **3.5. USB descriptors**.
4. Finally Chapter **3.6. USB audio** explains some non standard USB mechanisms which are specific to the USB audio class.

3.1. USB Versions & Speed Modes

As USB is an evolving protocol that has been around for years and over time it has become faster and feature-richer. To ensure compatibility between all available USB devices newer versions are designed to be backwards compatible with older ones. To determine which version to use upon connecting a device to a host USB version codes are used. Most new versions also introduced faster transfere speeds to the USB standard.

Table 3.1 shows the relation between USB speed modes, their speeds and the corresponding minimal USB version.

Mode	Abbr.	Speed		Version
Low Speed	LS	1.5 Mbit/s	187.5 KB/s	USB 1.0
Full Speed	FS	12 Mbit/s	1.5 MB/s	USB 1.0
High Speed	HS	480 Mbit/s	60 MB/s	USB 2.0
SuperSpeed	SS	5 Gbit/s	625 MB/s	USB 3.0
SuperSpeed+	SS+	10 Gbit/s	1.25 GB/s	USB 3.1
SuperSpeed+	SS+	20 Gbit/s	2.5 GB/s	USB 3.2

Table 3.1.: USB speed modes.

Considerations for this Thesis

In order to calculate the minimum speed mode for the Sound Card design, the following aspects need to be considered.

The audio input consists of 16 microphones / audio channels at 48kHz with 24 bits per sample. The input bit rate R_{in} is calculated as

$$R_{in} = 16 \cdot 48kHz \cdot 24 = 18,432Mbit/s. \tag{3.1}$$

The audio output is stereo with 48kHz and 24 bits per sample as well. The output bitrate R_{out} is calculated the same way as the input bit range, i.e.,

$$R_{out} = 2 \cdot 48kHz \cdot 24 = 2,304Mbit/s \tag{3.2}$$

The overall audio bit range R_{audio} is the sum of input and output bit range, i.e.,

$$R_{audio} = speed_{in} + speed_{out} = 20,736Mbit/s = 2,592Mbyte/s \tag{3.3}$$

With the result of Equation 3.2 and referencing Table 3.1 the minimum speed mode for the Sound Card has to be USB High Speed (HS) with a maximum of 480 Mbit/s. Therefore USB version 2.0 has to be used.

Since the maximum speed of USB High Speed is more than 10 times faster than required by the raw audio data it is not necessary to take USB data overhead into account.

3.2. USB Addresses & Endpoints

The USB protocol is comparable to the IP protocol in networks. All devices have an address so the host can direct data to the right device. They also have different endpoints which are comparable to ports of the IP protocol. The difference to the IP protocol is that clients can only talk to the host and therefore the host has no address and no endpoints. All communication is initiated by the host. Each endpoint has its own buffers (in soft- or hardware) on the device and on the host side which means that the data streams for each endpoint are independent from each other. Figure 3.1 shows the use of addresses and endpoints.

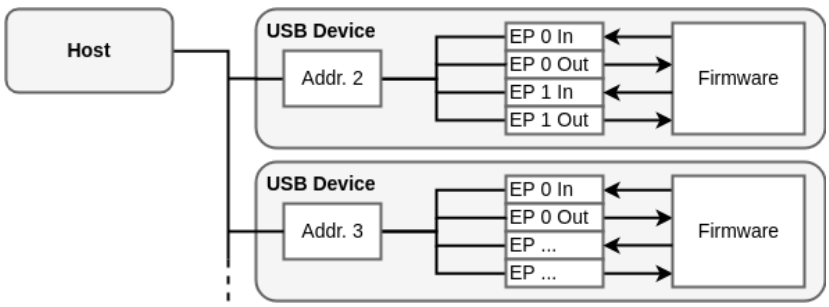


Figure 3.1.: USB data flow control through address and endpoints.

There is one special endpoint each device has to have which is endpoint zero. This is used for *Control Transfers* (see Chapter 3.4.2) and is also the first endpoint used for initiating the comunication with the device (see Chapter 3.4.4).

3.3. USB Device Classes, Configurations & Interfaces

There are many different USB devices on the market and most of them can be used the Plug'n'Play way. For that to work the USB host device needs to know which endpoints are used, how to configure the device and how to communicate through the used endpoints. To structure all that information the USB standard uses a system of device classes, configurations, and interfaces. This system is defined by USB descriptors (see Chapter 3.5) in the device which the host uses to identify all functions and all configuration options for a USB device. Figure 3.2 shows the virtual structure of a USB device in form of a tree. Details are presented in the following subchapters

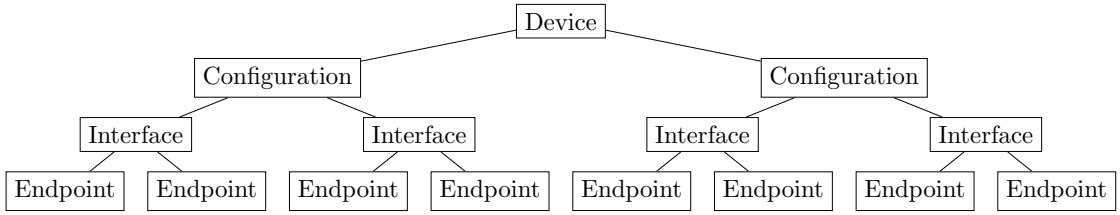


Figure 3.2.: USB device tree.

3.3.1. Device Classes

USB devices are grouped into classes which define the overall purpose of the device and characterize its capabilities.

Some classes are:

- Human Interface Device (HID)
  Everything a human can interact with related to a computer (keyboard, mouse, drawing tablet, touchscreen, display, joystick, ...).
- Audio Class
  This class is specifically for audio input and output. Also for all devices that control audio directly or indirectly (eg. mixers, volume control, ...).
- Mass Storage
  All devices where the host can directly read/written to the storage.
- Printer Class
- Video Class
- ...

Each physical device can have one or more device classes. Therefore a sound card could be an audio device and a human interface device at once if there are buttons on the sound card. If a device has only one class the class is defined for the whole device in the *Device*

Descriptor (see chapter Chapter 3.5). But if a device has multiple classes then each interface (see Chapter 3.5.4) can have its own class.

3.3.2. Interfaces

Each interface represents one 'function' a USB device can have. For an audio device this could be 3 functions for example: an audio input stream interface, an audio output stream interface and an audio control interface. The first two are the actual audio data and the last one would be used to control volume or mute the audio input or output. Each interface can contain an endpoint that is used to communicate with this interface (in this case an audio stream). If an interface needs bidirectional communication it will need two endpoints.

Alternate Settings

Sometimes interfaces can have different modes of operation. For example: a sound card could have a stereo and a Dolby Surround mode. In this case the interface is defined twice with the same interface ID but a different alternate settings ID (see Chapter 3.5.4). Then the USB host knows that it can switch the interface between these two interface definitions.

3.3.3. Configurations

If the alternate settings of interfaces are not flexible enough for some applications there are also configurations. A configuration is a collection of interfaces. Each configuration can define its own interfaces but they can also have interfaces in common (which have to be defined in each configuration separately). Each configuration defines one mode of operation for the whole device and there can only be one active configuration at a time. The USB host can decide which configuration to use but will normally choose the default configuration. The decision is in most cases based on the power consumption, speed or bandwidth needs of the device.

3.4. USB Communication

The USB communication is performed by sending packets. Since USB is bidirectional with only one physical differential data pair, every communication is always initiated by the host. Therefore there are some packets which only the host is allowed to send. Clients are only allowed to answer the host to avoid collisions on the bus. The start and end of every packet is signaled by two unique events on the differential pair of wires which cannot be mistaken for normal data. These are called start and end of packet events. Therefore the packet length can be variable and does not need to be known by the receiving party.

3.4.1. Packet Types

USB has 4 different packet types. Each of these packet types have sub-types which are listed in Table 3.2. These are identified by the packet identifier byte (PID) which is the first byte of every packet.

Type	Name	PID
Token	SOF	$0xA5$
	SETUP	$0x2D$
	OUT	$0xE1$
	IN	$0x69$
Data	DATA0	$0xC3$
	DATA1	$0x4B$
	DATA2	$0x87$
	MDATA	$0x0F$

Type	Name	PID
Handshake	ACK	$0xD2$
	NAK	$0x5A$
	STALL	$0x1E$
	NYET	$0x96$
Special	PRE	$0x3C$
	SPLIT	$0x78$
	ERR	$0x3C$
	PING	$0xB4$

Table 3.2.: Packet types and their PID used by USB.

The Token SOF (Start Of Frame) packet is sent at the beginning of each frame. A new frame starts every 1ms for USB full speed and lower and every $125\mu s$ for USB high speed and above. These are used for scheduling of time slots of all USB devices. The frame number is counted up every millisecond and loops around to zero when it overflows. Since USB high speed was introduced there are 8 SOF tokens sent with the same frame number (as there are 8 frames of $125\mu s$ in one millisecond). These frames are important for various transfer types (see Chapter 3.4.2).

8 bits	11 bits	5 bits
PID	frame number	CRC5

Figure 3.3.: Structure of a *Token Start Of Frame* packet.

The Token SETUP packet is used to initiate a *Control Transfer* (see Chapter 3.4.2). It is normally followed by a *Data* packet containing a request (see 3.4.3). The structure of this packet can be seen in Figure 3.4.

8 bits	7 bits	4 bits	5 bits
PID	address	endpoint number (containing IN/OUT bit)	CRC5

Figure 3.4.: Structure of a Token IN, OUT or SETUP packet.

The Token IN/OUT packet initiates a data transfer either from or to the device. Normally, after the *Token IN* packet follows a *Data* packet from the device to the host and after the *Token OUT* packet a *Data* packet from the host to the device. The structore is the same as for the SETUP packet seen in Figure 3.4.

The Data packet only contains data and has to always be preceded by a *Token SETUP*, *Token IN* or *Token OUT* packet for address and endpoint information. The data it contains depends on the transfer type and the application. The structure of the *Data* packet is shown in Figure 3.5.

The DATA0 and DATA1 packets are used for error detection and are alternately sent. This is needed because the host sends the same packet again if it does not get a *Handshake ACK* packet to acknowledge the previous *Data* packet.
In order to ilustrate this, an example is provided:

> Suppose the host sends a DATA0 packet which the device receives and then sends a ACK packet back. The device now expects the next *Data* packet. But if the ACK packet was lost and the host does not get this ACK it assumes there was an error receiving the previous packet and sends the same packet again. Without the alternating DATA0 and DATA1 packets the device would now assume that this packet is the new packet - which is wrong and would result in a catastrophic error.
> But due to the alternating packets the device now expects a DATA1 packet. The host on the other side sends the same packet again which was a DATA0 packet. Upon receiving the DATA0 packet the device knows that this packet is not the next packet but the same packet again and can send an ACK to the host again. This continues until the host receives the ACK correctly and then sends the next packet which is the DATA1 packet the device expects.

There are 2 other types of *Data* packets: DATA2 and MDATA. These have been necessary since the release of USB version 2.0 allows for *Isochronous Transfers* (see Chapter 3.4.2). Before USB version 2.0 there were only up to 2 packets per frame allowed but this has changed to up to 3 packets per (micro) frame.
Since *Isochronous Transfers* don't use *Handshake* packets each packet needs to be identifiable. Therefore these new *Data* packet types were added. The packet sequencing for IN and OUT *Isochronous Transfers* for 1 to 3 packets per (micro) frame is shown in Table 3.3.

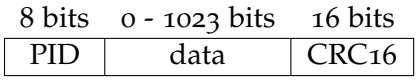


Figure 3.5.: Structure of a *Data* packet.

Direction	# of packets	PID Sequence		
		Pkt 1	Pkt 2	Pkt 3
IN	1	DATA0		
	2	DATA1	DATA0	
	3	DATA2	DATA1	DATA0
OUT	1	DATA0		
	2	MDATA	DATA1	
	3	MDATA	MDATA	DATA2

Table 3.3.: *Data* packet sequencing for *Isochronous Transfers*.

The Handshake packet is used for feedback on preceding packets. The structure of the *Handshake* packet is shown in Figure 3.6.

A *Handshake ACK* (acknowledge) indicates that the previous packet was received correctly.

A *Handshake NAK* (not acknowledge) indicates a communication error on the previously sent packet.

A *Handshake STALL* packet reports that the device is currently not able to send or receive anything. After a setup packet it has a special meaning: it indicates that the device does not support the hosts request.

The *Handshake NYET* (not yet) is a special packet that is used for Bulk OUT Transfers. It indicates that the buffer of the device is full and cannot receive any more data. The host then has to poll the device with PING packets to determine if the buffer has enough space to receive data again (see Bulk Transfere in Chapter 3.4.2).

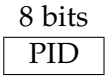


Figure 3.6.: Structure of a *Handshake* packet.

The PING packet is used for Bulk Transfers and is sent from the host to the device after the host has received a *Handshake NYET* packet from the device. This packet is used to determine if the device is able to receive Bulk Data or if its internal buffer is full. The device answers with a *Handshake ACK* if it can receive more data now or with a *Handshake NAK* if not (see Bulk Transfere in Chapter 3.4.2). The structure of the PING packet is shown in Figure 3.7.

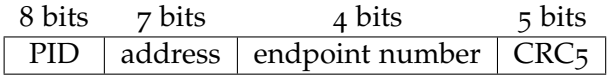


Figure 3.7.: Structure of a PING packet.

The PRE, SPLIT and ERR packets are used to enable lower speed communication on a higher speed bus. These packets are only used between USB hosts and USB hubs and are therefore not relevant for USB devices. This means that they are also not relevant for this thesis and therefore won't be explained in further detail here.

3.4.2. Transfer types

There are 4 data transfer types between host and client which are intended for different types of applications.

Control Transfer

This type of transfer is mostly used for requesting USB descriptors (see Chapter 3.5), configuring the device and for status operations. It is the first transfer mode used when a new device is added to the bus. Each *Control Transfer* usually contains a request which are explained in Chapter 3.4.3.

The *Control Transfer* can have up to three stages:

The Setup Stage (Figure 3.8) is where the host initiates the *Control Transfer* with a *Token SETUP* packet (see Chapter 3.4.1) followed by a *DATA*0 packet. The content of the *DATA*0 packet is a *Setup Packet* (see Chapter 3.4.3) containing the request. This is then acknowledged by the device.

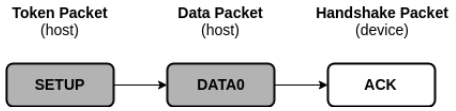


Figure 3.8.: Setup Stage of a USB *Control Transfer*.

The Data Stage (Figure 3.9) which is used for *Control Transfers / Requests* that need data to be transfered to or from the device. The amount of data transfered is defined in the previous stage. This stage consists of one or more *Data IN* or *Data OUT* transfers depending on the amount of data and the maximum packet size of the device.

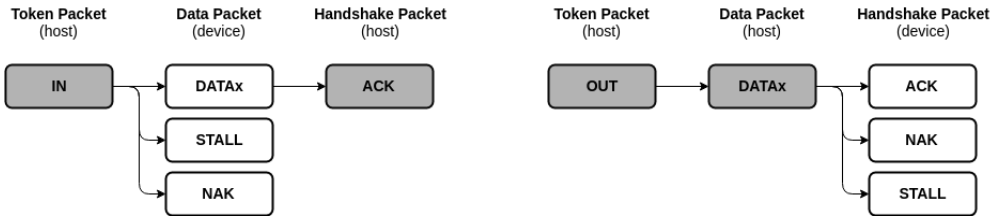


Figure 3.9.: Data Stage of a USB *Control Transfer* (left: dev-to-host, right: host-to-dev).

The Status Stage (Figure 3.10) is used to report the status (success or failure) of the whole *Control Transfer*. This stage starts with a *Token IN/OUT packet* in opposite direction of the *Data Stage*. A DATA0 packet with zero length acknowledges the successful reception of the data during the *Data Stage*.

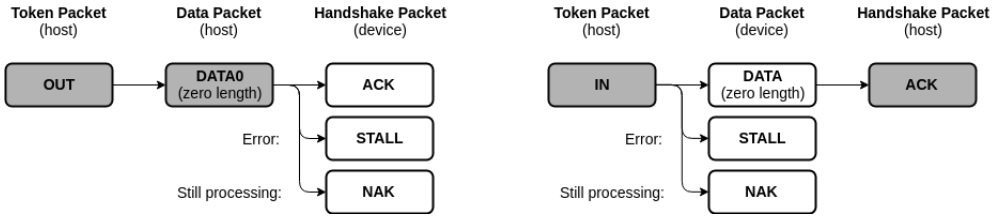


Figure 3.10.: Status Stage of a USB *Control Transfer* (left: dev-to-host, right: host-to-dev).

Isochronous Transfer

Isochronous Transfers are used for data that is constantly streaming. For this transfer type a time slot is reserved in every frame and therefore has guaranteed bandwidth.

The requested bandwidth is set in the *Endpoint Descriptor* (see Chapter 3.5.5). If there is not enough bandwidth left on the bus then the USB device is rejected upon starting (plugging in) or if there is an alternate setting in the interfaces with lower bandwidth the host uses that one. This can for example also be used as a fallback mode with lower quality audio.

In this transfer mode the host initiates a data transfer every frame. These data transfers have a checksum (as all *Data* packets have) but they are not acknowledged by the receiver. This means that a packet will not be resent if an error occurs. Therefore this transfer type should only be used if data loss can be tolerated such as in audio or video applications.

For real-time applications it is important to synchronize the data rate between host and device. This is necessary because no two clocks are at the same speed and the host has therefore a different length of a second than the device. To control the time basis of the data stream there are four possibilities:

No synchronization - this means that the data is not used for real time applications and so the data rate is not important.
Synchronous - in this case the host dictates the time. The device has to adjust its clock speed to the *Start Of Frame* packets (see Chapter 3.4.1) it receives from the host.
Asynchronous - this means that there is feedback on how full the receiver buffer is. The sender then automatically adjust its speed to keep the buffer from under- or overflowing. To send the feedback an additional endpoint with the opposite direction has to be defined as an isochronous feedback endpoint (see Chapter 3.5.5) and has to be linked to the original endpoint.
Adaptive - this mode means that the device is able to synchronize itself to the rate the data is send by the host.

Figure 3.11 shows the *Isochronous Transfer* which is not acknowledged by the receiving party.

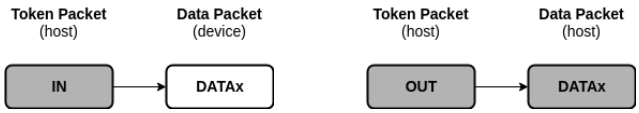


Figure 3.11.: USB *Isochronous Transfer* (left: dev-to-host, right: host-to-dev).

Interrupt Transfer

Interrupt Transfers are not interrupts in the original meaning of the word since all USB communication has to be initiated by the host and only host-to-device transfers are truly interrupts. To receive interrupts from the device the host has to poll the device periodically. The advantage of interrupt transfers is that there is a guaranteed latency. If a checksum

error occurs the transfer will be reattempted in the next period. The polling period can be defined in USB frames.

Interrupt Transfers are usually used for data that is not periodic, small in size and needs a low latency.

Figure 3.12 shows the structure of an Interrupt Transfer. It starts with a *Token IN* or *OUT* packet followed by a *Data* packet and a *Handshake* packet. If there is no data to transmit when the host polls the device with the *Token IN* packet then the device simply responds with a *Handshake NAK* packet.

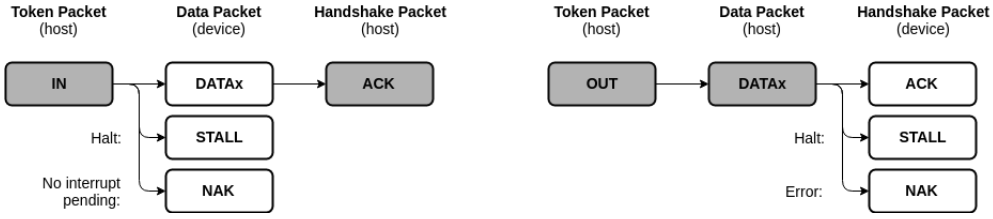


Figure 3.12.: USB Interrupt Transfer (left: dev-to-host, right: host-to-dev).

Bulk Transfer

Bulk Transfers are used for large data transfers which are not time critical but often come in bursts. Data is only transfered over the bus if there is spare time after Isochronous and Interrupt Transfers. Therefore there is neither a guarantee for latency nor for bandwidth.

This transfer type provides error detection and retransmission mechanisms to ensure that the data is received without errors and without any loss of data.

Due to the large amount of data that can be sent there is the possibility that an endpoint buffer of a device can run out of space. For this case there are special *Handshake* packets (see 3.4.1) used in bulk transfer:

> When the device buffer runs out of space in USB 1.x the device answers with a NAK and the host sends the data again. But if the device still has no space in the buffer this can go on and on for a long time thus blocking the bus. Since USB 2.0 the device now sends a NYET packet instead of the NAK. This tells the host that there is no more space in the devices buffer and the host does not try to send the next Bulk OUT Transfer. Instead the host sends a PING packet. If the device still has no space it responds with a NAK. This means that the host does not send the next *Data* packet until the device responds to the next PING with an ACK indicating that the buffer has enough space again. Therefore the bus is not blocked by the attempts of sending data. This is illustrated in a state diagram in Figure 3.14.

The structure of a Bulk Transfer is shown in Figure 3.13. First the host sends a *Token IN/OUT* packet followed by one or more *Data* packets from or to the device. The detection of the last packet is either done by sending a *Data* packet shorter than the maximum endpoint buffer size, by sending a zero length packet or by sending the maximum allowed data for this bulk

endpoint. Afterwards the receiving party has to acknowledge the transfer with a *Handshake* packet. As described above this can also be a *Handshake NYET* packet.

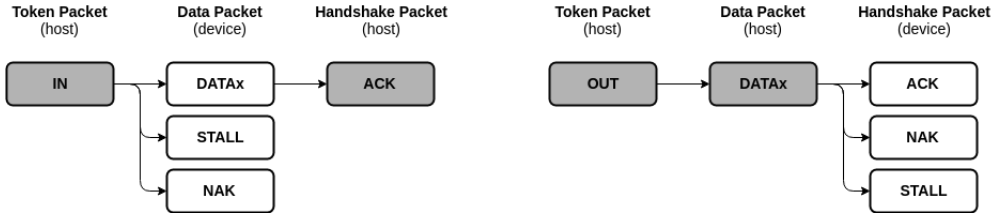


Figure 3.13.: USB Bulk Transfer (left: dev-to-host, right: host-to-dev).

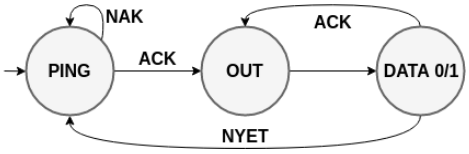


Figure 3.14.: USB PING and NYET packets.

3.4.3. Requests

USB Requests are always sent over the default endpoint zero and are *Control Transfers*. They are used to get information about the device (e.g. requesting USB descriptors), configuring the device or checking the status of the device. There are standard requests that every device has to implement and there are class specific requests which are implemented by different device classes. For example: An audio class request could be changing the sample rate of the audio input stream. There are also vendor specific requests which are not specified anywhere and can be used for custom purposes.

A request starts like all *Control Transfers* with the *Setup Stage*. The DATA0 packet of this stage contains a Setup Packet containing the request information. If the request needs data to be transfered then there follows a Data Stage. For details on the *Control Transfer* see Chapter 3.4.2.

Setup Packet

Table 3.4 shows the structure of a Setup Packet holding all information about the request.

Offset	Field name	Bytes	Description
0	bmRequestType	1	Direction, Type and Recipient.
1	bRequest	1	ID of the request.
2	wValue	2	A value (request dependent).
4	wIndex	2	A index (request dependent).
6	wLength	2	Length of the data in the data stage (if any).

Table 3.4.: The format of the Setup Packet for USB Requests.

The field **bmRequestType** is bit coded in the following way:

- Bit 7: Transfer direction of the data phase (OUT: 0, IN: 1)
- Bit 6 to 5: Type (Standard: 0, Class: 1, Vendor: 2)
- Bit 4 to 0: Recipient (Device: 0, Interface: 1, Endpoint: 2, Other:3)

The other fields are request dependent and illustrated in Table 3.5.

