Technische Universität Hamburg-Harburg

Master Thesis

Low-Latency Audio over IP on Embedded Systems

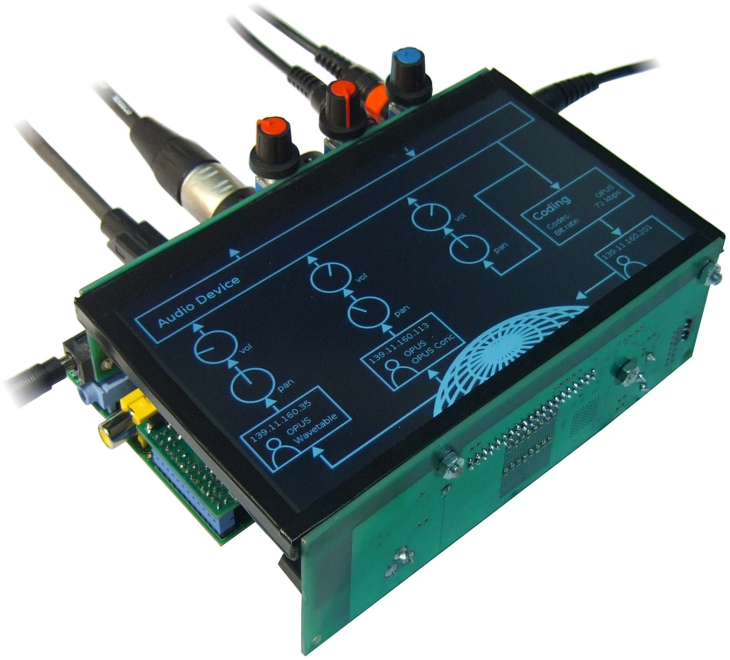
by

Florian Meier

Start date:	02. April 2013
End date:	02. October 2013
Supervisor:	Dipl.-Ing. Marco Fink
Supervising Professor:	Prof. Dr.-Ing. habil. Udo Zölzer
Second Examiner:	Prof. Dr. rer. nat. Volker Turau

Abstract

Transmission of audio data over networks, such as the Internet, is a widespread technology. Up to now, the primary application is voice transmission (Voice over IP). Although modern Voice over IP systems are designed to reduce the audio latency, it is still too high for the bidirectional transmission of live music. The construction of a low-latency music environment would enable distributed musical performances and "Jamming over IP". This thesis takes a step in this direction by building an Audio over IP system as an embedded system. All components needed for a jamming session over the Internet are integrated in a handy box on the basis of a Raspberry Pi. This required the development of a Linux kernel driver. Furthermore, a software for low-latency audio transmission is build and evaluated, considering audio quality, data rate, and the reduced computational power of the embedded system.



Declaration by Candidate

I, FLORIAN MEIER (student of Informatik-Ingenieurwesen at Hamburg University of Technology, matriculation number 20836390), hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been acknowledged.

Hamburg, 02. October 2013

Florian Meier

Table of Contents

List of Figures

List of Tables

List of Sourcecodes

List of Symbols

Symbol	Description	Page List
f_s	Sampling frequency	9, 10, 15, 19, 30, 41, 66, 67
N_F	Frames per audio data block	9, 10, 15, 19, 21, 22, 25, 50–52, 66, 67
$(s_n, s_{n+1}, \ldots)$	Sequence of packet sending times	9, 66
$(r_n, r_{n+1}, \ldots)$	Sequence of packet reception times	10, 21, 66
$(a_n, a_{n+1}, \ldots)$	Sequence of data transmission times to the hardware	21
δ_{block}	Blocking delay	15
δ_n	Internet delay of the n-th packet	9, 10
δ_{min}	Minimum Internet delay	9, 19, 66, 67
δ_{max}	Maximum Internet delay	9, 19, 66, 67
$\overline{Q}$	Average receiver queue length	20
σ_Q	Standard deviation of receiver queue length	20
Q_{target}	Target receiver queue length	20
β	Receiver queue length tuning factor	20, 50–52
Δt_C	Time needed for concealment	21
Δt_M	Time needed for mixing	21
Δt_S	Timing safety margin	21
Δt_P	Total time needed for processing	21

Chapter 1

Introduction

Generation and reception of sounds builds the basis of human interaction. Therefore, transmission of audio is of substantial importance for communication. Telephony builds up a feeling of proximity that is not possible with other forms of communication such as E-Mail or SMS. Human interaction is not limited to the exchange of messages. Making music together unites people. This is only possible for people within the same acoustical space. Current telephone systems can not fulfill the demands in terms of audio quality and latency, making long-range music sessions less pleasurable.

Despite its long history, the audio transmission technology is continuously developing. A major part of this development is attributed to the Internet. There are a lot of Internet-based telephony applications like Skype or Google Talk and even the long-established telephone system is using more and more Internet technologies. For transmitting audio data over the Internet, the continous audio stream is split into chunks of data. These packets have to find their own way through the Internet, competing with packets from all the other Internet applications.