Standard Requests

Table 3.5 shows all standard requests of the USB 2.0 protocol. Some requests have two variants: a get and a set request. These requests have two different request identifiers and also need to have the direction bit in the field *bmRequestType* set appropriately. The *recipient* bit has to be set according to Table 3.5. For all standard requests the *type* bit is set to zero.

bRequest		Recipient	wValue	wIndex	Data
0:	Get Status	Device Interface Endpoint	-	- Interface Endpoint	Status
1/3:	Clear/Set Feature	Device Interface Endpoint	Feature	- Interface Endpoint	-
5:	Set Address	Device	Address	-	-
6/7:	Get/Set Descriptor	Device	Type & Index	- / Language	Descriptor
8/9:	Get/Set Configuration	Device	-	-	Config. Value
10/11:	Get/Set Interface	Interface	Alternate Setting	Interface	-
12:	SYNCH_FRAME	Endpoint	-	Endpoint	Frame Nr.

Table 3.5.: USB Standard Requests with their recipients, parameters and data.

The Get Status request is used to retrieve a status of the device, interface or endpoint. It always returns two bytes where each bit represents a boolean status. For the device, bit 0 is *Self Powered* and bit 1 is *Remote Wakeup*. For an endpoint bit 0 is *Halted* and bit 1 is *Stalled*. An interface does not return any status at this point and its return value is reserved for future use.

The Clear/Set Feature request is used to set or clear boolean features. Directed to the device it can set the *Remote Wakeup* feature or put the device into a test mode. The only endpoint feature is to halt an endpoint. The interface does not implement any features.

The Set Address request sets the address of a USB device and is used while initiating communication after plugging in a new device (see Chapter 3.4.4).

The Get/Set Descriptor request is also used in the initiation phase to request descriptors. The Set Descriptor request is usually not used during normal operation.

The Get/Set Configuration request is used for setting a *Configuration Descriptor* as the current configuration. The get request returns the number of the currently active *Configuration Descriptor*.

The Get/Set Interface request switches between different alternate settings of a specific interface.

The SYNCH_FRAME request is for isochronous endpoints where the size of frames vary. Varying the size of frames is done in patterns (e.g. 2,2,3) where the frame number of the start of this pattern is returned by this request.

3.4.4. Initiating the communication

The communication starts when the host (or USB hub) detects the connection of a device by measuring the change in voltage caused by the USB devices pull-up resistors on the data lines. The USB host then sends a reset command to the physical port from where it detected the connection which sets the device address temporarily to zero. Therefore the host can only reset one device at a time to ensure that there is never more than one device with the address zero.
Now that the device is addressable communication with the device can start. At this point the device acts as a basic USB device with no specific function. This means that it is only allowed to answer to basic USB control requests on endpoint 0 since this is the reserved endpoint for USB control requests.
The host then requests the *Device Descriptor* (see Chapter 3.5.1) of the USB device in order to find out the maximum packet length (receive buffer length of the device). The host is not allowed to send packets longer than this number since the device cannot handle it. After this the device is reset again and the host sends a set address request which assigns an unused address to the USB device.
At this point the device has a unique address and the host is free to start the same procedure for other newly attached devices.
Now the actual initialization and configuration of the device starts. The host now requests all the USB descriptors (the *Device Descriptor*, all *Configuration Descriptors* and all *String Descriptors*). Then the host chooses a configuration (see Chapter 3.3) and sends a configuration request to the device which in turn activates the selected configuration.

3.5. USB descriptors

When a USB device starts to communicate with the USB host the USB descriptors are requested by the host (see Chapter 3.4). These descriptors consist of a hierarchical system of sub-descriptors and also a list of strings which can be referenced in other descriptors by their index inside. The general format of a USB descriptor is depicted in Table 3.6. Each descriptor has the same two fields at the beginning: the *bLength* field, which is the length in bytes of the whole descriptor (starting at and including the *bLength* field) and the field *bDescriptorType* which defines the type of the descriptor. After the first two bytes starts the descriptor specific part which is different for each type of descriptor.

Offset	Field name	Bytes	Description
0	bLength	1	Size of the descriptor in Bytes
1	bDescriptorType	1	Descriptor type
⋮	⋮	⋮	Descriptor specific

Table 3.6.: The common format of all USB descriptors.

The root element of the USB descriptor hierarchy is the *Device Descriptor*.

3.5.1. Device Descriptor

The *Device Descriptor* contains information about the whole device like USB version, manufacturer, serial number, ... Table 3.7 shows the structure of the *Device Descriptor*.

Offset	Field name	Bytes	Description
0	bLength	1	Device Desc. is 18 bytes long.
1	bDescriptionType	1	1 (*Device Descriptor* ID)
2	bcdUSB	2	USB version
4	bDeviceClass	1	Device Class Code
5	bDeviceSubClass	1	Subclass Code
6	bDeviceProtocol	1	Protocol Code
7	bMaxPacketSize	1	Max. Packet Size (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	Num. of configurations

Table 3.7.: The format of the *Device Descriptor*.

- **bLength** is the descriptor length and is always 18 bytes long.
- **bDescriptionType** is set to 1. It identifies the descriptor as *Device Descriptor*.
- **bcdUSB** is the USB version in the format $0xJJMN$ where J is the major, M the minor and N the sub minor version.
- **bDeviceClass** specifies the class for the whole device.
  If equal to $0x00$, each interface specifies its own class code.
  If equal to $0xFF$, the class code is vendor specified.
  If equal to $0xFE$ the interface defines the class and there is an *Interface Association Descriptor* present (see Chapter 3.7).
  Otherwise the value is a valid class code.
- **bDeviceSubClass & bDeviceProtocol** are used to select the correct driver for the device. Normally they are only defined at interface level.
- **bMaxPacketSize** is the maximum packet size the endpoint zero buffer in the device can hold.

- **idVendor & iProduct** uniquely identify the product. Vendor IDs are assigned by the USB Implementers Forum.
- **bcdDevice** is the device version number and has the same format as bcdUSB.
- **iManufacturer, iProduct & iSerialNumber** are human readable strings to describe the device.
- **bNumConfigurations** is the number of configurations this device has. This is then used to request all *Configuration Descriptors*.

3.5.2. String Descriptor

Each string used in any other descriptor is referenced by an ID and stored in a *String Descriptor*. Each *String Descriptor* can be requested by the host with a *String Descriptor Request* along with the ID of the requested *String Descriptor* and a language ID (see Chapter 3.4.3).The device then returns a *String Descriptor* as shown in Table 3.8.

The available language IDs can be found in the special *String Descriptor* with ID 0. This *String Descriptor* has a different layout to the others which is shown in Table 3.9. The first language with index 1 is advised to be English as there might be problems with some operating systems otherwise.

All strings used in any descriptor are not mandatory for USB to function as their purpose is to provide a human readable string.

Offset	Field name	Bytes	Description
0	bLength	1	String Desc. length.
1	bDescriptionType	1	3 (*String Descriptor* ID)
2	bString	2	string encoded in unicode

Table 3.8.: The format of the *String Descriptors* (id starting with 1).

Offset	Field name	Bytes	Description
0	bLength	1	String Desc. length.
1	bDescriptionType	1	3 (*String Descriptor* ID)
2	wLANGID[0]	2	LANGID Code 0
...	...	...	...
2 + 2·x	wLANGID[x]	2	LANGID Code x

Table 3.9.: The format of the language *String Descriptor* (with id 0).

- **bLength** is the length of the *String Descriptor*. It varies depending on the string length or in case of the *String Descriptor* with ID 0 depending on how many languages are defined.
- **bDescriptionType** is set to 3. It identifies this descriptor as a *String Descriptor*.
- **bString** are the actual bytes of the string encoded in unicode. The size may vary from string to string.
- **wLANGID[x]** contains a language id code in 2 bytes. For example *English (United States)* has the id 0x0409 in little endian representation.

3.5.3. Configuration Descriptor

The *Configuration Descriptor* represents one possible configuration of the device (see Chapter 3.3). How many possible configurations there are is defined in the field *bNumConfigurations* in the *Device Descriptor*. For the host to request a *Configuration Descriptor* an index lower than *bNumConfigurations* needs to be sent with the request.

This descriptor contains also all child descriptors in the descriptor tree (see Figure 3.15). That means by requesting the *Configuration Descriptor* with length *wTotalLength* (see below) the host gets the whole tree of descriptors except *Device Descriptor* and *String Descriptors*. Figure 3.15 shows the descriptor hierarchy - everything returned by a *Configuration Descriptor* request with given index length is shown on the right.

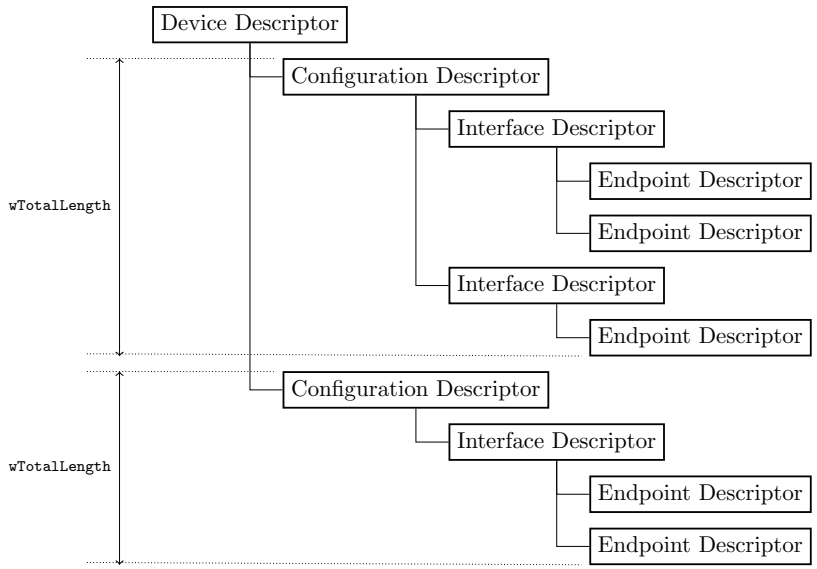


Figure 3.15.: Total length of a *Configuration Descriptors* defined in the field wTotalLength.

Offset	Field name	Bytes	Description
0	bLength	1	Length of Conf. Desc.
1	bDescriptionType	1	2 (*Config. Descriptor* ID)
2	wTotalLength	2	Length including subsecriptors
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 ($\times 2mA$)

Table 3.10.: The format of the *Configuration Descriptors*.

- **bLength** is the length of the *Configuration Descriptor* only and is set to 8 bytes in newer version of USB. By requesting this amount of data neither *Interface Descriptors* nor other descriptors in the hierarchy will be returned. To request all descriptors (direct and indirect children) the number of bytes in *wTotalLength* needs to be requested.

- **bDescriptionType** is set to 2. It identifies this descriptor as a *Configuration Descriptor*.
- **wTotalLength** is the total length of all data including this *Configuration Descriptor* and all appended descriptors (*Interface Descriptors*, endpoint descriptors, class specific descriptors, ...) which are direct children of this descriptor or further down the descriptor hierarchy.
- **bNumInterfaces** is the number of interfaces defined in this configuration.
- **bConfigurationValue** is the index used to retrieve this *Configuration Descriptor* and to pass along with the *set configuration* command to select this configuration.
- **iConfiguration** provides a human readable string describing this configuration.
- **bmAttributes** is one byte long where each bit represents a boolean flag.
  - Bit 7 was used to indicate a bus powered device in USB 1.0. For higher USB versions this should be set to 1.
  - Bit 6 indicates a self powered device.
  - Bit 5 is enabled if the device can wake the host from a sleep state.
  - Bits 4 to 0 are reserved and should be 0.
- **bMaxPower** is the maximum power consumption of the device in this configuration. This is one criterion for the host to choose a configuration. Since the USB specification only allows a maximum of $500mA$ one byte with $2mA$ per unit ($250\widehat{=}500mA$) is sufficient.

3.5.4. Interface Descriptor

This descriptor is part of the *Configuration Descriptor* hierarchy. It is returned along with the *Configuration Descriptor* after the host has issued a *Configuration Descriptor Request* with sufficient length (see *wTotalLength* in *Configuration Descriptor*). An *Interface Descriptor* defines one of the interfaces in a configuration. If this interface has alternate settings there needs to be an *Interface Descriptor* for each alternate setting with the same interface number.

Offset	Field name	Bytes	Description
0	bLength	1	The Interface Desc. is 9 bytes long.
1	bDescriptorType	1	4 (*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 3.11.: The format of the *Interface Descriptors*.

- **bLength** is the length of the *Interface Descriptor* which should be 9 bytes long.
- **bDescriptionType** is set to 4. It identifies this descriptor as an *Interface Descriptor*.
- **bInterfaceNumber** is the number of the interface this *Interface Descriptor* describes. As interfaces can have multiple alternate settings (see Chapter 3.3.2) there can be multiple *Interface Descriptors* that describe the same interface (with the same number) but they have to differ in *bAlternateSetting*.

- **bAlternateSetting** is the number of the alternate setting which this *Interface Descriptor* describes for the interface with number *bInterfaceNumber* (see *bInterfaceNumber*).
- **bNumEndpoints** is the number of endpoints this alternate setting for the interface with number *bInterfaceNumber* uses. It is therefore the number of endpoint descriptors this descriptor will be followed by.
- **bInterfaceClass** is the class code for this interface (see Chapter 3.3). As mentioned before each interface can have its own class (e.g. one interface with audio class and one with human interface device class). If the property *iDeviceClass* in the *Device Descriptor* is a valid class code than this should be the same.
- **bInterfaceSubClass** is the subclass of the selected interface class (see Chapter 3.3). This should match *iDeviceSubClass* in the *Device Descriptor* if the *Device Descriptor* defines the class.
- **bInterfaceProtocol** is a code for the protocol used. It depends on the selected class and is specified in the USB standard for each of the valid classes (see Chapter 3.3).

3.5.5. Endpoint Descriptor

Endpoint descriptors are also part of the configuration hierarchy. Same as the *Interface Descriptor* they are returned with the *Configuration Descriptor*. In the hierarchy they are children of the *Interface Descriptor* which can have any number of endpoints. Audio class specific descriptors which will be introduced in Chapter 3.6 can be siblings of this descriptor and can also be its children.

Offset	Field name	Bytes	Description
0	bLength	1	The Endpoint Desc. is 7 bytes long.
1	bDescriptorType	1	5 (*Endpoint Descriptor* ID)
2	bEndpointAddress	1	Address and data direction.
3	bmAttributes	1	Transfer type (and Iso types)
4	wMaxPacketSize	1	Maximum packet size
5	bInterval	1	Polling interval

Table 3.12.: The format of the *Interface Descriptors*.

- **bLength** is the length of the endpoint descriptor which should be 7 bytes long.
- **bDescriptionType** is set to 4. It identifies this descriptor as an endpoint descriptor.
- **bEndpointAddress** contains the number and the data direction of the endpoint.
  - Bit 7 is the direction 0 = Out, 1 = In
  - Bits 6 to 4 are reserved, set to 0.
  - Bits 3 to 0 are the endpoint number.
- **bmAttributes** contains the transfer type of the endpoint. For isochronous endpoints it also includes the synchronisation and usage types (see 3.4.2). If the transfer mode is not isochronous usage type and synchronisation type bit should be set to 0.
  - Bits 7 to 6 are reserved, set to 0.
  - Bits 5 to 4 are the Usage Type (in isochronous Mode)
    0 = Data Endp., 1 = Feedback Endp., 2 = Explicit Feedback Data Endp.

  - Bits 3 to 2 are the Synchronisation Type (in isochronous Mode)
    0 = No Synchonisation, 1 = Asynchronous, 2 = Adaptive, 3 = Synchronous.
  - Bits 1 to 0 are the Transfer Type
    0 = Control, 1 = Isochronous, 2 = Bulk, 3 = Interrupt.
- **wMaxPacketSize** defines the maximum size of a packet that can be sent to or from this endpoint. This has to do with the internal buffer size of the USB device.
- **bInterval** is the polling interval expressed in frames which means that every *bInterval* frames the USB host will ask the USB client if there is an interrupt pending on this endpoint.
  For control and bulk endpoints this has to be 0 since they are not polled periodically.
  For Isochronous endpoints this has to be 1 since they are streaming and can adjust the amount of data for each frame to match their desired data rate.
  For Interrupt endpoints this can be any value between 1 and 255. Interrupts then can only occur every *bInterval* frames.

3.6. USB audio

This chapter is going to introduce audio class specific requests and interfaces that are used by the sound card. It will also briefly explain the audio streaming and control mechanisms.

For this project the audio part of the USB protocol consists of two main features.

1. The Audio Streaming Interface responsible for sending and receiving of PCM audio streams via asynchronous *Isochronous Transfers* (see Chapter 3.4.2). This is fairly simple as it sends or receives the raw PCM samples over an endpoint defined in an *Interface Descriptor* of class *Audio Streaming*. As this is an asynchronous endpoint there is also a second feedback endpoint over which it sends the desired data rate changes to match the sampling rate of the device.

2. The Audio Control Device Interface contains a number of values (referred to as control values) which control the behavior of the sound card. These can be for example a value for the sample rate or a value for the volume. These values are defined in *AudioControl Descriptors* that are grouped into different types which represent functional parts of the device. *AudioControl Requests* are used to read or write these values.

Figure 3.16 shows the structure of a USB Audio device.

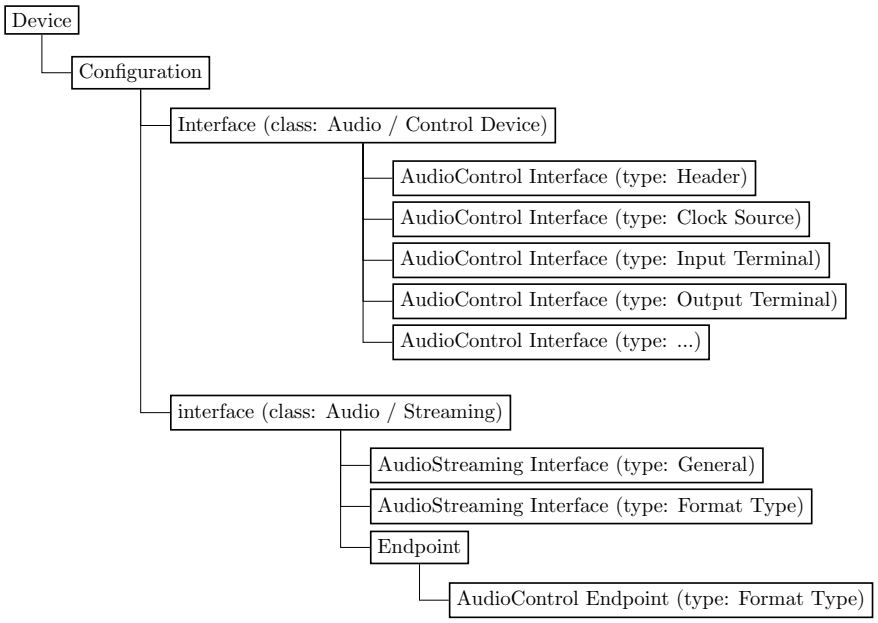


Figure 3.16.: Structure of a USB Audio device.

3.6.1. AudioControl Request

The *AudioControl Request* is a class specific request and follows the same rules as default requests introduced in Chapter 3.4.3. The purpose of *AudioControl Requests* is to control the audio device by changing certain control values of the device. Each used AudioControl descriptor in a device is assigned a unique ID to address requests to. Each control value inside a descriptor also has a defined number to be identifiable by a request. 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 zero as a normal *Control Transfer*.

The values in the fields of the requests *Setup Packet* (see Chapter 3.4.3) are set as follows:

- **bmRequestType:** Direction is set to IN/OUT according to get/set. Type is set to Class. Recipient is set to Interface.
- **bRequest** is 1 for a *Get/Set Current Value* request and 2 for a *Get Range* request.
- **wIndex** is split into a high and low byte. The high byte is the ID of the AudioControl descriptor to send the request to (see Chapter 3.6.2). The low byte is the interface number containing the *AudioControl Interface Descriptor*.
- **wValue** is also split into a high and low byte. The high byte is the control selector (CS) which is the number of the control value and the low byte is the audio channel number (to set or get control values for individual channels).

The data sent in the data stage of the control request is structured as follows: For a *Get/Set Current Value* request it is 1, 2 or 4 bytes containing only the value as a 8, 16, or 32 bit value. The number of bytes for each value can be found in **Chapter 3.6.2 AudioControl Interface Descriptors**. A *Get Range* request consists of a series of subranges and the first two bytes of

data are the number of 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. Therefore the total number of bytes for each subrange is 3, 6 or 12. dMIN is the lowest value alowed, dMAX the highest and dRES the step size between dMIN and dMAX. For example: if the data stage contains the values [2, 5, 11, 2, 20, 26, 3] then the valid values of this range are [5,7,9,11,20,23,26] (2 means two subranges, 5,11,2 means from 5 to 11 in steps of 2 and 20,26,3 means from 20 to 26 in steps of 3).

3.6.2. AudioControl Interface Descriptors

There are several *AudioControl Interface Descriptors* (see Figure 3.16) in the USB specification which define how to control different functions in an audio device. They are all grouped together in an *Interface descriptor* of class *Audio* and subclass *Control Device*. This *Interface Descriptor* and all *AudioControl Interface Descriptors* do not have any endpoints and only define which audio class specific requests the device listens to. All *AudioControl Interface Descriptors* have some fields in common and have mostly the same structure with different meaning of certain fields. Table 3.13 shows the general structure of all *AudioControl Interface Descriptors* except the *Header*.

Field name	Bytes	Description
bLength	1	Length of this descriptor.
bDescriptorType	1	36 (*AudioControl Interface Descriptor* ID)
bDescriptorSubtype	1	Subtype ID (Header, Clock Source, Feature Unit, ...)
bXxxID	1	Unique ID for this unit.
...	...	Subclass specific fields.
bmControls	2	Bitmap with control value definition.
iXxx	1	Index of string descriptor with the name of this unit.

Table 3.13.: The format of the *AudioControl Interface Descriptors*.

The field **bXxxID** represents a unique ID for every *AudioControl Interface Descriptor* and needs to be specified in a request in the high byte of the request field **wIndex**. Even though the name of this ID field is different in every *AudioControl Interface Descriptor* (illustrated by the **"Xxx"**) its purpose is always the same. Each *AudioControl Interface Descriptor* also implements the field **bmControls** which is a bitmap where every pair of two bits represent one control value (e.g. volume, clock speed, ...). These values can either be not present in the device ($0b00$ in the bitmap), be read-only ($0b01$) or read and writable ($0b11$). The control selector (CS) which is the number of each control value has to be specified in the high byte of the request field **wValue** in order to read or write this specific control value. The different control values implemented by this sound card, their place in the bitmap **bmControls** and their control selectors are listed for each *AudioControl Interface Descriptor* sub type in the following sections.

Header

The Header is the first child of the *Interface Descriptor* of class *Audio* subclass *Control Device* and has the subtype ID 1. It contains the version of the Audio Device Class in field **bcdADC**

encoded in the same way as the USB version in the *Device Descriptor* (see Chapter 3.5.1). There is also a field named **bCategory** containing a constant representing the primary use of this interface. The field **wTotalLength** is the total length of this and all following *AudioControl Interface Descriptors* which follows the same principle as field **wTotalLength** in the *Configuration Descriptor*. The sound card does not implement any control values of this descriptor.

Clock Source

This descriptor is used to control the audio clock and has subtype ID 10. The frequency control value sets the audio sample rate. The clock validity control is normally read-only and returns a boolean status. The sound card implements the control values listed in Table 3.14.

Bits	CS	Nr. of bytes	Control value name
1..0	1	4	Clock Frequency Control
3..2	2	1	Clock Validity Control

Table 3.14.: AudioControl Clock Source: control values, bits in bitmap, control selector (CS) and number of bytes.

Input Terminal & Output Terminal

The *AudioControl Interface Descriptor* of type Input / Output Terminal controls an audio input or an audio output signal. An input signal can be the signal from the USB host to the device or the signals from the microphones. Output signals are for example the signal from the device to the USB host or the signal to the headphone output. The sound card does not implement any control values of this descriptor.

The input and output descriptor contains the following subclass specific field:

- **bSourceID** is the ID of the *AudioControl Interface Descriptor* of type Clock Source which provides the sampling rate for this input or output terminal.

Additionally only the input terminal descriptor contains the following fields:

- **bNrChannels** is the number of input channels.
- **bmChannelConfig** specifies the spatial location of each channel. Each bit corresponds to a specific location and the order of the channels must correspond to the order of the bits in this bitmap. If there are more channels than bits set to 1 in this bitmap then the names of those channels are defined by the field iChannelNames. For this sound card there are no bits set in this bitmap as there use is not predefined.
- **iChannelNames** is the index of the first *String Descriptor* holding the name for the first channel whose location is not specified in bmChannelConfig. All further channels correspond to the *String Descriptor* indices following the index iChannelNames.

Feature Unit

The *AudioControl Interface Descriptor* of type Feature Unit controls mostly audio properties and has the subtype ID 6. The sound card implements the control values listed in Table 3.15. This descriptor contains the subclass specific field **bSourceID** which is the ID of the Input or Output Terminal to which the features in this descriptor apply.

Bits	CS	Nr. of bytes	Control value name
1..0	1	1	Mute Control
3..2	2	2	Volume Control

Table 3.15.: AudioControl Clock Source: control values, bits in bitmap, control selector (CS) and number of bytes.

3.6.3. AudioStreaming Descriptors

These descriptors are mainly for defining the audio format sent or received. They are children of the *Interface Descriptor* of class *Audio* subclass *Streaming* and split into subtypes. There are two subtypes used in this sound card, *general* and *format type*. The *general* descriptor defines the audio encoding used which in this case is PCM and also defines the number of audio channels for this interface. It also has an index of a *String Descriptor* which holds the first channel name. All subsequent *String Descriptors* hold the names for the rest of the channels. The *format type* defines the resolution of the samples in bits per sample (in this project normally 24) and how many bytes per sample are transferred (even though there are only 24 bits per sample there are 4 bytes transferred with the highest byte being 0).

3.7. Interface Association Descriptor

The *Interface Association Descriptor* (IAD) was later added to the USB 2.0 standard in the form of an Engineering Change Notice [7] along with a usage model document [8]. The purpose of this descriptor is to group *Interface Descriptors* together that logically belong to one function of the USB device. This also means that there only needs to be one device driver for each association of interfaces instead of one driver for each interface. Figure 3.17 illustrates this new behavior on the right side.

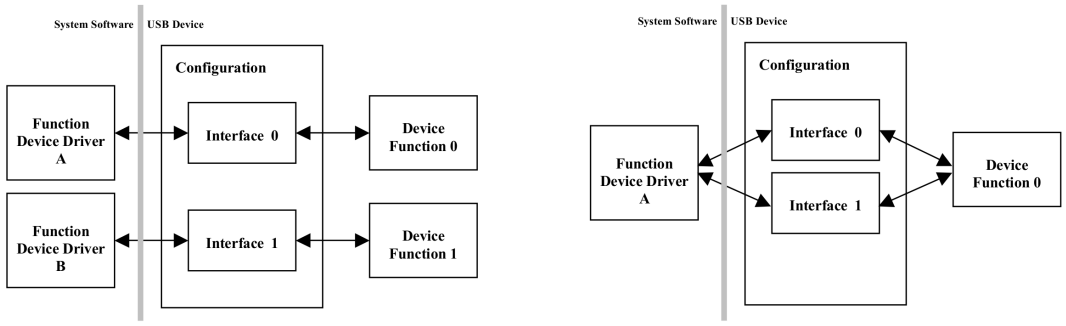


Figure 3.17.: Impact of the *Interface Association Descriptor* (left: without IAD, right: with IAD). [7]

The group is defined by the index of the first interface in the group along with the number of following interfaces (with sequential indices) also belonging to the same group. The class, subclass and protocol which for simple USB devices are defined in the *Device Descriptor* is now defined separately for each interface association. When using an *Interface Association Descriptor* the following fields in the *Device Descriptor* (see Chapter 3.5.1) have to be set to specific values: bDeviceClass: $0xEF$, bDeviceSubClass: $0x02$, bDeviceProtocol: $0x01$.

Field name	Bytes	Description
bLength	1	Length of this descriptor.
bDescriptorType	1	11 (*Interface Association Descriptor* ID)
bFirstInterface	1	The index of the first interface in this group.
bInterfaceCount	1	The total number interfaces in this group.
bFunctionClass	1	The class of this group.
bFunctionSubClass	1	The sub class of this group.
bFunctionProtocol	1	The protocol of this group.
iFunction	1	Index of string desc. with the name of this function.

Table 3.16.: The format of the *Interface Association Descriptor*.

4. Processor

The processor used for this thesis is a XUF216-512-TQ128 [9] of the xCORE-200 series from XMOS Ltd.

In this chapter the architecture of the processor is outlined and the programming language XC, used to program the xCORE processor family is explained. Finally two development tools, the *xTime Composer* which is an Integrated Development Environment(IDE) and the *xTAG 3* programmer are introduced.

4.1. Architecture

xCORE processors are multi-core processors that consists of multiple logical cores grouped into one or more physical tiles.

The block diagram of the XUF216 processor is illustrated in Figure 4.1.

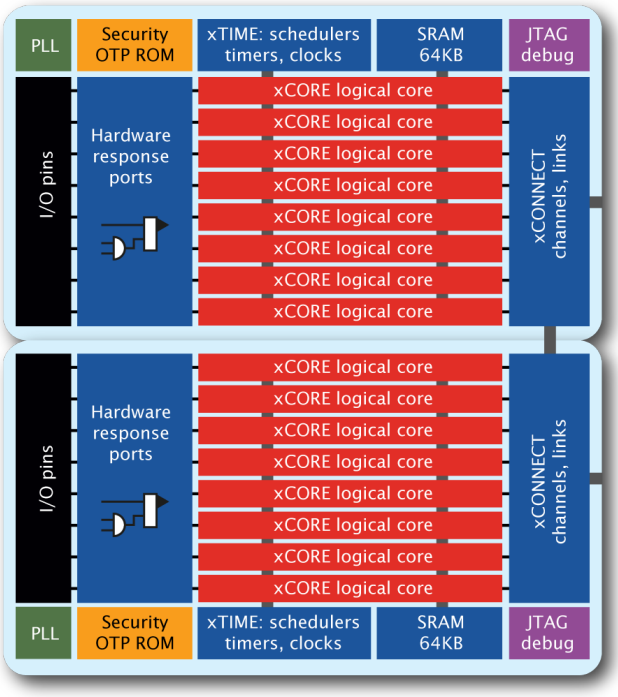


Figure 4.1.: Block diagram of the XUF216 processor. [10]

The processor consists of two **tiles** each containing 8 **logical cores**, integrated I/O and on-chip memory. There is also an **xTIME scheduler** on each tile which handles scheduling

and synchronizes events to remove the need of interrupt routines. The scheduler notifies threads on events triggered by timers, hardware I/O, channel communications and the like.

4.2. XC-Language

For easy and comfortable usage of the processors features, XMOS developed an extension to the C programming language called XC [11]. It introduces new keywords, operators and types and also includes a library with defines and functions to configure the processor. The most important features needed to understand the source code of this project are explained in this chapter. For more features of the XC language and the xCORE processors consult the *XMOS Programming Guide* [12].

4.2.1. Paralellism

Since the processor has multiple cores on two tiles there needs to be a way to run code in parallel and to assign code to a certain core / tile. Therefore the `par` keyword was introduced. This statement is followed by a block of functions where each function is run as a separate task on a separate core.

```
par {
  task1();
  task2();
}
```

The assignment to specific cores / tiles is done with the `on` keyword.

```
on tile[0]:         task1();
on tile[1].core[0]: task2();
```

4.2.2. Events

The XC language introduces event based programming to C. There are some functions or operators that are blocking and wait for an event to occur. Since it would normally not be possible to wait on two blocking functions at once the *select* keyword was introduced. It waits on multiple blocking events and when any of the specified events occur the corresponding code is executed. If a *select* is put inside an endless loop then the task only responds to events.

```
select {
  case eventA:
    // ...
  break;
  case eventB:
    // ...
  break;
}
```

4.2.3. Channels

Channels are a way of communication between tasks on different cores of the processor. For communication between tiles it is essential to use channels since each tile has its own memory which is not shared with other tiles. There are two types of channels: normal channels which are synchronous and streaming channels which are asynchronous. The difference is that synchronous channels are blocking and wait until the value is received whereas asynchronous channels are non blocking and have a First-In-First-Out (FIFO) buffer to temporarily store values.
Channels and streaming channels are defined by special keywords in the XC-language:

```
chan c1;
streaming chan c2;
```

In order to connect two tasks with a channel the channel has to be created first and then passed as a function argument to the two tasks which are started inside a `par` statement. To pass channels (normal and streaming) as function arguments the keyword `chanend` is used.

```
void task1(chanend c);
void task2(chanend c);
chan c;

par {
  task1(c);
  task2(c);
}
```

To send and receive with channels there are two new operators: `:>` and `<:`. The following code snippet demonstrates their use.

```
void task1(chanend c) {
  c <: 4;     // sending the integer 4
}
void task2(chanend c) {
  int value;
  c :> value; // receiving a value
}
```

Channels can also work in a `select` where they trigger an event once data is ready to be received.

```
select {
  case c :> int x:
    // x now holds the received data ...
  break;
}
```

4.2.4. Clock Blocks

Clock blocks are hardware programmable clocks that are used for the hardware ports (see next Chapter). There are 6 clock blocks per tile. The first one runs at a fixed rate of $100MHz$

and can not be programmed. All ports are initially connected to that clock block. For the rest there are two options: externally driven (clock input pin) or an integer divider of the system clock. A clock can also drive an output pin directly. Non externally driven clocks also need to be started before they can be used. The following code snippet illustrates the use of clocks.

```
#include <xs1.h>

clock clk1 = XS1_CLKBLK_1;          // clock block 1
clock clk2 = XS1_CLKBLK_2;          // clock block 2
in port p_clk_in = XS1_PORT_1A;     // clock input pin
out port p_clk_out = XS1_PORT_1B;   // clock output pin

configure_clock_rate(clk1, 100, 8);           // sets clk1 to 100/8MHz = 12.5MHz
configure_port_clock_output(p_clk_out, clk1); // outputs clock clk1 on a pin
start_clock(clk1);                            // starts the clock

configure_clock_src(clk2, p_clk_in);          // sets clk2 to input from a pin
```

4.2.5. Ports

The processor has multiple physical IO-pins which are connected to either tile 0 or tile 1 and are named $X0D00$ where the first number stands for the tile number and the second one for the pin number. Each tile has 44 IO-pins named $X0D00$ to $X0D43$ for tile 0 and $X1D00$ to $X1D43$ for tile 1.
The IO-pins are grouped into ports with different width which can consist of 1, 4, 8 or 16 pins (see Table 4.1). The name of each port starts with a number representing the number of pins in the port and a letter which distinguishes different ports of same width from each other. To represent an individual pin of a port a subscript to the port name is used with the number of the pin inside the port (eg. $8B^0$ - $8B^7$). Table 4.1 shows the beginning of the pins and ports table. Not all of these ports can be used simultaneously as parts of ports with different width overlap each other rendering all but one port for each pin unusable. Each of the ports can be selected to be an input or output port. Mixing input and output pins inside one port is not supported.
To create a port variable in the code the keyword *port* is used and in the file *xs1.h* the port names are defined.

```
#include <xs1.h>

in port p = XS1_PORT_4A;  // port 4A
out port p = XS1_PORT_4B; // port 8B
```

All ports are clocked which means a shift register is in front of each port and the port is only sampled or written on each clock tick. To connect a clock block to a port the functions

```
void configure_out_port(port p, clock clk, int initial_value)
```

and

```
void configure_in_port(port p, clock clk)
```

are used.

Ports are read and written just like streams with the operators :> and <:. All pins are read or set as one number where each bit of the number in binary form represents one of the pins. Ports can also be sampled inside a select. The select is continuously triggered when the specified condition applies. To trigger an event only when the port changes its value the following code snippet can be used which changes the event condition based on the last port value.

```
in port p = ...;
unsigned i = 0;
p :> x; // initialize x with the value of the port - prevents initial trigger
while (1) {
  select {
    case p when pinsneq(x) :> x : // event when pins not equal x
      // x now contains the new port value
      // ...
      break ;
  }
}
```

Signal	Port			
$X0D00$	$1A^0$			
$X0D01$	$1B^0$			
$X0D02$		$4A^0$	$8A^0$	$16A^0$
$X0D03$		$4A^1$	$8A^1$	$16A^1$
$X0D04$		$4B^0$	$8A^2$	$16A^2$
$X0D05$		$4B^1$	$8A^3$	$16A^3$
$X0D06$		$4B^2$	$8A^4$	$16A^4$
$X0D07$		$4B^3$	$8A^5$	$16A^5$
$X0D08$		$4A^2$	$8A^6$	$16A^6$
$X0D09$		$4A^3$	$8A^7$	$16A^7$
$X0D10$	$1C^0$			
$X0D11$	$1D^0$			
$X0D12$	$1E^0$			
$X0D13$	$1F^0$			
$X0D14$		$4C^0$	$8B^0$	$16A^8$
$X0D15$		$4C^1$	$8B^1$	$16A^9$
$X0D16$		$4D^0$	$8B^2$	$16A^{10}$
⋮		⋮		

Table 4.1.: Beginning of the ports definition table of the processor. [9]

4.3. XTime Composer - Integrated Development Environment

To develop firmware for the xCORE processors XMOS provides their own customized Integrated Development Environment (IDE) build upon the open source IDE Eclipse [13]. The main benefit of this IDE is that it seamlessly works with the xTAG programmers and the xCore processors. But it also has some very useful features for developing firmware. One feature is a timing analyzer which analyzes the compiled binary and shows best and worst case signal runtimes. Another feature is the trace view which lets you inspect xSCOPE

traces. The xSCOPE is used to read signal values via the programmer in real time and plot them over time. The xSIM simulator is also a useful feature that can be used to simulate the processor. The simulator can also create value change dumps (VCD) that contain data about the state of variables or ports and can be viewed with the trace feature as a plot. For a full list of features and information on how to use the IDE there is the "xTIMEcomposer User Guide" [14].

4.4. xTAG 3 - Programmer

XMOS developed their own programmer called xTAG of which version 3 was used during development of this project. The xTAG programmer is an extended Joint Test Action Group (JTAG) programmer. In addition to the standard JTAG signals it has additional custom signals which replace some ground (GND) pins on the standard 10 pin JTAG connector and therefore ensures compatibility. These pins are used for the XMOS xSCOPE in the xTime Composer to visualize variables and signals of the processor in realtime. Figure 4.2 shows the pinout of the connector on the sound card with the default JTAG pins and the extra xSCOPE pins labeled *XL*1-*UP*0, *XL*1-*UP*1, *XL*1-*DN*0 and *XL*1-*DN*1. It also shows the xTAG 3 programmer which can be directly plugged into the connector on the sound card without the need of any wires.

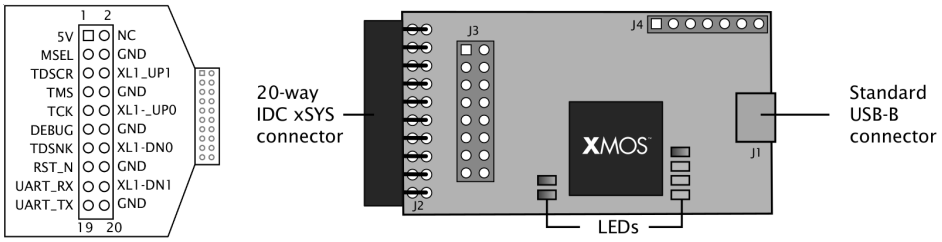


Figure 4.2.: Programmer xTAG 3 and pinout of the connector on the board [15].

5. Hardware

The hardware of this project was based on the *xCORE Array Microphone Board* [1]. It was adopted to the needs for this project. Figure 5.1 shows the block diagram of the hardware. The full schematic can be found in Appendix B.

The processor is the center part oft the hardware design. Its capabilities and function is explained in chapters 4. Processor and 6. Firmware.

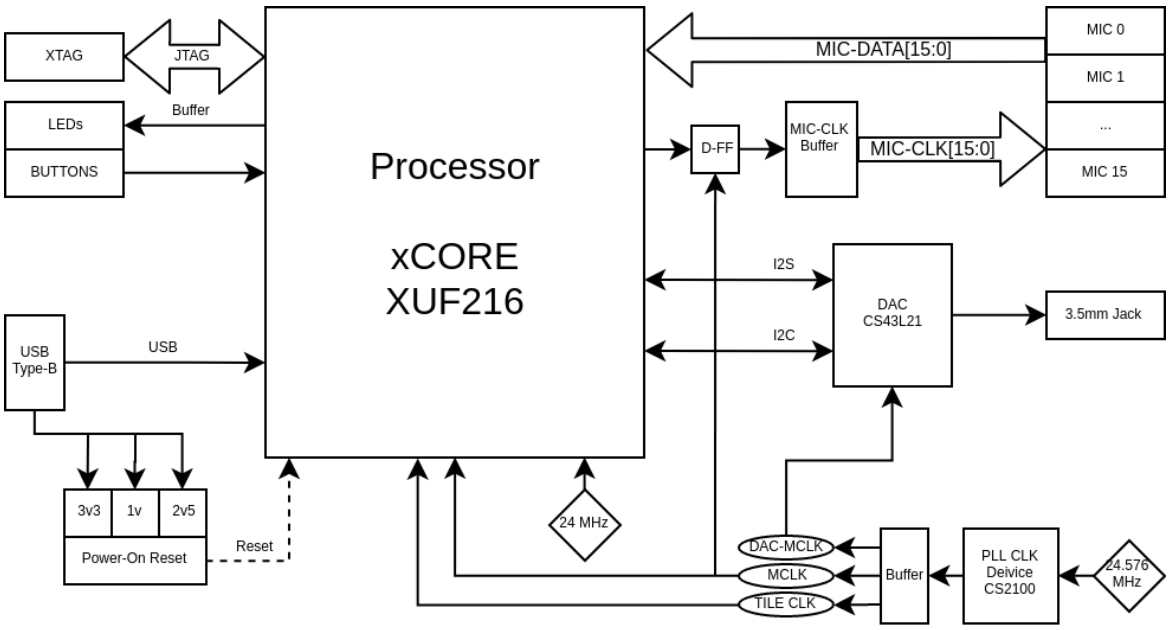


Figure 5.1.: Hardware block diagram.

In the following the different building blocks are introduced in more detail.

5.1. Voltage Regulation

The circuit requires 3 different voltages to operate: 3.3V, 1V and 2.5V. The 3.3V are used by most of the ICs as the primary power source. The processor uses the 1V as its primary power supply and the 3.3V are only used for the IO-peripheral to communicate with other ICs that also use 3.3V [9]. The DAC is the only IC which uses the 2.5V which is used for both analog and digital supply. Like the processor the DAC uses the 3.3V only for interfacing with other ICs [16].

5.1.1. 3.3V & 1V

For the 3.3V and 1V two ST1S06 [17] adjustable, step-down switching regulators are used. Since this IC uses a 1.5 MHz pulse with modulated signal (PWM) to generate the desired voltage an output low-pass filter is needed. This filter is implemented as a first order LC-Filter with an inductivity of $2.2\mu H$ and a capacity of $2.2\mu F$. The cutoff frequency is $72kHz$ which is low enough to filter the faster PWM signal. A schematic is shown in Figure 5.2.

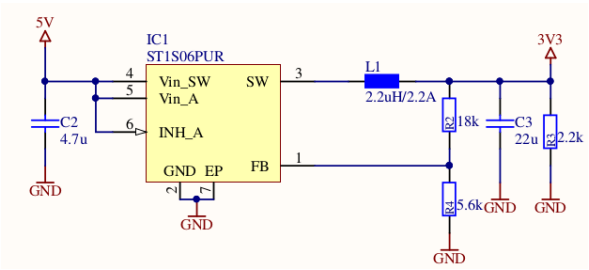


Figure 5.2.: 3.3V voltage regulation schematic.

5.1.2. 2.5V

The 2.5V is needed by the DAC CS43L21 [16]. Therefore a low-dropout regulator (LDO) TLV70025 [18] which is a fixed-voltage regulator is used. This means that apart from two capacitors for voltage stabilization there are no other components needed by this IC (see Figure 5.3). As input voltage for this LDO the 3.3V are used. Since the dropout voltage is rated at a maximum of 250mV the difference of 800mV is more than sufficient for this voltage regulator to operate correctly. The TLV70025 also has a very high power-supply rejection ratio (PSRR) of 68 dB at 1 kHz which makes it perfect for audio applications where noise is not desirable.

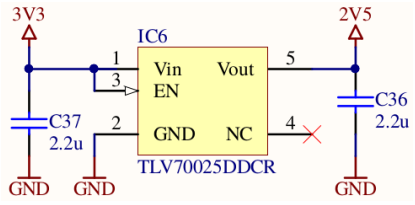


Figure 5.3.: 2.5V voltage regulation schematic.

5.2. Power Supply Sequencing & Power-On Reset

The term Power Supply Sequencing refers to the process of starting all the power supply ICs in a predefined sequence. This is important because some ICs or some parts of a circuit need other ICs to be fully operational before they are powered in order for the circuit to work correctly.

Power-On Reset (POR) is a concept for reseting circuits or ICs during the power-on phase - when the power supply is turning on and the voltage is rising from zero to the desired supply voltage. Due to the fact that this transition cannot be done instantly there is a short

time when the voltage is high enough for some ICs to start operating but lower than the minimum operating voltage where proper operation is ensured. In this time the IC that is powered on is not guaranteed to work correctly and could therefore enter an undefined or undesired state. In case of the microprocessor this means that the program could start running but other internal circuits are not yet working correctly.

The communication with other peripherals that are also not yet powered up correctly is also not guaranteed to work in the desired manner. For all ICs to start operating at a point in time where the input voltage is stable a POR circuit is needed. It will keep the reset input of the processor low which will keep the processor in reset mode where all execution is halted until the reset input is switched to high again and the execution of the firmware starts.

The first component in the power sequencing chain is a STM1061N28WX (see Figure 5.4) Low Power Voltage Detector[19] which monitors the output of the 3.3V voltage regulator which is the first one to power up. The voltage detector is a fairly simple component that has only two input pins for the power and 0V line and one output that is pulled low until the input voltage reaches a certain level and is then pulled high (to the current input voltage level). The output of this voltage detector is called *Power Good* $3.3V$.

The second voltage regulator with an output voltage of 1V is being enabled by the *Power Good* $3.3V$ signal and only starts when the 3.3V are stable.

Next in line is the ADM1085 sequencing circuit [20] (see Figure 5.4) which is supplied with 3.3V and therefore also enabled with the power good 3.3V signal. After being powered on the ADM1085 measures the 1V supply voltage and waits for it to rise above a fixed threshold. Since this threshold is specified at a minimum of $0.56V$ to a maximum of $0.64V$ (due to tolerances) a voltage divider is used that reduces the input voltage by a factor of about 0.72. Including a 5% tolerance for the resistors the factor is at a minimum of 0.65 which is still above the specified maximum threshold of the ADM1085. Once the threshold is reached a time delay is triggered which has a minimum delay of $35\mu s$ realized by an internal capacitor. This time delay can be extended by adding an external capacitor. In this case a 2.2nF capacitor is used. The resulting time delay can be calculated with Equation 5.1 resulting in approximately $11ms$. After this time delay the enable output which is used as the POR signal in this circuit is switched from low to high.

$$t_{EN} = \left(C \cdot 4.8 \cdot 10^{6}\right) + 35\mu s = \left(2.2 \cdot 10^{-9} \cdot 4.8 \cdot 10^{6}\right) + 35\mu s \approx 11ms \tag{5.1}$$

The POR signal from the ADM1085 is then used to keep the processor in the reset state until the power-on delay is over. Due to the fact that the JTAG interface used for programming also needs to control the reset state of the processor both the POR and the reset signal of the JTAG interface need to be connected to the reset pin of the processor. Therefore an NC7WZ07P6X Dual Buffer [21] (see Figure 5.4) with an open-drain output is used to act as an AND-gate by connecting both outputs of the buffer.

The POR signal is also connected to an LED near the USB connector to indicate that the device is powered.

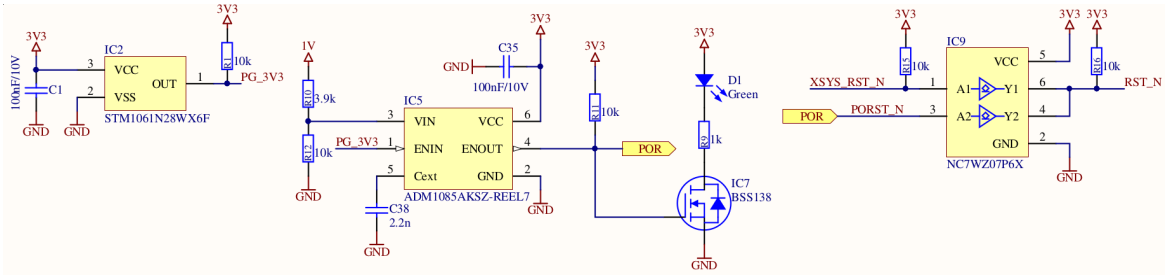


Figure 5.4.: Power good and power-on reset schematic.

5.3. System Clock

The main clock source for the processor is an ASFL1 [22] crystal clock oscillator with $24MHz$. This clock is connected to the processor's clock input pin which is in turn connected to the processor's internal PLL that creates the higher clock speed at which the processor runs. The value of $24MHz$ is one of two possible values if the use of USB is desired (the other one being $12MHz$).

5.4. Audio Clock

To provide a low jitter clock for the microphones and the DAC the low jitter fractional-N phase locked loop (PLL) CS2100-CP [23] (see Figure 5.5) was added. This PLL is freely configurable via an I^2C interface where the output frequency can be set. To generate the output clock a synchronization clock is needed which is generated by the processor and runs at $1MHz$. A secondary input clock is needed by the PLL as a low jitter reference clock which is provided by another ASFL1 [22] crystal clock oscillator with $24.576MHz$. The output of the PLL is buffered with a NC7NZ34K8X TinyLogic UHS Triple Buffer [24].

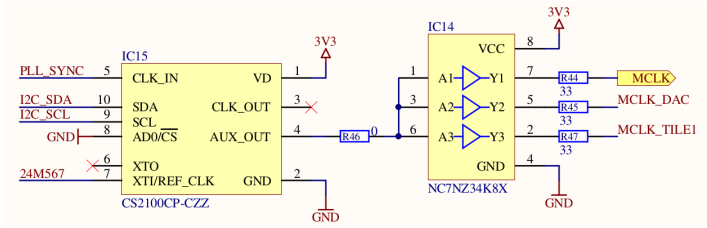


Figure 5.5.: Schematic: PLL with Buffer.

The generated clock signal is configured based on the audio sampling frequency and is either $48kHz$ or $44.1kHz$. This signal is then fed into the processor and into the audio DAC. It is also used for jitter reduction of the microphone clock which is generated by the processor. This is done with a D-Flip-Flop as shown in Figure 5.6. The clock signal generated by the processor is routed to the D-input of the Flip-Flop and the faster clock of the PLL serves as the clock-input signal for the Flip-Flop. This ensures that the output only changes with the PLL clock which leads to a jitter reduction.

After the Flip-Flop the microphone clock is fed into two SN74LVC125ARGYR [25] buffer ICs (see Appendix B). Since there is a total of eight individual buffers each buffer drives two

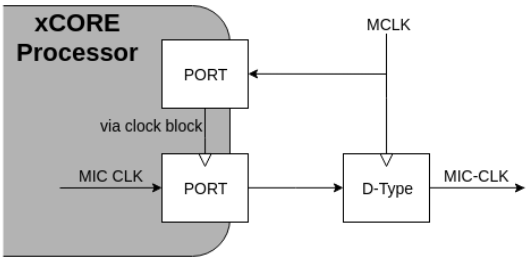


Figure 5.6.: Microphone clock jitter reduction block diagram.

microphones.

5.5. USB

The USB part of the schematic is fairly simple as it only consists of the USB connector and a diode array for protection of the processor. There is also one resistor with 43.2Ω pulling one pin of the processor to ground as the datasheet indicates this requirement.

5.6. Audio output DAC

For the stereo audio output a CS43L21 Low-Power, Stereo Digital-to-Analog Converter [16] is used (see Figure 5.7). There are two interfaces used. The first is an I^2C interface for configuration which has a shared bus with the audio clock PLL. The second is the serial audio input which is connected to the I^2S interface of the processor.
There is a charge pump included in the DAC which provides a negative voltage to make ground the center voltage of the output. This removes the need for DC blocking capacitors in series to the audio output signals.
Since the DAC needs to be configured in under $10ms$ after reset to avoid entering a standalone mode the reset pin of the DAC is connected to an output pin of the processor to ensure the correct timing via the firmware.

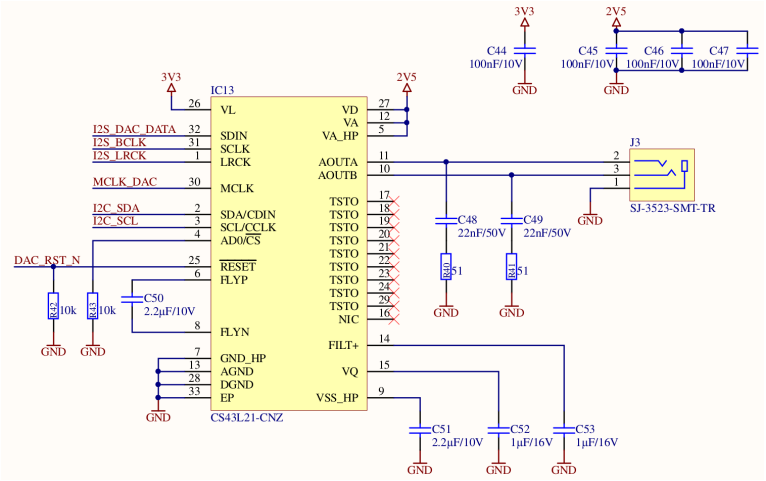


Figure 5.7.: Schematic: DAC.

6. Firmware

The firmware is based on the application *app_usb_aud_mic_array* inside the *XMOS USB Audio 2.0 Software* package [26] which was designed for the *xCORE Array Microphone Board* [1] with 7 microphones. To support 16 microphones the base application as well as the library *lib_mic_array* [27] had to be modified. Figure 6.1 illustrates the general structure of the firmware and the communication with the peripherals.

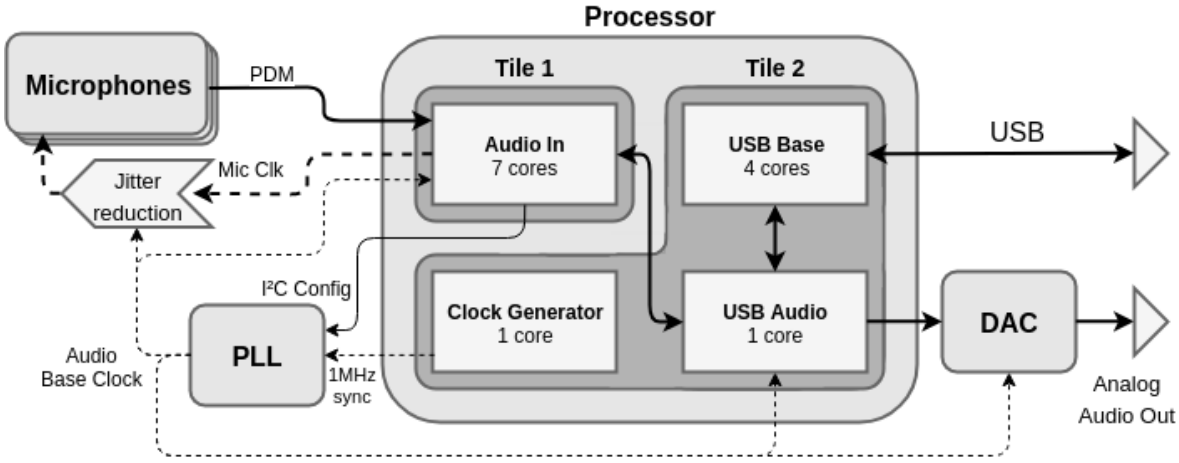


Figure 6.1.: Block diagram of the firmware.

In the following the individual blocks are discussed in more detail.

6.1. Audio In (Microphones)

Due to the computational power needed for the audio input processing chain, it was split into several tasks running on multiple cores of the processor. There are three distinct tasks. The first one (pdm_rx) can handle 8 audio signals per core, the second one (decimate_to_pcm_4ch) can handle 4 audio signals per core and the third one (pdm_process) needs only one core for all 16 audio signals. This totals to 7 used cores. Figure 6.2 shows the audio in processing chain with the tasks in three columns and each task instance (each core) as a dark grey rectangle.

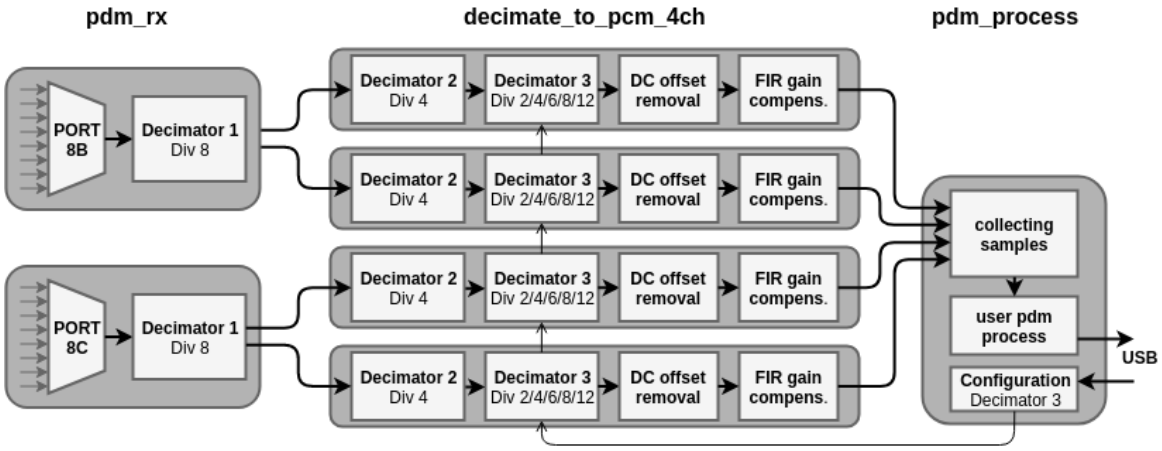


Figure 6.2.: General structure of the PDM interface.

6.1.1. Task: pdm_rx

```
void mic_array_pdm_rx(in buffered port, streaming chanend, streaming chanend)
```

The task *pdm_rx* is the first step of the audio processing chain. The first argument is a physical input port (see Chapter 4.2.5) where the data pins of 8 MEMS microphones are connected. The second and third arguments are two streaming channels (see Chapter 4.2.3) where the processed data is split into two parts with 4 audio signals each and sent to the next tasks of the processing chain. In order to process 16 audio signals two instances of this task need to be running.

The *pdm_rx* task consists of two parts. The first one is reading the 8 bits of the input port. The second part is the first stage of the decimation process: an optimized high speed FIR filter with 48 tabs which reduces the input sample rate by a factor of 8. Since this filter operates at a high data rate (with a maximum input sample rate of 3.072MHz and output sample rate of 384kHz) a generic FIR filter implementation would require too many cores to process the 600MIPS or more ($16 \cdot 48 \cdot 384kHz = 294.912.000$ multiply-accumulate-operations plus overhead for accessing data and coefficients).

First Stage Decimation Filter

Since the input PDM signal from the microphones is binary, the first factor of each multiplication can only asume the values one or zero. To exploit this fact the first step is splitting up the 48 coefficient filter into 6 groups of 8 sample values, i.e.,

$$y[n] = \sum_{i=0}^{48-1} x[n-i] \cdot b_i = \sum_{i=0}^{6-1} \left(\sum_{j=0}^{8-1} x[n-(8i+j)] \cdot b_{8i+j} \right). \tag{6.1}$$

The result is a sum of six terms with 8 multiplications. Every multiplication has one binary input sample x as the first factor and a constant filter coefficient b_{8i+j} as the second factor. This means that every multiplication has only two possible results and each sum of 8

multiplications has only 256 possible outcomes. Therefore each sum of 8 multiplications can be converted into a table lookup. For the whole filter this results in the sum of 6 table lookups in 6 different lookup tables. This saves 8 multiply-accumulate-operations along with loading 8 coefficients into registers.

The index of the lookup tables is one byte wide as there are 256 different values. The bits of this byte are the 8 binary input samples in one of the 6 groups. This is determined as

$$index_{lut}(n,i) = \sum_{j=0}^{8-1} 2^j \cdot x\left[n - (8i + j)\right]. \tag{6.2}$$

The result of the FIR filter can now be reduced to the sum of 6 values from 6 lookup tables (lut_i) as

$$y[n] = \sum_{i=0}^{6-1} lut_i\left[index_{lut}(n,i)\right] \tag{6.3}$$

The drawback of this method is that it consumes more memory for the precalculated coefficient lookup table. Since it needs 256 values per block of 8 samples it needs 32 times the memory of a normal FIR filter.

As this is a linear phase filter the coefficients must be symmetric. Three of the six lookup tables are, therefore, mirrored versions of the first three. A lookup with index i in lookup table 6 yields the same result as a lookup with the bit reversed index $bitrev(i)$ in lookup table 1. Since bitreversing is an efficient instruction on the used processor architecture lookup table 3-6 can be omitted, saving 3072 bytes of memory. The result can be determined as

$$y[n] = \sum_{i=0}^{2} lut_i\left[index_{lut}(n,i)\right] + \sum_{i=3}^{5} lut_{5-i}\left[bitrev\left(index_{lut}(n,i)\right)\right], \tag{6.4}$$

and this computation is visualized in Figure 6.3.

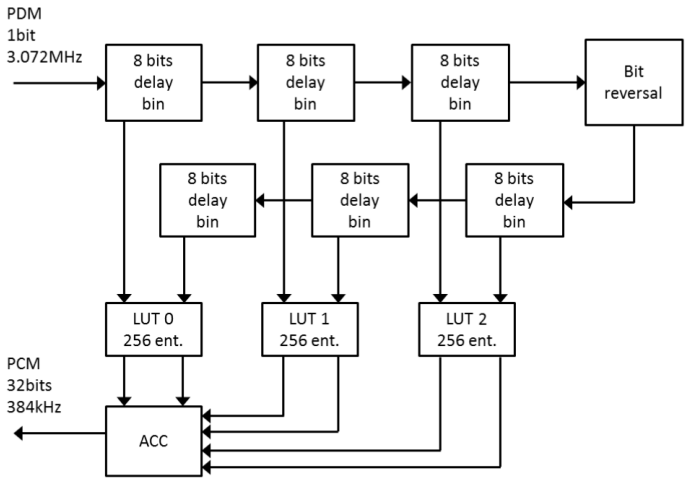


Figure 6.3.: Strtucture of the first decimation filter [28].

6.1.2. Task: decimate_to_pcm_4ch

```
void mic_array_decimate_to_pcm_4ch(streaming chanend, streaming chanend)
```

As arguments this task receives one streaming channel connected to one of the previous task instances (*pdm_rx*) from which it receives samples from 4 audio channels. The second argument is also a streaming channel connected to the last task of the signal processing chain, i.e., *pdm_process*. On that channel this task sends the processed samples to the last task and receives the filter configuration for the third stage FIR filter. Four instances of this task are needed to process 16 audio signals.

There are four stages as seen in Figure 6.2. First is an optimized linear phase FIR filter and decimator which decimates the input by a fixed factor of 4, reducing the sample rate to 96 kHz (assuming a PDM clock of 3.072MHz) or 88.2 kHz (for a pdm clock of 2.8224MHz). Next there is a DC offset removal followed by a FIR gain compensation.

Second Stage Decimation Filter

This FIR filter decimates by a fixed factor of 4 and has 16 coefficients. Since this filter also runs fairly fast it is implemented in assembler and is optimized to reduce processing time.

The first optimization is theoretical as, the filter is linear phase and all linear phase filters have symmetric coefficients. So for each coefficient loaded from memory, two multiply accumulate actions can be done on two different (symmetric) samples. Therefore a new coefficient is only loaded every second data sample. This halves loading time of coefficients.

The second optimization is architecture specific. It uses the load double assembler command which can load two data samples or two coefficients in one command. This results in the following linear phase FIR implementation block that handles 4 data samples and repeats 4 times where n is the number of the repetition and N the total number of data samples.

- Load 2 coefficients $(2n, 2n+1)$
- Load 2 data samples $(2n, 2n+1)$
- MAC coefficient 1 & data sample 1
- MAC coefficient 2 & data sample 2
- Load 2 symmetric data samples $(N-(2n+1), N-2n)$
- MAC coefficient 1 & data sample 3
- MAC coefficient 2 & data sample 4

This results in 7 operations for 4 coefficients which results in an overall 1.75 operations per coefficients. Also, there is no stall in the execution pipeline since loading and multiply-accumulate operations are interleaved.

Circular Buffer Simulation

Most FIR implementations rely on circular buffers for the data samples but since the processor architecture does not have a circular buffer in hardware the circular buffer management needs to be implemented in software.

One way to implement a circular buffer is to have an array of the length of the circular buffer and when writing or reading from it the index used is incremented in conjunction with the modulo of the length of the circular buffer. To avoid any extra calculations while reading each sample from the buffer the used circular buffer is implemented in a way that all the buffer management is done while writing to the buffer. To achieve this the buffer is made twice the size needed to hold all samples and each sample is written twice - once at its current position and again one buffer-length away. This can be seen in Figure 6.4. With this trick a convolution of the samples can always be performed as a whole operation without having to check the buffer boundaries for every sample thus saving on instructions per tap for the price of only double the memory for the circular buffer.

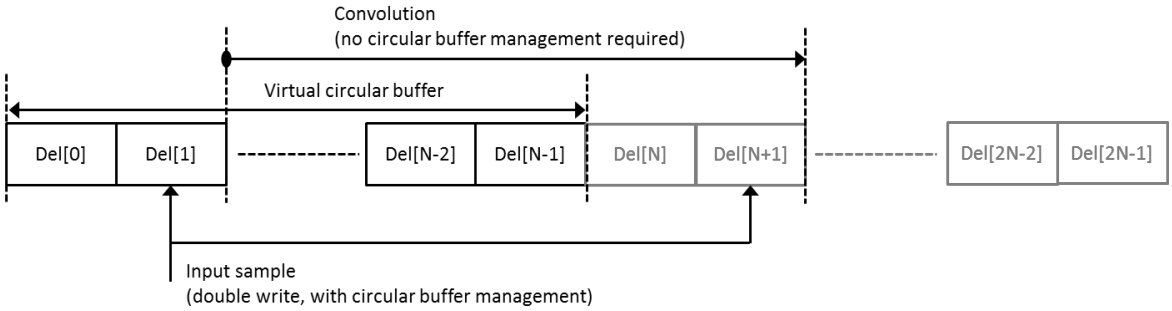


Figure 6.4.: Implementation of the circular buffer [28].

Third Stage Decimation Filter

The third and final FIR decimation filter is configurable during runtime and decimates the sample rate of the previous decimation filter by another factor 2, 4, 6, 8 or 12 according to the selected output sample rate of 48kHz, 24kHz, 16kHz, 12kHz or 8kHz for a PDM clock of 3.072MHz or a sample rate of 44.1kHz, 22.05kHz, 14.7kHz, 11.025kHz or 7.35kHz for a PDM clock of 2.8224MHz. The configuration is done by receiving the decimation factor and the filter coefficients over the channel connected to the next task (pdm_process).

DC Offset Removal

The DC offset removal is done by an IIR filter with a single pole as

$$Y[n] = Y[n-1] \cdot \alpha + x[n] - x[n-1]. \tag{6.5}$$

The coefficient α adjusts the stability and the settling time of the filter in opposite directions.

FIR Gain Compensation

The FIR gain compensation is a simple multiplication with a fixed point number. This factor is calculated for all three decimation filters and changes with the different decimation values of the third decimation filter as the filter changes its characteristics for different sampling frequencies.

6.1.3. Task: pdm_process

```
void pdm_process(streaming chanend [4], chanend)
```

This task is connected to the USB Audio task via a bidirectional channel which is the second argument. It receives the sample rate when the USB port is connected to a host and continuously checks for sample rate changes. Upon receiving such a sample rate change event over the channel it reconfigures the FIR decimation filter to the desired decimation factor. If switching between sample rates with different base frequencies (48kHz or 44.1kHz) the PLL is also reconfigured via I^2C. This task is also connected via 4 unidirectional channels (first argument) to the 4 previous tasks receiving a continuous stream of audio samples which are then sent to the USB task after being routed through a user-defined function. This function is designed to contain user-defined signal processing and can be used to extend the functionality of this Sound Card (see Chapter 6.3.2). At the current state it only reorders the channels to match the order of the connectors on the PCB. Figure 6.5 shows a flow graph of this task's functionality.

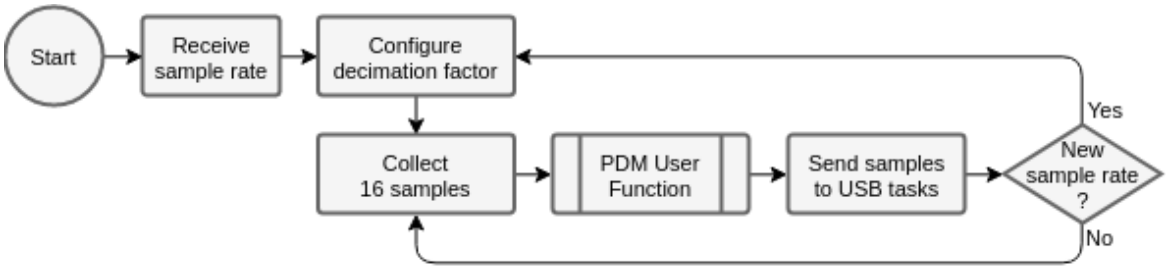


Figure 6.5.: Flow graph of the task pdm_process.

6.2. USB & Audio IO

This chapter contains a brief description of the XMOS USB library including the USB to I^2S bridge used for the DAC.

6.2.1. Tasks: XUD Manager & Endpoint 0

The XMOS USB Device (XUD) Manager task takes care of all the low level USB communication. It is the base for writing USB device firmware for XMOS processors. The Endpoint 0 task is also a basic building block for USB firmware as it handles all control transfers over the special USB Endpoint 0 including initialization of the USB transfer and configuration. It handles all requests including audio requests and informing the necessary task (e.g. sending a new sample rate to the audio task).

6.2.2. Tasks: Endpoint Buffer & Decoupling

These two tasks are responsible for buffering data between the *Audio IO* task and the *XUD Manager* task. The *Endpoint Buffer* task is connected via a channel to the *XUD Manager* task and the *Decoupling* task is connected to the *Audio IO* task via another channel. Since the tasks *Audio IO* and *XUD Manager* of this firmware run on different time basis a way to synchronize them is needed. Therefore the *Endpoint Buffer* task and the *Decoupling* task communicate via a shared part of memory where a bidirectional first-in-first-out (FIFO) buffer is implemented. The sample rate of the USB connection is then controlled so the buffer never runs out of audio samples nor becomes full.

6.2.3. Task: Audio IO

The *Audio IO* task sends and receives audio streams to and from the *Decoupling* task. The stream it receives is then sent over the I^2S interface to the audio DAC (see Chapter 5.6). The stream it sends comes from the *pdm_process* task containing the 16 audio signals from the microphones. It is also responsible for taking actions when a sample rate change occurs. The first action taken is informing the *pdm_process* task via the shared channel. Then it determines if the change in sample rate changes the base audio clock of the PLL (see Chapter 5.4). If so it then reconfigures the PLL over the I^2C interface. The same interface is also used to reconfigure the audio DAC for the new clock rate.

6.3. Extending the Firmware

6.3.1. Changing The Existing Filters

As explained in Chapter 6, Chapter 6.1.1 and Chapter 6.1.2 there are three filters responsible for low pass filtering the microphone signals before each of the three decimation steps. The chosen filters are suited to normal audio recording but can be changed to any kind of filter (that includes a low pass to prevent aliasing) if the application requires it. This chapter explains where to edit the coefficients and introduces the Python tool *fir_design.py* to calculate these coefficients for each of the three filters. The characteristics of the default filters can be found in Appendix A.
The script *fir_design.py* can be found in */lib_mic_array/src/fir/*. The coefficients of all of the filters are stored in the files *fir_coefs.xc* and *fir_coefs.h* in the same directory. This file is created automatically by the python script. This means that any manual changes to the filter coefficients are lost when running the python script.

The python script takes different arguments to adjust the filters. The visualization of these arguments can be seen in Figure 6.6, 6.7 and 6.8.

- `--pdm-sample-rate` this is the sample rate all other frequencies are normed to. The filter will have a slightly different behaviour in 3.072MHz mode than in 2.8224MHz mode. This value defaults to 3.072MHz.
- `--stopband-attenuation` this is set for all of the filters (1-3). Defaults to 80dB.
- `--first-stage-pass-bw`, `--second-stage-pass-bw` are the pass band bandwidth in kHz for the first two filters. Both default to 16kHz.

- `--first-stage-stop-bw`, `--second-stage-stop-bw` this is the stop bandwidth in kHz for the two filters. Both default to 100kHz.

The third stage filter is best changed by editing the python script as one can only add new decimation filters as opposed to changing existing filters. The filters are defined in a variable named `third_stage_configs` at the end of the script. The variable holds an array with an entry for each decimation factor. Each entry is also an array containing the following values: *decimation factor* (do not change!), *normalized passband cutoff frequency*, *normalized stopband cutoff frequency*, *filter name* (do not change!), *number of tabs* (do not change!).

If a different type of filter is desired which is not a low pass with the constraints described above then there are two possibilities. The first is to edit the coefficients directly. To do this it is advised to modify the python script to export the coefficients to the files *fir_coefs.xc* and *fir_coefs.h*. In the script the functions `generate_first_stage`, `generate_second_stage` and `generate_third_stage` all contain a variable `coefs` which can be assigned the desired coefficients instead of them being calculated by the script. For the third stage there need to be different coefficients for each decimation factor.

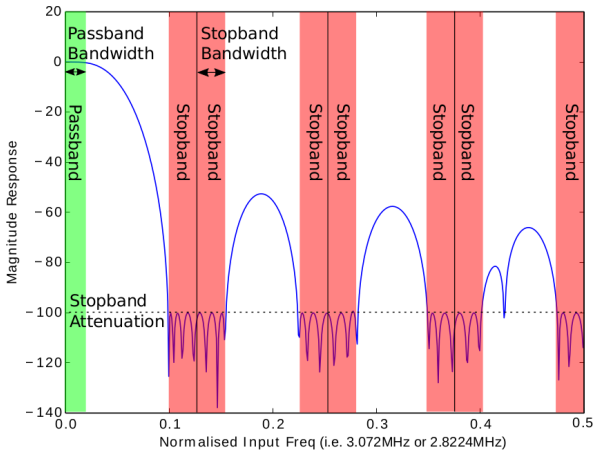


Figure 6.6.: First stage decimation filter design parameters [27].

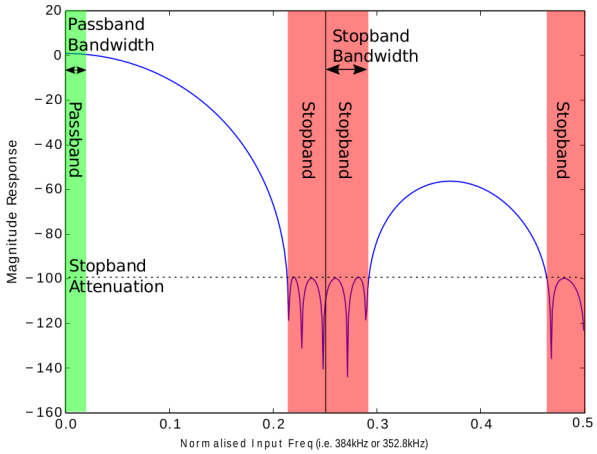


Figure 6.7.: Second stage decimation filter design parameters [27].

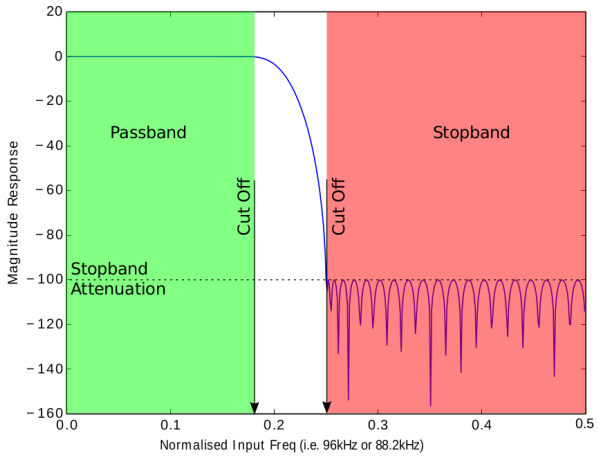


Figure 6.8.: Third stage decimation filter design parameters [27].

6.3.2. Adding Custom Signal Processing

To introduce custom signal processing after the third decimation filter into the processing chain the function *user_pdm_process* which is called from the function *pdm_process* (Chapter 6.1.3) and found in the file */app_usb_aud_mic_array/src/extensions/pdm_user.xc* can be be modified. At the moment this function only reorders the audio channels to match the numbers of the connectors.

This function is called after the third FIR decimation filter is applied and before the samples are sent over USB. It is called with the following parameters:

```
void user_pdm_process(mic_array_frame_time_domain* unsafe audio, int output[])
```

The function is called once per sampling time point. The samples for the 16 microphones which are 32 bit integer values can be accessed in the following way:

```
audio->data[micNr][0]
```

The samples which are sent back over USB are also 32 bit integers and are set like this:

```
output[micNr] = ...
```

Only the upper 24 bit of these 32 bits are sent over USB.

7. Prototype

This Chapter shows the finished sound card and contains some basic functional tests.

7.1. Developed Sound Card

From the schematics found in Appendix B a PCB layout was created. Two PCBs were ordered along with all electronic parts needed. The two PCBs were then soldered by hand. Microphones were then needed to test the sound card. Therefore a second PCB layout containing a MEMS microphone and a connector was created which is not part of this thesis. Thirty-two of these PCBs were ordered and soldered by hand. For the connection between the MEMS microphones and the sound card thirty-two ribbon cables were also assembled. To hold the sixteen individual MEMS microphones of one sound card physically together a plate was 3D-printed. This plate contains holes for the sound ports of the microphones as well as holes for screws. The finished prototype can be seen in Figures 7.1, 7.2 and 7.3.

Figure 7.1.: Prototype of the sound card.

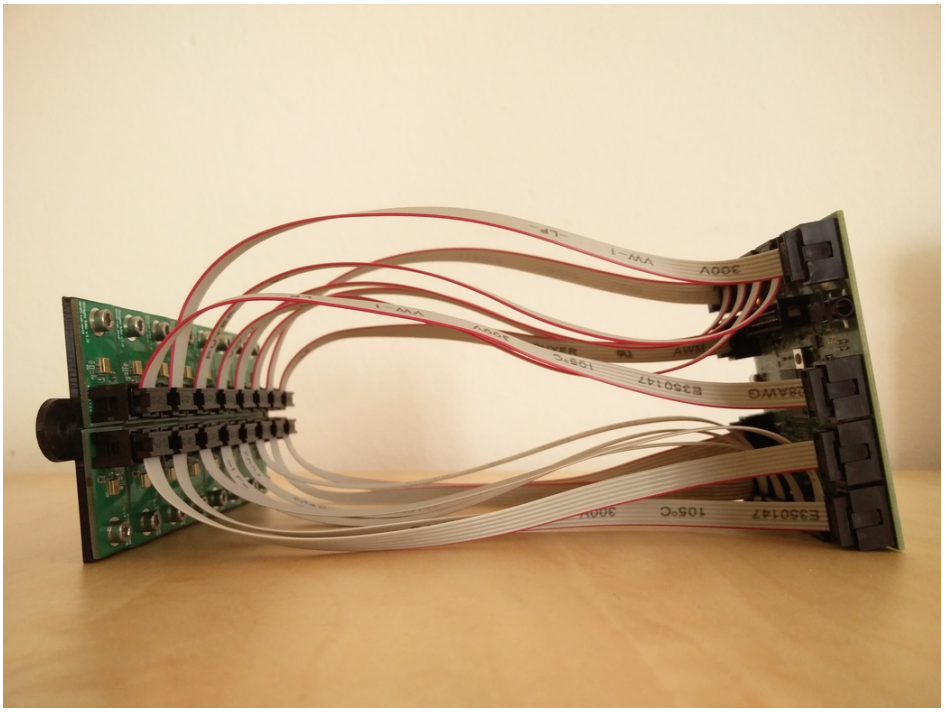


Figure 7.2.: Assembled sound card prototype with microphones.

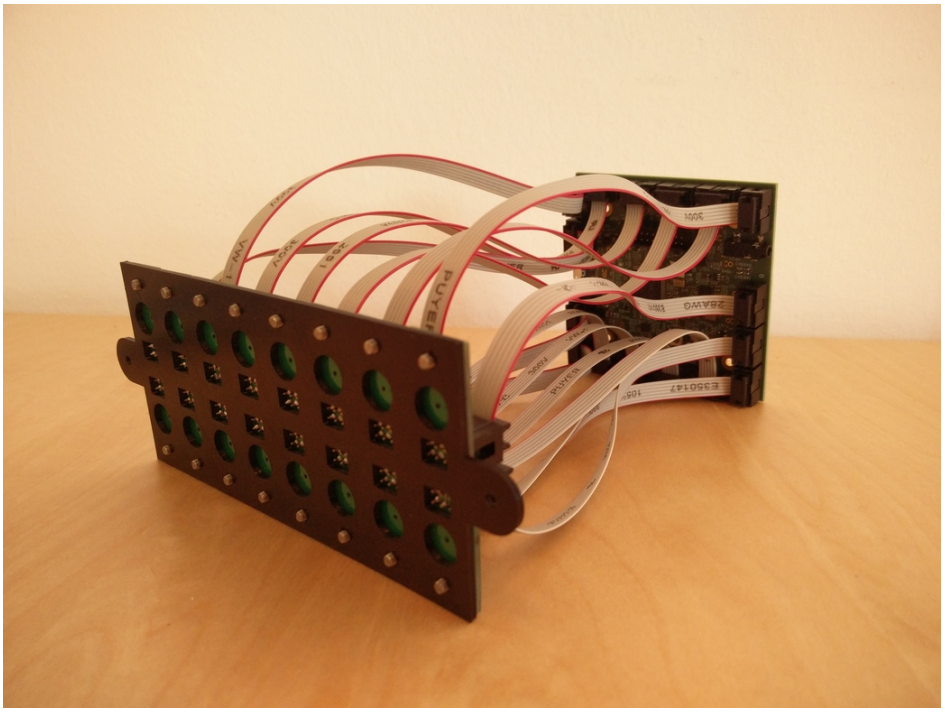


Figure 7.3.: Assembled sound card prototype with microphones.

7.2. Functional Test

The sound card and the microphone PCBs were soldered by hand. For the test setup a plastic plate was 3D-printed to hold the 16 microphone PCBs together. The sound card was then connected via sixteen ribbon cables with a length of 20cm. For the individual channel test an in-ear headphone was playing a 2kHz tone and was held separately near each of the 16 microphones. All of the tested microphones worked as expected and the results can be seen in Figure 7.4. Since the microphones were not physically separated the tone can be seen in the other channels too with a very low amplitude. The variation in amplitude displays the constraints of working by hand which resulted in the distance and angle of the speakers and microphones varying.

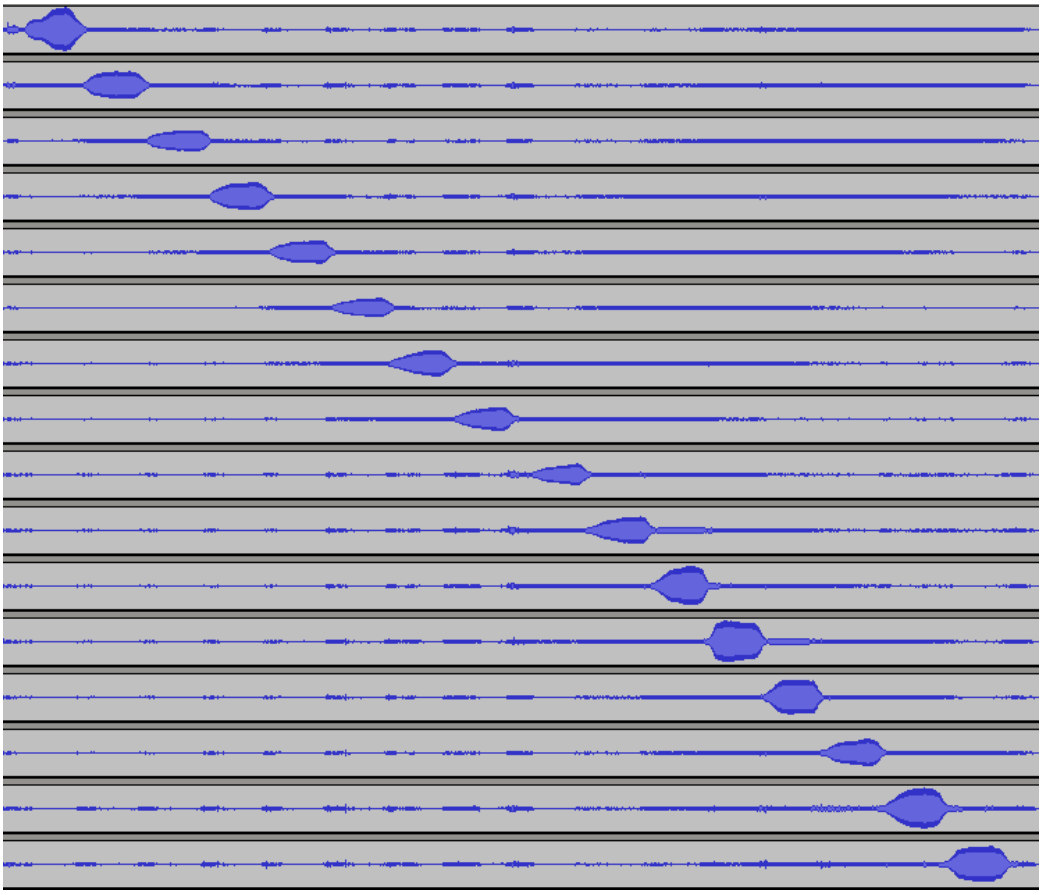


Figure 7.4.: Individual microphone test with a 2kHz tone.

For the next test one microphone was connected to the sound card with a 75cm ribbon cable and soundproofed with several blankets. It then recorded the noise of the system which can be seen in Figure 7.5.

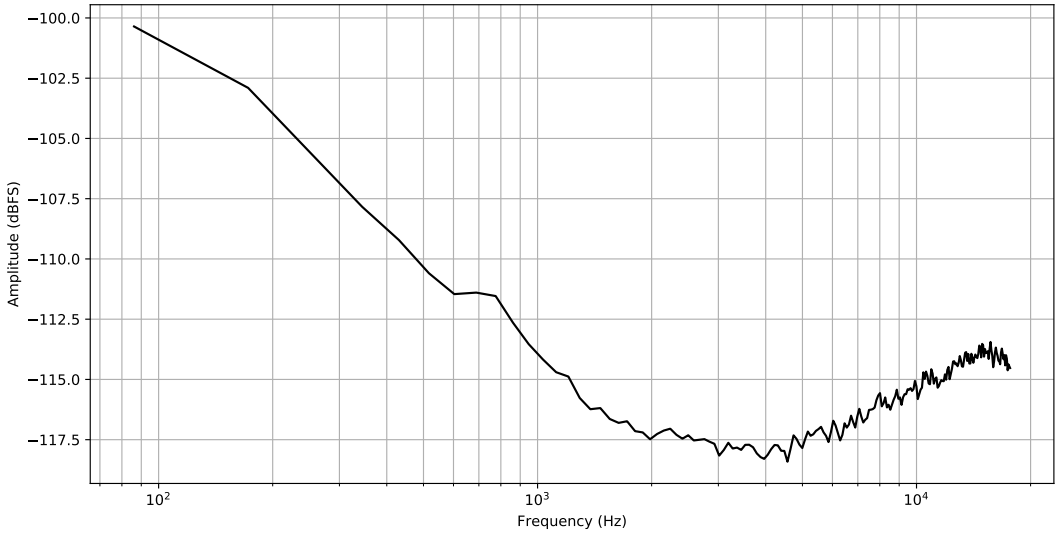


Figure 7.5.: Spectrum of the system's noise.

To verify that the blankets were soundproof enough a second recording was done without the blanket which can be seen in Figure 7.6. This showed that all the noise seen in Figure 7.5 is only system noise and not coming from any other source.

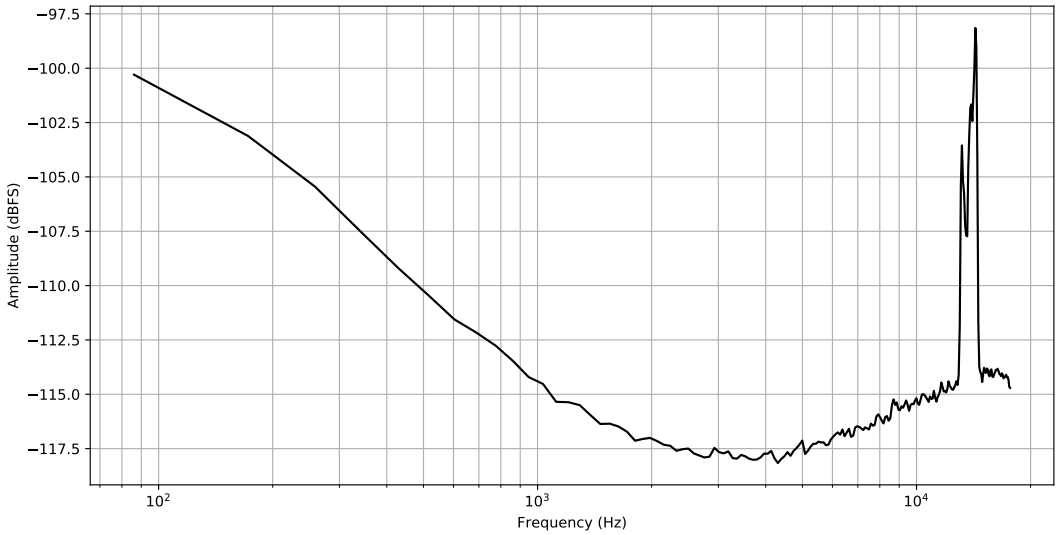


Figure 7.6.: Spectrum of the noise in the room without soundproofing.

The noise of the system above $3kHz$ shows the effect of the noise shaping Delta-Sigma-Converter inside the MEMS microphone. It also shows the cutoff frequency of the third decimation filter stage which is at about $18kHz$.

The last test was to evaluate the effect of different ribbon cable length. This test was done at a sample rate of 44.1kHz. Starting with a 2 meter long ribbon cable the cable was cut shorter for each individual measurement. A measurement consists of one microphone wrapped in a blanket for soundproofing connected to the sound card with the ribbon cable of various lengths. With this setup the noise spectrum was measured. Figure 7.7 shows these spectra

with different lengths. As there is no way to identify the different length in this figure (as there are not enough colors), a second figure, Figure 7.8, is provided showing the mean noise amplitude on the y-Axis and the ribbon cable length on the x-Axis.

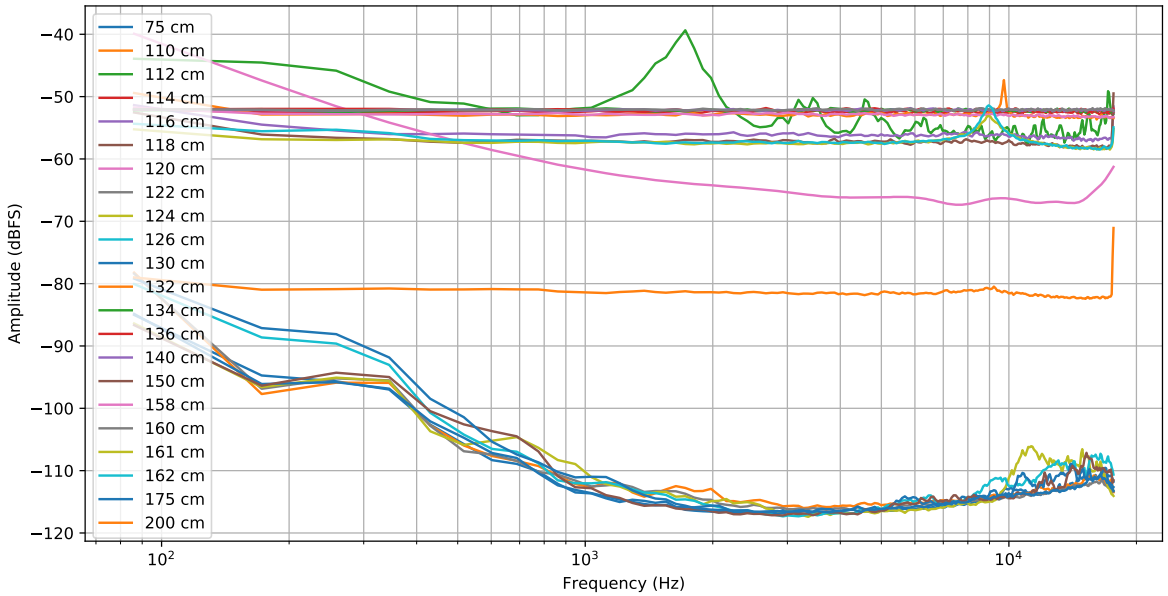


Figure 7.7.: Noise spectra with different ribbon cable lengths.

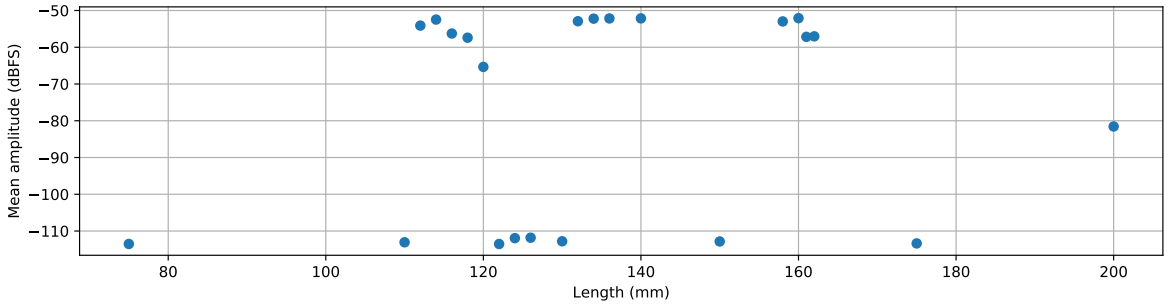


Figure 7.8.: Mean of the noise spectrum for different ribbon cable lengths.

Figure 7.8 shows that there are certain cable lengths with normal noise levels below -100dB and some with high noise levels. This can be caused by clock and data signal reflections at the ends of the transmission line. The result is a superposition of the original signal and the reflected one. If the ribbon cable has an unfavorable length the signals overlap destructively causing bit errors in the bit stream which results in noise.

Reflections can be minimized by adding source termination resistors. The value of the resistor depends on the transmission line and therefore on the cable length. As an exact length of the cable was not specified, this is not part of this thesis and as such it is just briefly mentioned here. Source termination resistors for the clock signals are present on the PCB but not used. Instead 0Ω jumpers were used. These can be replaced by any desired termination resistor. The numbers of these resistors are R19 to R26 and R30 to R37.

8. Summary

For this theses a sound card was developed. Firstly, a microphone type was selected, namely MEMS microphones. Research on these microphones was done and the functional principle of digital MEMS microphones was presented. The pulse density modulation (PDM) was also included in this research as it is the output signal of the microphones. As the second part of research the USB 2.0 protocol was closely examined along with the protocol extensions for the USB audio class. The gained knowledge was structured and presented in a chapter of this thesis. The next step was to select a processor which had enough calculation power to handle sixteen audio channels and includes a hardware USB 2.0 interface. A processor from the company XMOS was found along with a demo board containing the basic hardware and firmware building blocks needed for this thesis. The schematics of this demo board was then redesigned to meet the thesis' specifications. The resulting circuit was explained in its own chapter. From the schematics a PCB layout was designed and the firmware of the demo board was then examined and researched. Following this the firmware was modified and extended to fulfill the requirements of the changed hardware. The structure of the resulting firmware and the most important algorithms were then presented. Finally the PCB was ordered and once received it was assembled and soldered by hand. As a last step the proper operation of the sound card was ensured through some basic functional tests.

Appendix A.

Filter Characteristics

First & Second Decimator

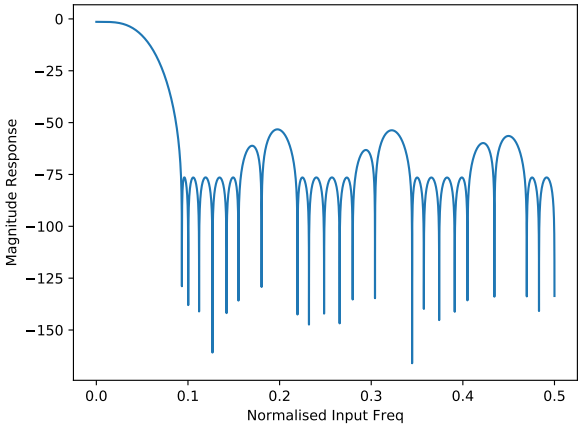


Figure A.1.: Magnitude response of the first decimator.

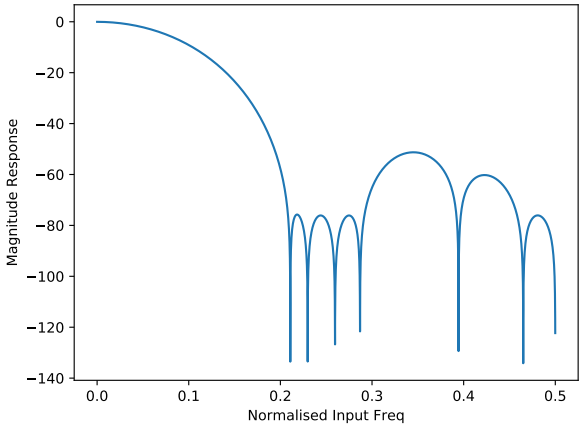


Figure A.2.: Magnitude response of the second decimator.

Third Decimator

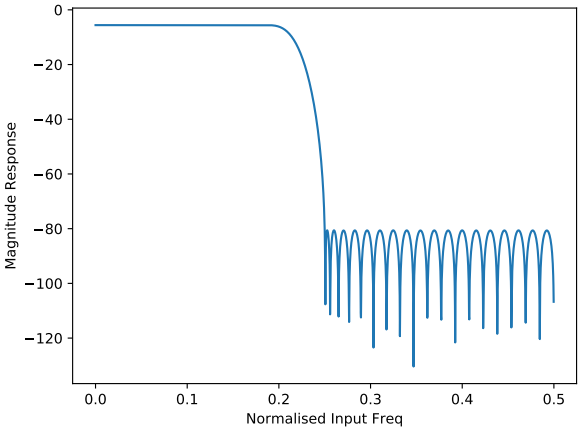


Figure A.3.: Magnitude response of the third decimator with a decimation factor of 2.

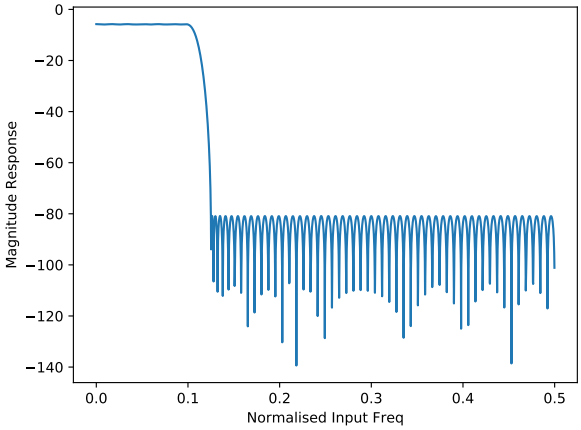


Figure A.4.: Magnitude response of the third decimator with a decimation factor of 4.

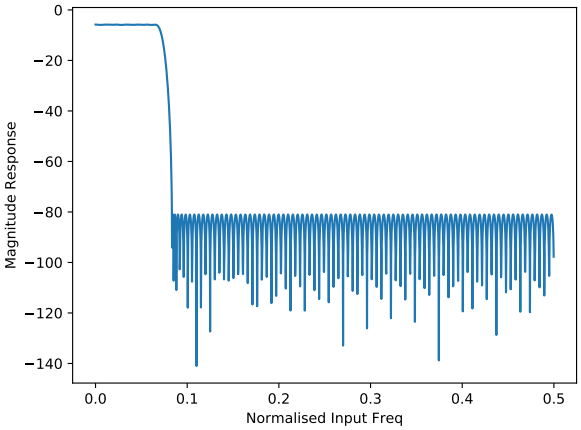


Figure A.5.: Magnitude response of the third decimator with a decimation factor of 6.

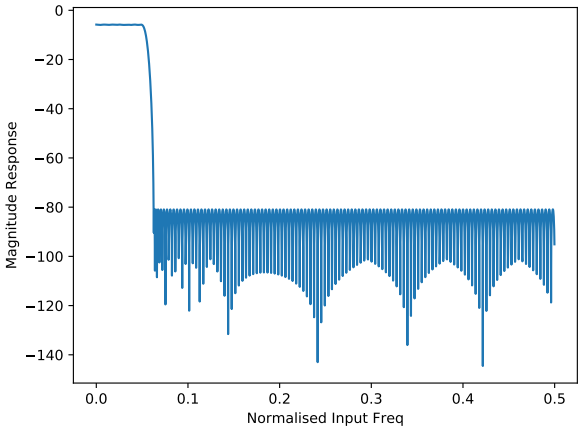


Figure A.6.: Magnitude response of the third decimator with a decimation factor of 8.

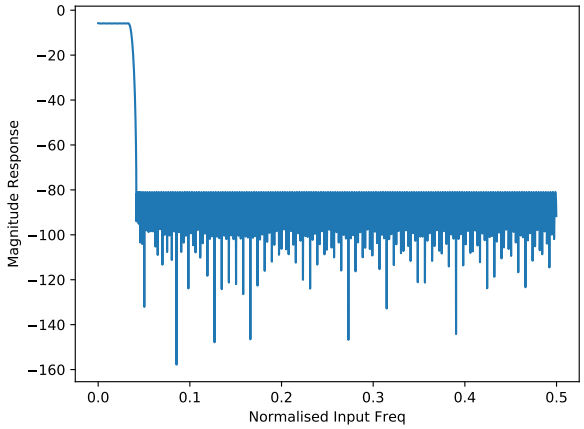


Figure A.7.: Overall magnitude response of the output with a decimation factor of 12.

Overall Magnitude Response

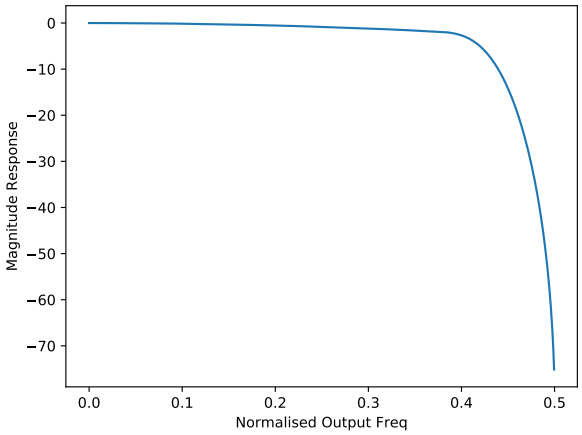


Figure A.8.: Overall magnitude response of the output with a decimation factor of 2.

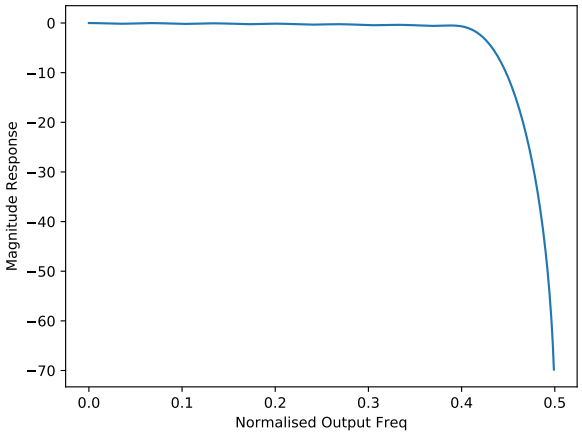


Figure A.9.: Overall magnitude response of the output with a decimation factor of 4.

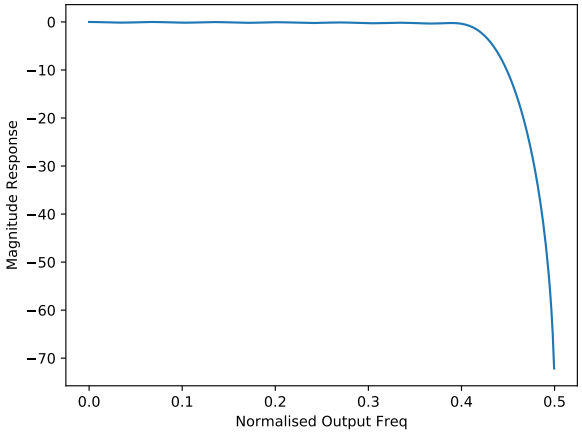


Figure A.10.: Overall magnitude response of the output with a decimation factor of 6.

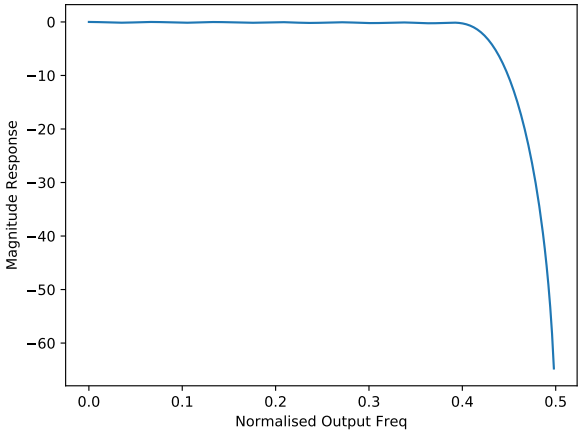


Figure A.11.: Overall magnitude response of the output with a decimation factor of 8.

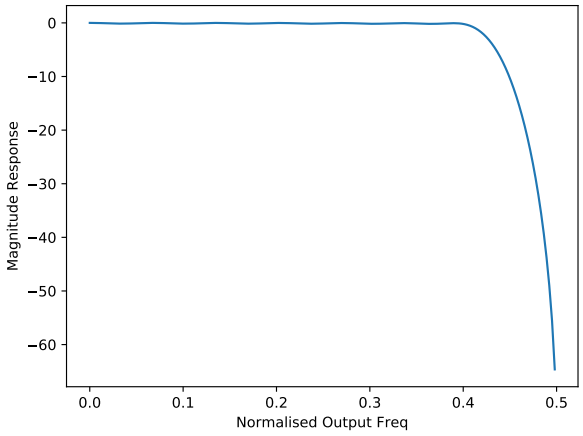
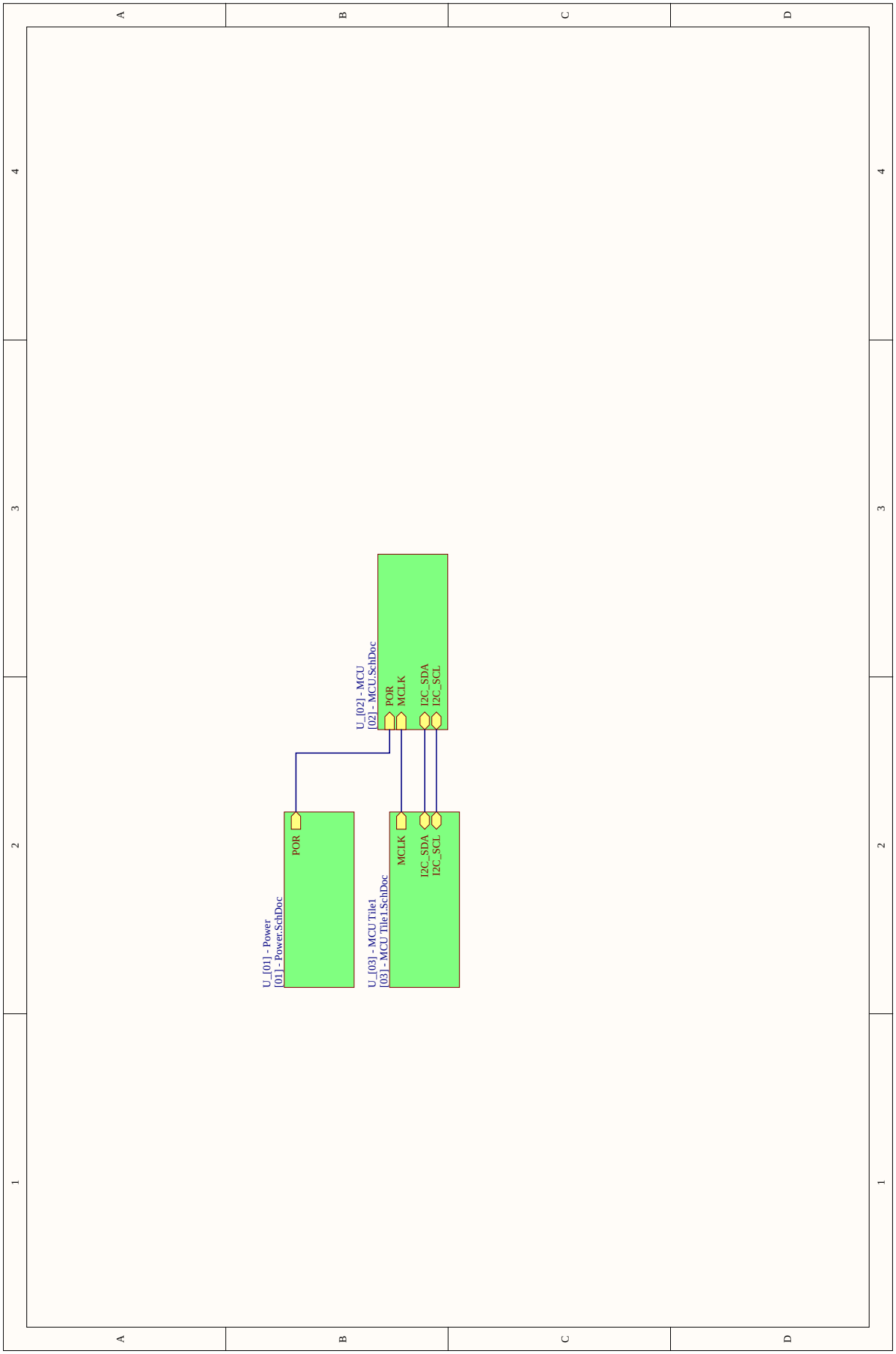


Figure A.12.: Overall magnitude response of the output with a decimation factor of 12.

Appendix B.

Schematics


A
B
C
D
1
2
3
4
U_[02] - MCU
[02] - MCU.SchDoc
POR
MCLK
I2C_SDA
I2C_SCL
U_[01] - Power
[01] - Power.SchDoc
POR
U_[03] - MCU Tile1
[03] - MCU Tile1.SchDoc
MCLK
I2C_SDA
I2C_SCL

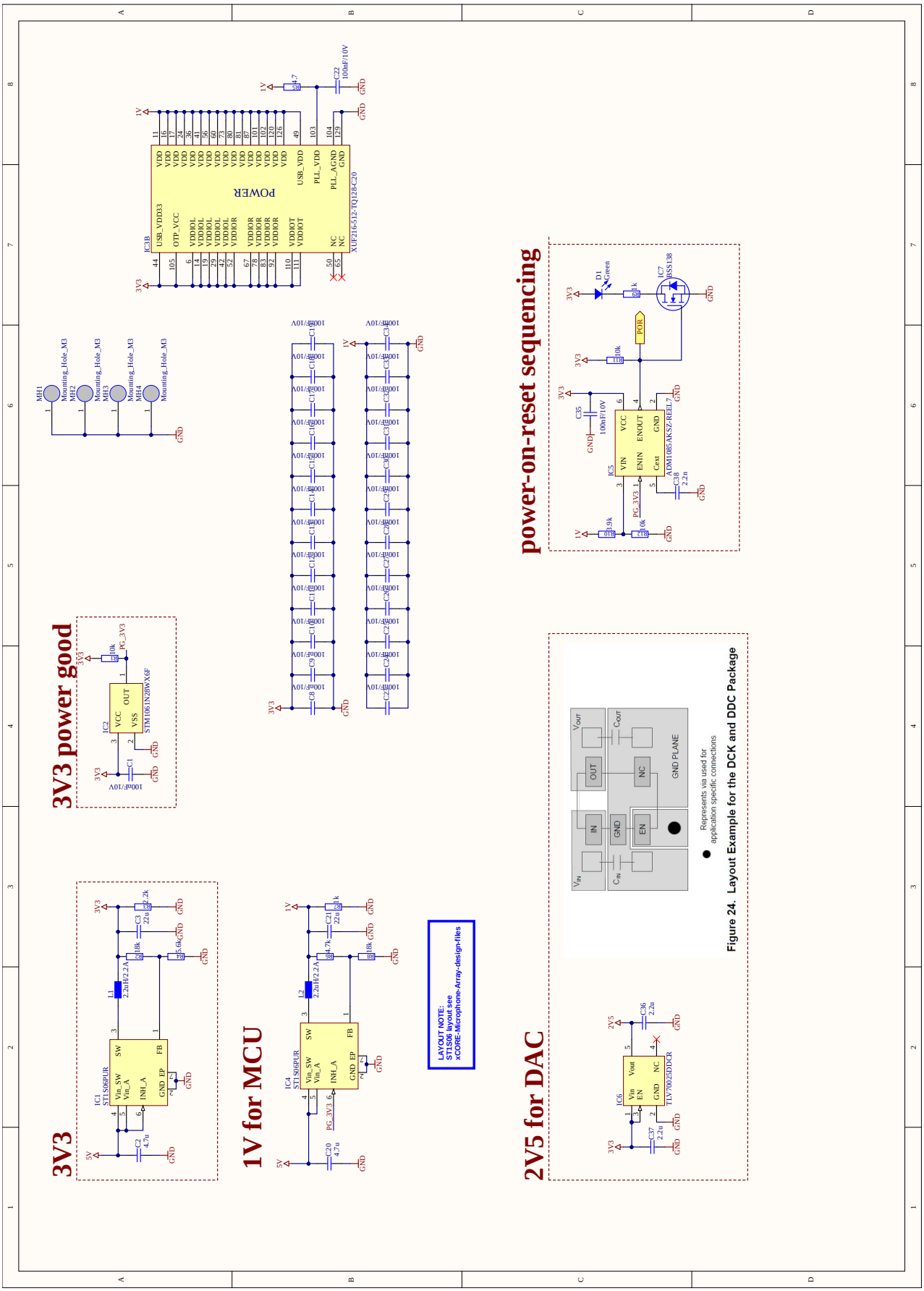
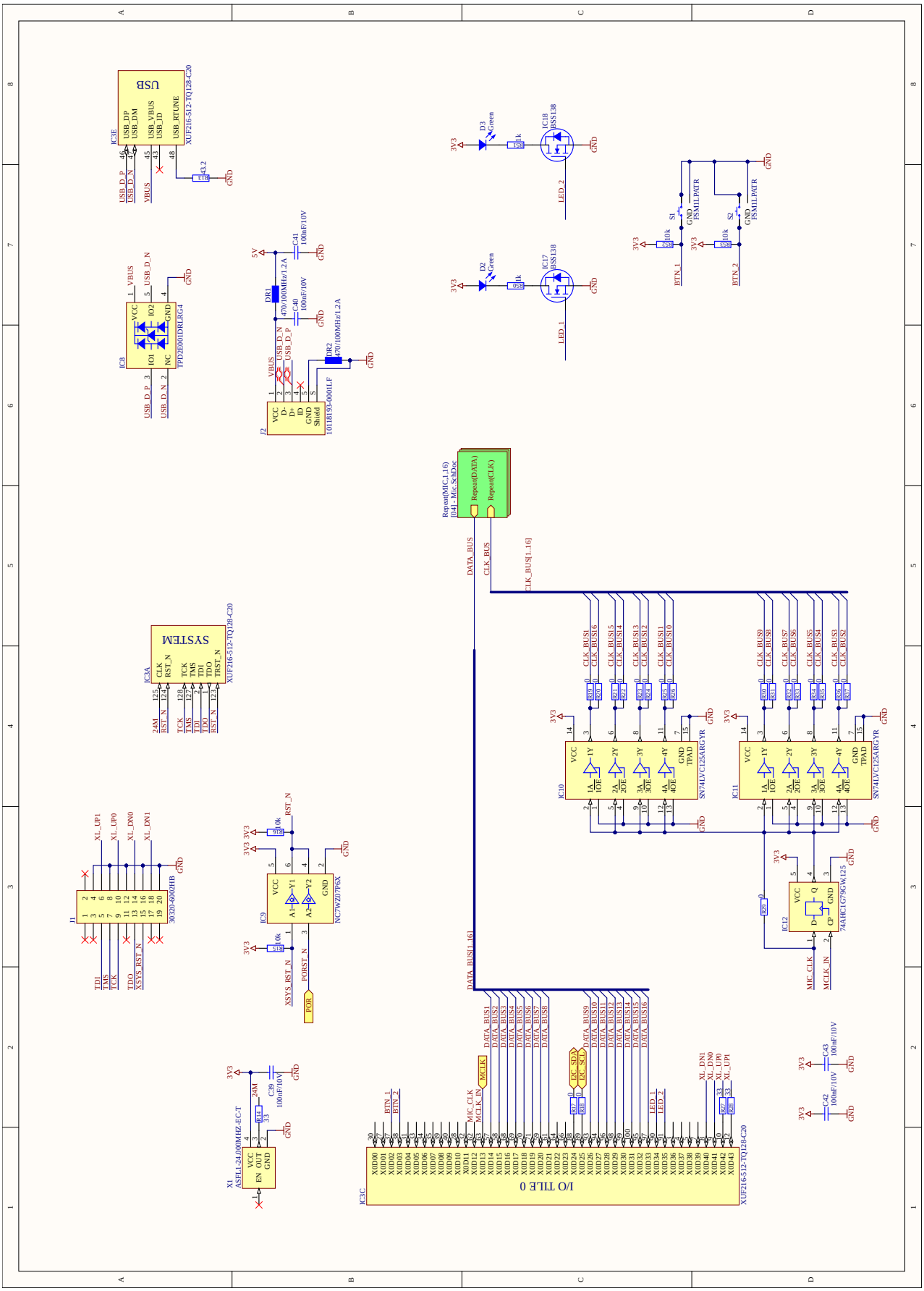
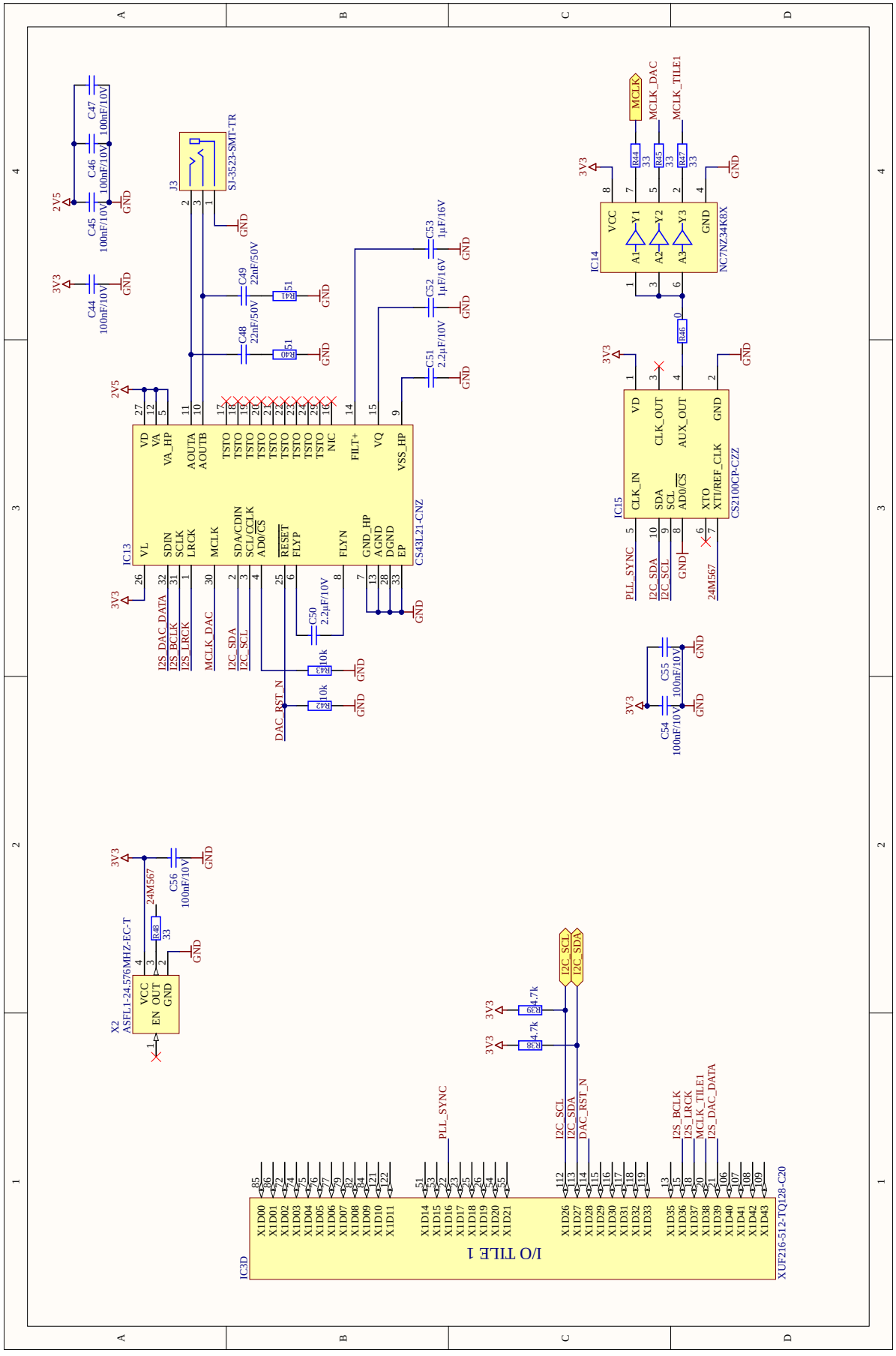
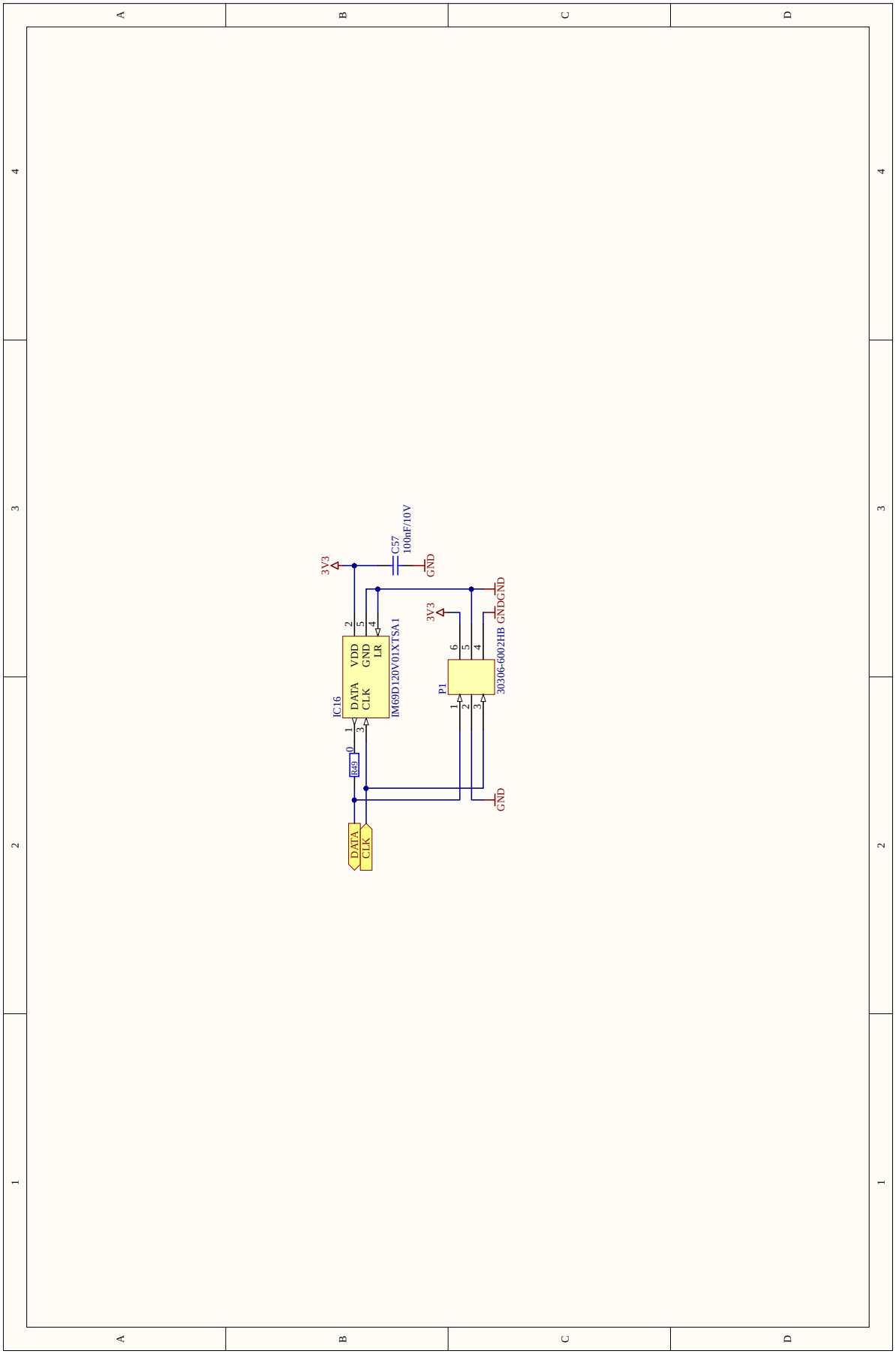


Figure 24. Layout Example for the DCK and DDC Package




IC3D
I/O TILE 1
XUF216-512-TQ128-C20
X1D00 X1D01 X1D02 X1D03 X1D04 X1D05 X1D06 X1D07 X1D08 X1D09 X1D10 X1D11
X1D14 X1D15 X1D16 X1D17 X1D18 X1D19 X1D20 X1D21
X1D26 X1D27 X1D28 X1D29 X1D30 X1D31 X1D32 X1D33
X1D35 X1D36 X1D37 X1D38 X1D39 X1D40 X1D41 X1D42 X1D43
PLL_SYNC
I2C_SCL
I2C_SDA
DAC_RST_N
I2S_BCLK
I2S_LRCK
MCLK_TILE1
I2S_DAC_DATA
X2
ASFL1-24.576MHZ-EC-T
VCC EN OUT GND
24M567
R48 33
C56 100nF/10V
IC13
CS43L21-CNZ
VL VD VA VA_HP SDIN SCLK LRCK MCLK SDA/CDIN SCL/CCLK AD0/CS RESET FLYP FLYN GND_HP AGND DGND EP AOUTA AOUTB TSTO NIC FILT+ VQ VSS_HP
I2S_DAC_DATA I2S_BCLK I2S_LRCK MCLK_DAC I2C_SDA I2C_SCL DAC_RST_N
C50 2.2µF/10V
R42 10k
R43 10k
C44 100nF/10V
C45 100nF/10V
C46 100nF/10V
C47 100nF/10V
J3
SJ-3523-SMT-TR
C48 22nF/50V
C49 22nF/50V
R40 51
R41 51
C51 2.2µF/10V
C52 1µF/16V
C53 1µF/16V
R38 4.7k
R39 4.7k
IC14
NC7NZ34K8X
VCC Y1 Y2 Y3 GND A1 A2 A3
MCLK MCLK_DAC MCLK_TILE1
R44 33
R45 33
R47 33
R46 0
IC15
CS2100CP-CZZ
CLK_IN VD CLK_OUT SDA SCL AD0/CS AUX_OUT XTO XTI/REF_CLK GND
PLL_SYNC I2C_SDA I2C_SCL
24M567
C54 100nF/10V
C55 100nF/10V
2V5
3V3
GND

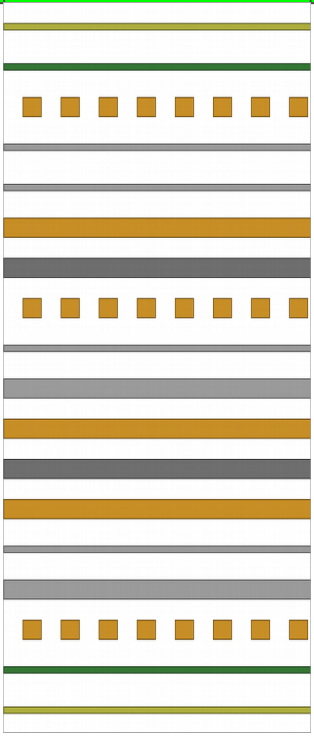

A
B
C
D
1
2
3
4
3V3
C57
100nF/10V
GND
3V3
GNDGND
IC16
DATA
CLK
VDD
GND
LR
IM69D120V01XTSA1
P1
30306-6002HB
R49
0
GND
DATA
CLK

Appendix C.

PCBs

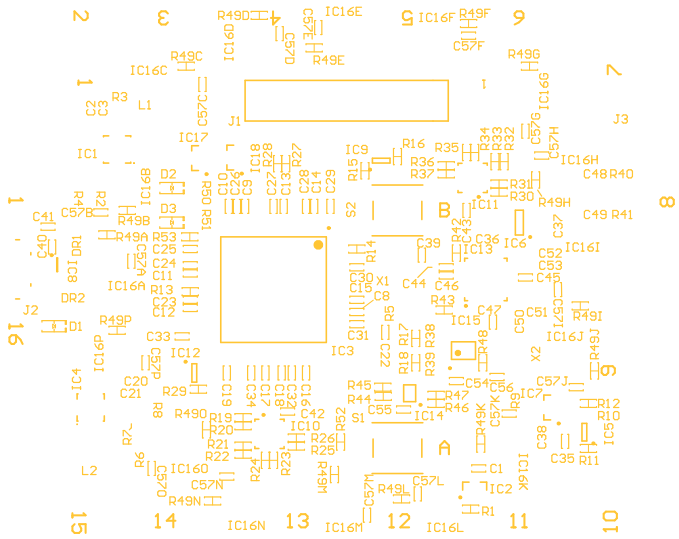
Layer Stack

Board Stack Report

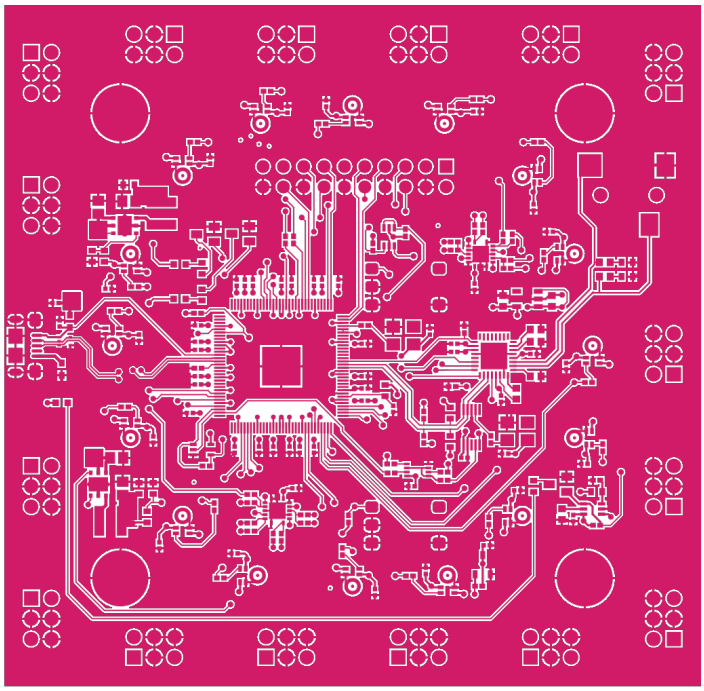
Stack Up		Layer Stack			
Layer	Board Layer Stack	Name	Material	Thickness	Constant
1		Top Paste			
2		Top Overlay			
3		Top Solder	Solder Resist	0,010mm	3,5
4		Component Side	Copper	0,035mm	
5		1080	FR-4	0,078mm	4,2
6		1080	FR-4	0,078mm	4,2
7		Ground Plane (GND)	Copper	0,035mm	
8		Dielectric 3	FR-4	0,400mm	4,2
9		Inner Layer 1	Copper	0,035mm	
10		1080	FR-4	0,127mm	4,2
11		1080	FR-4	0,127mm	4,2
12		Power Plane (VCC)	Copper	0,036mm	
13		Dielectric 5	FR-4	0,400mm	4,2
14		Ground Plane (GND)	Copper	0,035mm	
15		1080	FR-4	0,078mm	4,2
16		1080	FR-4	0,078mm	4,2
17		Solder Side	Copper	0,035mm	
18		Bottom Solder	Solder Resist	0,010mm	3,5
19		Bottom Overlay			
20		Bottom Paste			

Height : 1,597mm

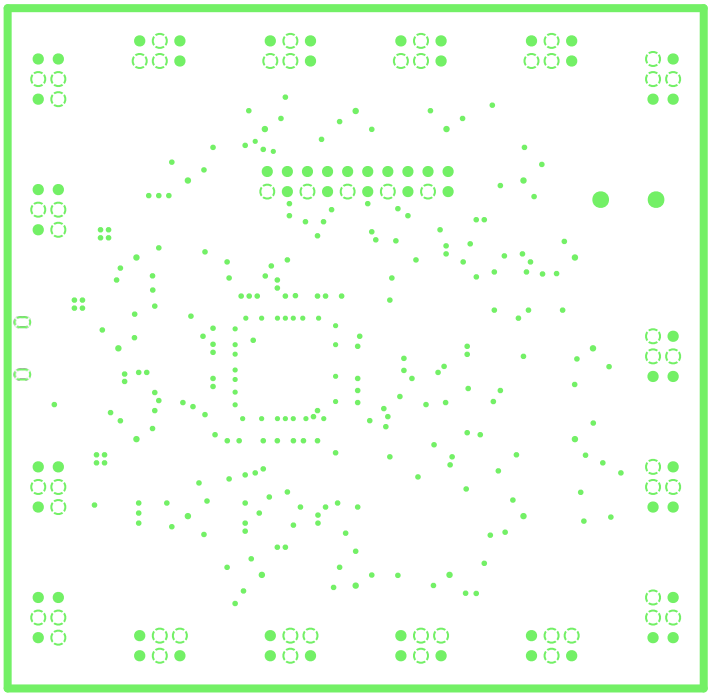
Layout



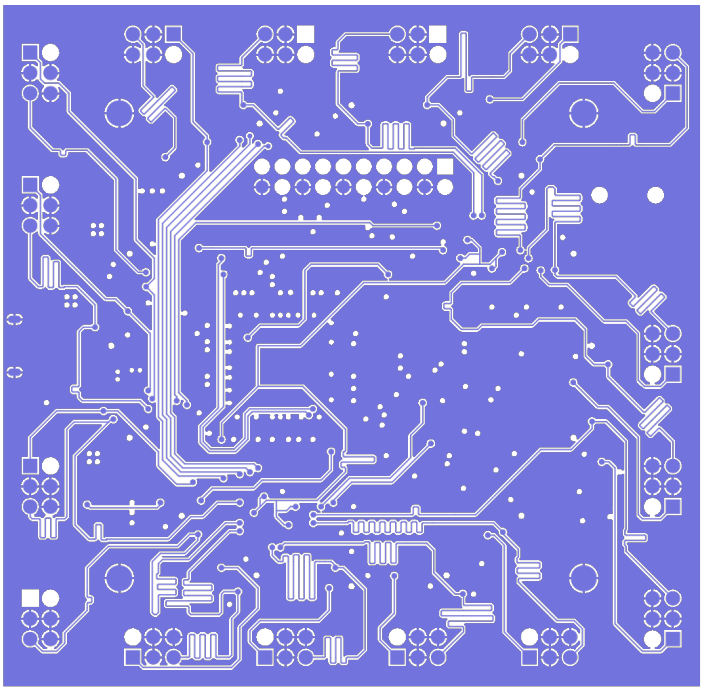
Layer 2: Top overlay



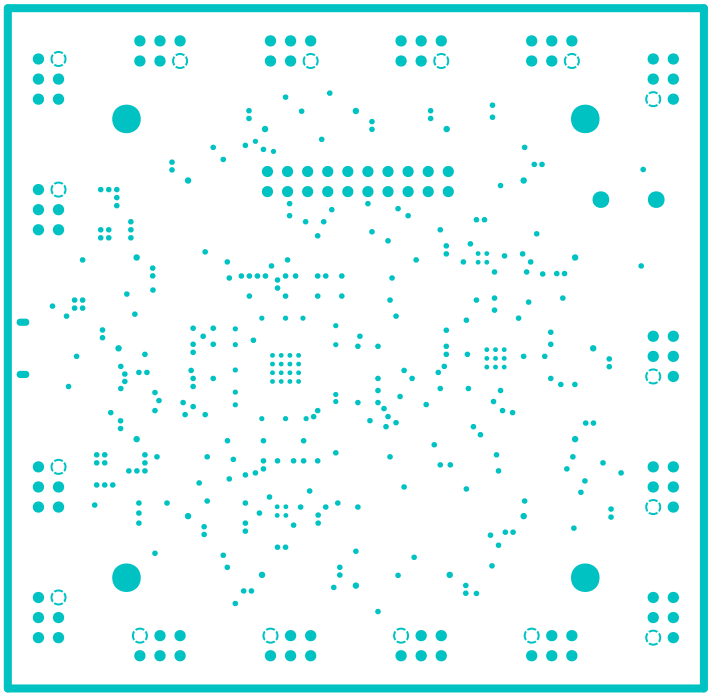
Layer 4: Top Layer



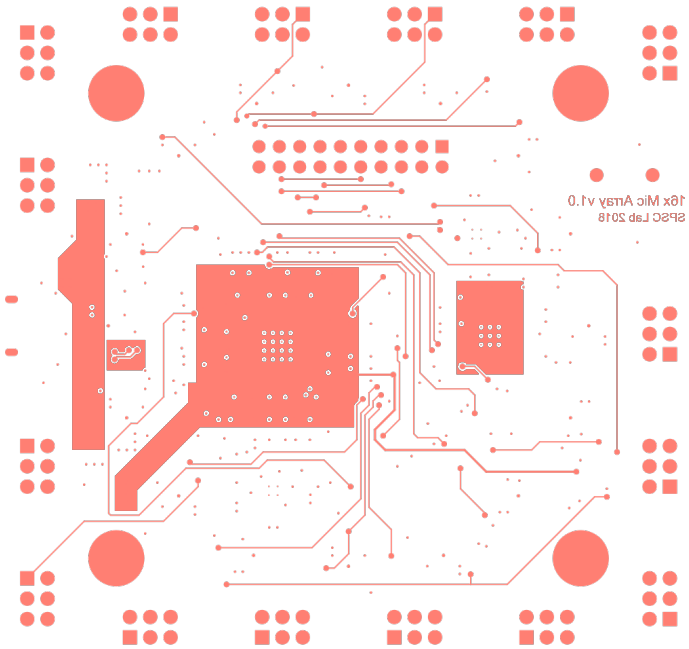
Layer 7 & 14: GND plane (inverted)



Layer 9: Inner Layer



Layer 12: VCC plane (inverted)



Layer 17: Bottom Layer

Description	Designator	Manufacturer Number	Qty.	Digi-Key Part Number
CAP CER 0.1UF 10V X5R 0402	C1, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, C22, C23, C24, C25, C26, C27, C28, C29, C30, C31, C32, C33, C34, C35, C39, C40, C41, C42, C43, C44, C45, C46, C47, C54, C55, C56, C57A, C57B, C57C, C57D, C57E, C57F, C57G, C57H, C57I, C57J, C57K, C57L, C57M, C57N, C57O, C57P	CL05A104KP5NNNC	55	1276-1022-1-ND
CAP CER 22UF 6.3V X5R 0805	C2, C20	CL21A475KOFNNNE	2	1276-1065-1-ND
CAP CER 2.2UF 10V X7R 0603	C36, C37	CL10B225KP8NNNC	2	1276-1134-1-ND
IC D-TYPE POS TRG SNGL 5TSSOP	IC12	74AHC1G79GW,125	1	1727-6019-1-ND
MIC MEMS DIGITAL	IC16A, IC16B, IC16C, IC16D, IC16E, IC16F, IC16G, IC16H, IC16I, IC16J, IC16K, IC16L, IC16M, IC16N, IC16O, IC16P	IM69D120V01XTSA1	16	IM69D120V01XTSA1CT-ND
CAP CER 1UF 16V X7R 0603	C52, C53	CL10B105KO8VPNC	2	1276-6613-1-ND
24.576MHz Oscillator	X2	ASFL1-24.576MHZ-EC-T	1	300-8253-1-ND
IC MCU 32BIT 2MB FLASH 128TQFP	IC3	XUF216-512-TQ128-C20	1	880-1122-ND
TVS DIODE 5.5VWM 100VC SOT5	IC8	TPD2E001DRLRG4	1	296-35960-1-ND
MOSFET N-CH 50V 220MA SOT-23	IC7, IC17, IC18	BSS138	3	BSS138CT-ND
IC REG BUCK ADJ 1.5A SYNC DFN6	IC1, IC4	ST1S06PUR	2	497-5802-1-ND
Audio D/A Converter	IC13	CS43L21-CNZ	1	598-1187-ND
CAP CER 22UF 6.3V X5R 0805	C3, C21	CL21A226MQCLQNC	2	1276-2412-1-ND
IC SIMPLE SEQUENCER OD SC70-6	IC5	ADM1085AKSZ-REEL7	1	ADM1085AKSZ-REEL7CT-ND
IC CLK MULT FRACTIONAL N 10MSOP	IC15	CS2100CP-CZZ	1	598-1750-ND
IC REG LINEAR 2.5V 200MA SOT23-5	IC6	TLV70025DDCR	1	296-32411-1-ND
Header 3x2	P1A, P1B, P1C, P1D, P1E, P1F, P1G, P1H, P1I, P1J, P1K, P1L, P1M, P1N, P1O, P1P	30306-6002HB	16	3M15451-ND
IC BUFFER TRPL SCHM TRIG US8	IC14	NC7NZ34K8X	1	NC7NZ34K8XCT-ND
IC VOLT DETECTOR 2.8V LP SOT-23	IC2	STM1061N28WX6F	1	497-4955-1-ND
24MHz Oscillator	X1	ASFL1-24.000MHZ-EC-T	1	300-8252-1-ND

Description	Designator	Manufacturer Number	Qty.	Digi-Key Part Number
Header, 10-Pin, Dual row	J1	30320-6002HB	1	3M11932-ND
SWITCH TACTILE SPST-NO	S1, S2	FSM1LPATR	2	450-1760-1-ND
IC BUS BUFF TRI-ST QD 14VQFN	IC10, IC11	SN74LVC125ARGYR	2	296-13961-1-ND
FIXED IND 2.2UH 2.2A 50.4 MOHM	L1, L2	NRS4018T2R2MDGJ	2	587-2890-1-ND
3.50mm Headphone Phone Jack Stereo Connector Solder	J3	SJ-3523-SMT-TR	1	CP-3523SJCT-ND
CAP CER 2.2UF 10V X5R 0603	C50, C51	CL10A225KP8NNNC	2	1276-1085-1-ND
USB - micro B Connector	J2	10118193-0001LF	1	609-4616-1-ND
CAP CER 0.022UF 50V X7R 0603	C48, C49	CL10B223JB8NNNC	2	1276-1996-1-ND
IC BUFF DL UHS O/DRAIN SC706	IC9	NC7WZ07P6X	1	NC7WZ07P6XCT-ND
FERRITE BEAD 470 OHM 0603 1LN	DR1, DR2	CIC10P471NC	2	1276-6358-1-ND
RES SMD 43.2 OHM 1% 1/16W 0402	R13	RC0402FR-0743R2L	1	YAG3161CT-ND
RES SMD 3.9K OHM 1% 1/10W 0603	R10	RC0603FR-073K9L	1	311-3.90KHRCT-ND
CAP CER 2200PF 50V X7R 0603	C38	CL10B222KB8NNNC	1	1276-1110-1-ND
RES SMD 4.7 OHM 1% 1/10W 0603	R5	RC0603FR-074R7L	1	YAG3371CT-ND
RES SMD 5.6K OHM 1% 1/10W 0603	R4	RC0603FR-075K6L	1	311-5.60KHRCT-ND
RES SMD 18K OHM 1% 1/10W 0603	R2, R8	RC0603FR-0718KL	2	311-18.0KHRCT-ND
RES SMD 51 OHM 1% 1/10W 0603	R40, R41	RC0603FR-0751RL	2	311-51.0HRCT-ND
RES SMD 2.2K OHM 1% 1/10W 0603	R3	RC0603FR-072K2L	1	311-2.20KHRCT-ND
RES SMD 33 OHM 1% 1/16W 0402	R14, R27, R28, R44, R45, R47, R48	RC0402FR-0733RL	7	311-33.0LRCT-ND
RES SMD 4.7K OHM 1% 1/10W 0603	R6, R38, R39	RC0603FR-074K7L	3	311-4.70KHRCT-ND
RES SMD 1K OHM 1% 1/10W 0603	R7, R9, R50, R51	RC0603FR-071KL	4	311-1.00KHRCT-ND
RES SMD 0 OHM JUMPER 1/16W 0402	R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R29, R30, R31, R32, R33, R34, R35, R36, R37, R46, R49A, R49B, R49C, R49D, R49E, R49F, R49G, R49H, R49I, R49J, R49K, R49L, R49M, R49N, R49O, R49P	RC0402JR-070RL	36	311-0.0JRCT-ND
LED GREEN CLEAR 0603 SMD	D1, D2, D3	LTST-C191KGKT	3	160-1446-1-ND
RES SMD 10K OHM 1% 1/16W 0402	R1, R11, R12, R15, R16, R42, R43, R52, R53	RC0402FR-0710KL	9	311-10.0KLRCT-ND

Appendix D.

USB Descriptors

The following is the output of the linux program `lsusb`:

```
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass          239 Miscellaneous Device
  bDeviceSubClass         2 ?
  bDeviceProtocol         1 Interface Association
  bMaxPacketSize0        64
  idVendor           0x20b1 XMOS Ltd
  idProduct          0x0008
  bcdDevice            6.f2
  iManufacturer           1 XMOS
  iProduct                3 XMOS Microphone Array UAC2.0
  iSerial                 0
  bNumConfigurations      2
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength          375
    bNumInterfaces          4
    bConfigurationValue     1
    iConfiguration          0
    bmAttributes         0x80
      (Bus Powered)
    MaxPower              500mA
    Interface Association:
      bLength                8
      bDescriptorType       11
      bFirstInterface        0
      bInterfaceCount        3
      bFunctionClass         1 Audio
      bFunctionSubClass      0
      bFunctionProtocol     32
      iFunction              0
    Interface Descriptor:
      bLength                9
      bDescriptorType        4
      bInterfaceNumber       0
      bAlternateSetting      0
      bNumEndpoints          0
      bInterfaceClass        1 Audio
      bInterfaceSubClass     1 Control Device
      bInterfaceProtocol    32
      iInterface             3 XMOS Microphone Array
      UAC2.0
      AudioControl Interface Descriptor:
        bLength                9
        bDescriptorType       36
        bDescriptorSubtype     1 (HEADER)
        bcdADC              2.00
        bCategory              8
        wTotalLength         175
        bmControl           0x00
      AudioControl Interface Descriptor:
        bLength                8
        bDescriptorType       36
        bDescriptorSubtype    10 (CLOCK_SOURCE)
        bClockID              41
        bmAttributes        0x03 Internal programmable
        Clock
        bmControls          0x07
          Clock Frequency Control (read/write)
          Clock Validity Control (read-only)
        bAssocTerminal         0
        iClockSource           9 XMOS Internal Clock
      AudioControl Interface Descriptor:
        bLength                8
        bDescriptorType       36
        bDescriptorSubtype    11 (CLOCK_SELECTOR)
        bUnitID               40
        bNrInPins              1
        baCSourceID( 0)       41
        bmControls          0x03
          Clock Selector Control (read/write)
        iClockSelector         8 XMOS Clock Selector
      AudioControl Interface Descriptor:
        bLength               17
        bDescriptorType       36
        bDescriptorSubtype     2 (INPUT_TERMINAL)
        bTerminalID            2
        wTerminalType      0x0101 USB Streaming
        bAssocTerminal         0
        bCSourceID            40
        bNrChannels            2
        bmChannelConfig   0x00000000
        bmControls    0x0000
        iChannelNames         11 Analogue 1
        iTerminal              6 XMOS Microphone Array
      UAC2.0
      AudioControl Interface Descriptor:
        bLength               18
        bDescriptorType       36
        bDescriptorSubtype     6 (FEATURE_UNIT)
        bUnitID               10
        bSourceID              2
        bmaControls( 0)      0x0000000f
          Mute Control (read/write)
          Volume Control (read/write)
        bmaControls( 1)      0x0000000f
          Mute Control (read/write)
          Volume Control (read/write)
        bmaControls( 2)      0x0000000f
          Mute Control (read/write)
          Volume Control (read/write)
        iFeature               0
      AudioControl Interface Descriptor:
        bLength               12
        bDescriptorType       36
        bDescriptorSubtype     3 (OUTPUT_TERMINAL)
        bTerminalID           20
        wTerminalType      0x0301 Speaker
        bAssocTerminal         0
        bSourceID             10
        bCSourceID            40
        bmControls         0x0000
        iTerminal              0
      AudioControl Interface Descriptor:
        bLength               17
        bDescriptorType       36
        bDescriptorSubtype     2 (INPUT_TERMINAL)
        bTerminalID            1
        wTerminalType      0x0201 Microphone
        bAssocTerminal         0
        bCSourceID            40
        bNrChannels           16
        bmChannelConfig   0x00000000
        bmControls    0x0000
        iChannelNames         13
```

```
      iTerminal              0
    AudioControl Interface Descriptor:
      bLength                74
      bDescriptorType        36
      bDescriptorSubtype      6 (FEATURE_UNIT)
      bUnitID                11
      bSourceID               1
      bmaControls( 0)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 1)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 2)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 3)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 4)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 5)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 6)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 7)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 8)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 9)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(10)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(11)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(12)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(13)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(14)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(15)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(16)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      iFeature               0
    AudioControl Interface Descriptor:
      bLength               12
      bDescriptorType       36
      bDescriptorSubtype     3 (OUTPUT_TERMINAL)
      bTerminalID           22
      wTerminalType      0x0101 USB Streaming
      bAssocTerminal         0
      bSourceID             11
      bCSourceID            40
      bmControls         0x0000
      iTerminal              7 XMOS Microphone Array
    UAC2.0
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        1
    bAlternateSetting       0
    bNumEndpoints           0
    bInterfaceClass         1 Audio
    bInterfaceSubClass      2 Streaming
    bInterfaceProtocol     32
    iInterface              4 XMOS Microphone Array
    UAC2.0
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        1
    bAlternateSetting       1
    bNumEndpoints           1
    bInterfaceClass         1 Audio
    bInterfaceSubClass      2 Streaming
    bInterfaceProtocol     32
    iInterface              4 XMOS Microphone Array
    UAC2.0
    AudioStreaming Interface Descriptor:
      bLength                16
      bDescriptorType        36
      bDescriptorSubtype      1 (AS_GENERAL)
      bTerminalLink           2
      bmControls           0x00
      bFormatType             1
      bmFormats         0x00000001
        PCM
      bNrChannels             2
      bmChannelConfig   0x00000000
      iChannelNames          11 Analogue 1
    AudioStreaming Interface Descriptor:
      bLength                 6
      bDescriptorType        36
      bDescriptorSubtype      2 (FORMAT_TYPE)
      bFormatType             1 (FORMAT_TYPE_I)
      bSubslotSize            4
      bBitResolution         24
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x01  EP 1 OUT
      bmAttributes            5
        Transfer Type            Isochronous
        Synch Type               Asynchronous
        Usage Type               Data
      wMaxPacketSize     0x0038  1x 56 bytes
      bInterval               1
      AudioControl Endpoint Descriptor:
        bLength               8
        bDescriptorType      37
        bDescriptorSubtype    1 (EP_GENERAL)
        bmAttributes       0x00
        bmControls         0x00
        bLockDelayUnits       2 Decoded PCM samples
        wLockDelay            8
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        1
    bAlternateSetting       2
    bNumEndpoints           1
    bInterfaceClass         1 Audio
    bInterfaceSubClass      2 Streaming
    bInterfaceProtocol     32
    iInterface              4 XMOS Microphone Array
    UAC2.0
    AudioStreaming Interface Descriptor:
      bLength                16
      bDescriptorType        36
      bDescriptorSubtype      1 (AS_GENERAL)
      bTerminalLink           2
      bmControls           0x00
      bFormatType             1
      bmFormats         0x00000001
        PCM
      bNrChannels             2
      bmChannelConfig   0x00000000
      iChannelNames          11 Analogue 1
    AudioStreaming Interface Descriptor:
      bLength                 6
      bDescriptorType        36
      bDescriptorSubtype      2 (FORMAT_TYPE)
      bFormatType             1 (FORMAT_TYPE_I)
      bSubslotSize            2
      bBitResolution         16
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x01  EP 1 OUT
      bmAttributes            5
        Transfer Type            Isochronous
        Synch Type               Asynchronous
        Usage Type               Data
      wMaxPacketSize     0x001c  1x 28 bytes
      bInterval               1
      AudioControl Endpoint Descriptor:
        bLength               8
        bDescriptorType      37
        bDescriptorSubtype    1 (EP_GENERAL)
        bmAttributes       0x00
        bmControls         0x00
        bLockDelayUnits       2 Decoded PCM samples
        wLockDelay            8
```

```
      Interface Descriptor:
        bLength                 9
        bDescriptorType         4
        bInterfaceNumber        2
        bAlternateSetting       0
        bNumEndpoints           0
        bInterfaceClass         1 Audio
        bInterfaceSubClass      2 Streaming
        bInterfaceProtocol     32
        iInterface              5 XMOS Microphone Array
      UAC2.0
      Interface Descriptor:
        bLength                 9
        bDescriptorType         4
        bInterfaceNumber        2
        bAlternateSetting       1
        bNumEndpoints           1
        bInterfaceClass         1 Audio
        bInterfaceSubClass      2 Streaming
        bInterfaceProtocol     32
        iInterface              5 XMOS Microphone Array
      UAC2.0
        AudioStreaming Interface Descriptor:
          bLength                16
          bDescriptorType        36
          bDescriptorSubtype      1 (AS_GENERAL)
          bTerminalLink          22
          bmControls           0x00
          bFormatType             1
          bmFormats         0x00000001
            PCM
          bNrChannels            16
          bmChannelConfig   0x00000000
          iChannelNames          13
        AudioStreaming Interface Descriptor:
          bLength                 6
          bDescriptorType        36
          bDescriptorSubtype      2 (FORMAT_TYPE)
          bFormatType             1 (FORMAT_TYPE_I)
          bSubslotSize            4
          bBitResolution         24
        Endpoint Descriptor:
          bLength                 7
          bDescriptorType         5
          bEndpointAddress     0x81  EP 1 IN
          bmAttributes           37
            Transfer Type            Isochronous
            Synch Type               Asynchronous
            Usage Type               Implicit feedback Data
          wMaxPacketSize     0x01c0  1x 448 bytes
          bInterval               1
          AudioControl Endpoint Descriptor:
            bLength                8
            bDescriptorType       37
            bDescriptorSubtype     1 (EP_GENERAL)
            bmAttributes        0x00
            bmControls          0x00
            bLockDelayUnits        2 Decoded PCM samples
            wLockDelay             8
      Interface Descriptor:
        bLength                 9
        bDescriptorType         4
        bInterfaceNumber        3
        bAlternateSetting       0
        bNumEndpoints           0
        bInterfaceClass       254 Application Specific
      Interface
        bInterfaceSubClass      1 Device Firmware Update
        bInterfaceProtocol      1
        iInterface             10 XMOS DFU
        Device Firmware Upgrade Interface Descriptor:
          bLength                             9
          bDescriptorType                    33
          bmAttributes                        7
            Will Not Detach
            Manifestation Tolerant
            Upload Supported
            Download Supported
          wDetachTimeout                    250 milliseconds
          wTransferSize                      64 bytes
          bcdDFUVersion                    1.10
    Configuration Descriptor:
      bLength                 9
      bDescriptorType         2
      wTotalLength          375
      bNumInterfaces          4
      bConfigurationValue     1
      iConfiguration          0
      bmAttributes         0x80
            (Bus Powered)
      MaxPower              500mA
      Interface Association:
        bLength                 8
        bDescriptorType        11
        bFirstInterface         0
        bInterfaceCount         3
        bFunctionClass          1 Audio
        bFunctionSubClass       0
        bFunctionProtocol      32
        iFunction               0
      Interface Descriptor:
        bLength                 9
        bDescriptorType         4
        bInterfaceNumber        0
        bAlternateSetting       0
        bNumEndpoints           0
        bInterfaceClass         1 Audio
        bInterfaceSubClass      1 Control Device
        bInterfaceProtocol     32
        iInterface              3 XMOS Microphone Array
      UAC2.0
        AudioControl Interface Descriptor:
          bLength                 9
          bDescriptorType        36
          bDescriptorSubtype      1 (HEADER)
          bcdADC               2.00
          bCategory               8
          wTotalLength          175
          bmControl           0x00
        AudioControl Interface Descriptor:
          bLength                 8
          bDescriptorType        36
          bDescriptorSubtype     10 (CLOCK_SOURCE)
          bClockID               41
          bmAttributes         0x03 Internal programmable
      Clock
          bmControls           0x07
            Clock Frequency Control (read/write)
            Clock Validity Control (read-only)
          bAssocTerminal          0
          iClockSource            9 XMOS Internal Clock
        AudioControl Interface Descriptor:
          bLength                 8
          bDescriptorType        36
          bDescriptorSubtype     11 (CLOCK_SELECTOR)
          bUnitID                40
          bNrInPins               1
          baCSourceID( 0)        41
          bmControls           0x03
            Clock Selector Control (read/write)
          iClockSelector          8 XMOS Clock Selector
        AudioControl Interface Descriptor:
          bLength                17
          bDescriptorType        36
          bDescriptorSubtype      2 (INPUT_TERMINAL)
          bTerminalID             2
          wTerminalType      0x0101 USB Streaming
          bAssocTerminal          0
          bCSourceID             40
          bNrChannels             2
          bmChannelConfig   0x00000000
          bmControls    0x0000
          iChannelNames          11 Analogue 1
          iTerminal               6 XMOS Microphone Array
      UAC2.0
        AudioControl Interface Descriptor:
          bLength                18
          bDescriptorType        36
          bDescriptorSubtype      6 (FEATURE_UNIT)
          bUnitID                10
          bSourceID               2
          bmaControls( 0)      0x0000000f
            Mute Control (read/write)
            Volume Control (read/write)
          bmaControls( 1)      0x0000000f
            Mute Control (read/write)
            Volume Control (read/write)
          bmaControls( 2)      0x0000000f
            Mute Control (read/write)
            Volume Control (read/write)
          iFeature                0
        AudioControl Interface Descriptor:
          bLength                12
          bDescriptorType        36
          bDescriptorSubtype      3 (OUTPUT_TERMINAL)
          bTerminalID            20
          wTerminalType      0x0301 Speaker
          bAssocTerminal          0
```

```
      bSourceID             10
      bCSourceID            40
      bmControls          0x0000
      iTerminal              0
    AudioControl Interface Descriptor:
      bLength               17
      bDescriptorType       36
      bDescriptorSubtype     2 (INPUT_TERMINAL)
      bTerminalID            1
      wTerminalType      0x0201 Microphone
      bAssocTerminal         0
      bCSourceID            40
      bNrChannels           16
      bmChannelConfig   0x00000000
      bmControls    0x0000
      iChannelNames         13
      iTerminal              0
    AudioControl Interface Descriptor:
      bLength               74
      bDescriptorType       36
      bDescriptorSubtype     6 (FEATURE_UNIT)
      bUnitID               11
      bSourceID              1
      bmaControls( 0)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 1)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 2)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 3)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 4)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 5)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 6)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 7)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 8)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls( 9)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(10)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(11)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(12)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(13)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(14)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(15)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      bmaControls(16)      0x0000000f
        Mute Control (read/write)
        Volume Control (read/write)
      iFeature               0
    AudioControl Interface Descriptor:
      bLength               12
      bDescriptorType       36
      bDescriptorSubtype     3 (OUTPUT_TERMINAL)
      bTerminalID           22
      wTerminalType      0x0101 USB Streaming
      bAssocTerminal         0
      bSourceID             11
      bCSourceID            40
      bmControls         0x0000
      iTerminal              7 XMOS Microphone Array
    UAC2.0
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        1
    bAlternateSetting       0
    bNumEndpoints           0
    bInterfaceClass         1 Audio
    bInterfaceSubClass      2 Streaming
    bInterfaceProtocol     32
    iInterface              4 XMOS Microphone Array
    UAC2.0
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        1
    bAlternateSetting       1
    bNumEndpoints           1
    bInterfaceClass         1 Audio
    bInterfaceSubClass      2 Streaming
    bInterfaceProtocol     32
    iInterface              4 XMOS Microphone Array
    UAC2.0
    AudioStreaming Interface Descriptor:
      bLength               16
      bDescriptorType       36
      bDescriptorSubtype     1 (AS_GENERAL)
      bTerminalLink          2
      bmControls          0x00
      bFormatType            1
      bmFormats         0x00000001
        PCM
      bNrChannels            2
      bmChannelConfig   0x00000000
      iChannelNames         11 Analogue 1
    AudioStreaming Interface Descriptor:
      bLength                6
      bDescriptorType       36
      bDescriptorSubtype     2 (FORMAT_TYPE)
      bFormatType            1 (FORMAT_TYPE_I)
      bSubslotSize           4
      bBitResolution        24
    Endpoint Descriptor:
      bLength                7
      bDescriptorType        5
      bEndpointAddress    0x01  EP 1 OUT
      bmAttributes           5
        Transfer Type            Isochronous
        Synch Type               Asynchronous
        Usage Type               Data
      wMaxPacketSize     0x0038  1x 56 bytes
      bInterval              1
      AudioControl Endpoint Descriptor:
        bLength              8
        bDescriptorType     37
        bDescriptorSubtype     1 (EP_GENERAL)
        bmAttributes        0x00
        bmControls          0x00
        bLockDelayUnits        2 Decoded PCM samples
        wLockDelay             8
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        1
    bAlternateSetting       2
    bNumEndpoints           1
    bInterfaceClass         1 Audio
    bInterfaceSubClass      2 Streaming
    bInterfaceProtocol     32
    iInterface              4 XMOS Microphone Array
    UAC2.0
    AudioStreaming Interface Descriptor:
      bLength               16
      bDescriptorType       36
      bDescriptorSubtype     1 (AS_GENERAL)
      bTerminalLink          2
      bmControls          0x00
      bFormatType            1
      bmFormats         0x00000001
        PCM
      bNrChannels            2
      bmChannelConfig   0x00000000
      iChannelNames         11 Analogue 1
    AudioStreaming Interface Descriptor:
      bLength                6
      bDescriptorType       36
      bDescriptorSubtype     2 (FORMAT_TYPE)
      bFormatType            1 (FORMAT_TYPE_I)
      bSubslotSize           2
      bBitResolution        16
    Endpoint Descriptor:
      bLength                7
```

```
        bDescriptorType         5
        bEndpointAddress     0x01  EP 1 OUT
        bmAttributes            5
          Transfer Type            Isochronous
          Synch Type               Asynchronous
          Usage Type               Data
        wMaxPacketSize     0x001c  1x 28 bytes
        bInterval               1
        AudioControl Endpoint Descriptor:
          bLength                8
          bDescriptorType        37
          bDescriptorSubtype      1 (EP_GENERAL)
          bmAttributes         0x00
          bmControls           0x00
          bLockDelayUnits         2 Decoded PCM samples
          wLockDelay              8
      Interface Descriptor:
        bLength                 9
        bDescriptorType         4
        bInterfaceNumber        2
        bAlternateSetting       0
        bNumEndpoints           0
        bInterfaceClass         1 Audio
        bInterfaceSubClass      2 Streaming
        bInterfaceProtocol     32
        iInterface              5 XMOS Microphone Array
        UAC2.0
      Interface Descriptor:
        bLength                 9
        bDescriptorType         4
        bInterfaceNumber        2
        bAlternateSetting       1
        bNumEndpoints           1
        bInterfaceClass         1 Audio
        bInterfaceSubClass      2 Streaming
        bInterfaceProtocol     32
        iInterface              5 XMOS Microphone Array
        UAC2.0
        AudioStreaming Interface Descriptor:
          bLength               16
          bDescriptorType       36
          bDescriptorSubtype     1 (AS_GENERAL)
          bTerminalLink         22
          bmControls          0x00
          bFormatType            1
          bmFormats         0x00000001
            PCM
          bNrChannels           16
          bmChannelConfig   0x00000000
          iChannelNames         13
        AudioStreaming Interface Descriptor:
          bLength                6
          bDescriptorType       36
          bDescriptorSubtype     2 (FORMAT_TYPE)
          bFormatType            1 (FORMAT_TYPE_I)
          bSubslotSize           4
          bBitResolution        24
        Endpoint Descriptor:
          bLength                7
          bDescriptorType        5
          bEndpointAddress    0x81  EP 1 IN
          bmAttributes           37
            Transfer Type            Isochronous
            Synch Type               Asynchronous
            Usage Type               Implicit feedback Data
          wMaxPacketSize     0x01c0  1x 448 bytes
          bInterval              1
          AudioControl Endpoint Descriptor:
            bLength                8
            bDescriptorType        37
            bDescriptorSubtype      1 (EP_GENERAL)
            bmAttributes         0x00
            bmControls           0x00
            bLockDelayUnits         2 Decoded PCM samples
            wLockDelay              8
        Interface Descriptor:
          bLength                 9
          bDescriptorType         4
          bInterfaceNumber        3
          bAlternateSetting       0
          bNumEndpoints           0
          bInterfaceClass       254 Application Specific
          Interface
          bInterfaceSubClass      1 Device Firmware Update
          bInterfaceProtocol      1
          iInterface             10 XMOS DFU
          Device Firmware Upgrade Interface Descriptor:
            bLength                             9
            bDescriptorType                    33
            bmAttributes                        7
              Will Not Detach
              Manifestation Tolerant
              Upload Supported
              Download Supported
            wDetachTimeout                    250 milliseconds
            wTransferSize                      64 bytes
            bcdDFUVersion                    1.10
Device Qualifier (for other device speed):
  bLength                10
  bDescriptorType         6
  bcdUSB               2.00
  bDeviceClass          239 Miscellaneous Device
  bDeviceSubClass         2 ?
  bDeviceProtocol         1 Interface Association
  bMaxPacketSize0        64
  bNumConfigurations      2
Device Status:     0x0000
  (Bus Powered)
```

Bibliography

[1] **xcore array microphone — xmos**. 2018. URL https://www.xmos.com/cn/support/boards?product=20258.

[2] **AN4426: Tutorial for MEMS microphones**. STMicroelectronics, 2017. DocID025704 Rev 2.

[3] Park S: **Principles of sigma-delta modulation for analog-to-digital converters** 1999.

[4] **Universal Serial Bus Specification**. Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V., 2000. Revision 2.0.

[5] **Universal Serial Bus Device Class Definition for Audio Devices**. USB Implementers Forum, Inc., 2006. Release 2.0.

[6] **Universal Serial Bus Device Class Definition for Audio Data Formats**. USB Implementers Forum, Inc., 2006. Release 2.0.

[7] **USB Engineering Change Notice: Interface Association Descriptor**. USB Implementers Forum, Inc., 2003.

[8] **USB Interface Association Descriptor Device Class Code and Use Model**. USB Implementers Forum, Inc., 2003. Revision 1.0.

[9] **XUF216-512-TQ128 — XVSM-2000-TQ128 Datasheet**. XMOS, 2018. Document Number: X006990.

[10] **xCORE: Architecture Overview**. XMOS, 2013. Version 1.2.

[11] **XC Specification**. XMOS, 2011. Document Number: X5965A.

[12] **XMOS Programming Guide**. XMOS, 2014. Document Number: XM004440A.

[13] **Eclipse foundation**. 2018. URL https://www.eclipse.org/.

[14] **xTIMEcomposer User Guide**. XMOS, 2015. Document Number: XM009801A.

[15] **xTAG v3.0 Hardware Manual**. XMOS, 2015. Document Number: XM006125A.

[16] **Low-Power, Stereo Digital-to-Analog Converter**. Cirrus Logic, 2013. Rev. F1.

[17] **Synchronous rectification with inhibit, 1.5 A, 1.5 MHz fixed or adjustable, step-down switching regulator**. STMicroelectronics, 2009. Doc ID 12236 Rev 9.

[18] **200-mA Low-I Q Low-Dropout Regulator for Portable Devices**. Texas Instruments, 2016. SLVSA61H – FEBRUARY 2010 – REVISED AUGUST 2016.

[19] **Low Power Voltage Detector**. STMicroelectronics, 2006. Rev 4.

[20] **Simple Sequencers® in 6-Lead SC70**. Analog Devices, 2014. Rev. B.

[21] **TinyLogic® UHS Dual Buffer (Open-Drain Outputs)**. Fairchild Semiconductor, 2013. Rev. 1.0.7.

[22] **3.3V HCMOS / TTL COMPATIBLE SMD CRYSTAL CLOCK OSCILLATOR**. Abracon Corporation, 2011. Revised: 04.13.11.

[23] **Fractional-N Clock Multiplier**. Cirrus Logic, 2008. Doc ID: DS840PP1.

[24] **TinyLogic® UHS Triple Buffer**. Fairchild Semiconductor, 2001. Doc ID DS500494.

[25] **Quadruple Bus Buffer Gate With 3-State Outputs**. Texas Instruments, 2015. Doc ID SCAS290Q.

[26] **Usb audio class 2.0 software — xmos**. 2018. URL `http://www.xmos.com/support/software/uac2`.

[27] **Microphone array library**. XMOS, 2017. Document Number: XM010267.

[28] Thierry H, Stanford-Jason A and Tiziano L: **Real-time digital signal processing on the xcore-200 architecture** 2016. Document Number: X010269.