Using the Internet Protocol has several advantages compared to the former circuit-switching technique. First of all, the system utilization is much better because the medium is shared between the different users and their applications. Although, it has a major disadvantage, caused by the shared and best-effort nature of the Internet: The paket delay is not deterministic. This leads to problems for resassembling the packets into a continous audio stream again.

It is commonly counteracted by using large buffers that lead to an artificial delay of the packets. Furthermore, data compression targets at reducing the data rate. Commonly used audio compression algorithms such as mp3 have a high inherent algorithmic delay. Both procedures induce high overall latency of audio.

High latency is a moderate problem for standard telephony, but for some applications it is very important to have exact timing of the audio. This includes remote guidance systems for medical or maintenance environments and networked music performances as mentioned above.

The purpose of this master thesis is the development of an autonomous device for networked music performances. It is ready for plugging in a guitar and a network cable and start jamming

over the Internet. The core of this embedded system is a Raspberry Pi, a credit-card sized single board computer running Linux.

Many hardware and software components complement each other in this system. This includes circuit boards for the audio hardware, for the attachment of musical instruments, and for driving a touchscreen that is used for user interaction. These are developed during this thesis together with the software parts such as a Linux kernel driver for interfacing the audio hardware and the application for networked music performances itself.

For this software, the consideration of latency is of particular importance. Therefore, it dynamically adapts to the network conditions and reduces the time of buffering. In the case of a delayed or missing packet, the technique of packet loss concealment is used to calculate the missing data from the previous packets.

The thesis is structured as follows. The current state of research concerning the influcence of latency on musical interaction is presented in Chap. 2 together with the presentation of existing systems. In Chap. 3, fundamentals of Audio over IP and audio on embedded systems are covered. Chap. 4 presents the Audio over IP software and the underlying algorithms, whereas Chap. 5 describes the developed kernel driver that interfaces the audio hardware. This hardware is exposed in Chap. 6 and evaluated in Chap. 7 together with the software. The thesis is concluded in Chap. 8.

Chapter 2

State of the Art

The general idea of networked music performances is to play music together while the audio data is transferred via the Internet. By using this technique it is no longer necessary for the musicians to be at the same place. This enables rehersal or practice sessions of bands that got scattered over the continent as well as it provides a basis to form new communities of musicians [1]. It can also be used for remote teaching of music and composing [2].

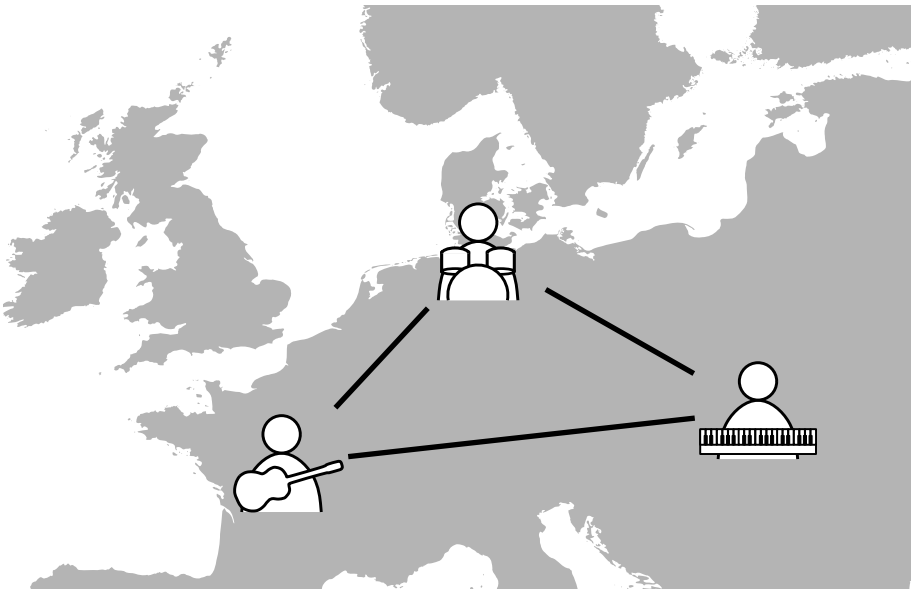


Figure 2.1: *General idea of Networked Music Performances*

The transfer of a session from a rehersal room to the Internet implies many technical issues. The most severe is the increased latency. In a classical environment, the time between one player plays a note and another one hears it is determined by the distance and the speed of sound in air. If the sound is transferred via wire, for example in recording studios, the transmission delay is determined by the speed of light and is negligible for classical environments. This is no longer true for transmission via the Internet. Most of all because of the much larger distance and the additional processing that takes place. This has significant consequences for the musical interaction.

2.1 Effect of Latency for Musical Interaction

The effect of latency for musical interaction and the maximum tolerable latency is widely investigated. In [3], Chafe et al. have conducted the following simple experiment: Two persons in two seperate rooms were given a simple (but different) rhythm that they should reproduce by clapping their hands. The own sound can be heared directly, while the sound from the other person is delayed by a variable amount of time. This was evaluated by measuring the onsets by an automated procedure. The overall result was that the larger the delay gets, the more the persons tend to deccelerate. This is because they hear the sound of the other person later than it is expected, so they think in order to be synchronous they have to slow down.

It is interesting to note, that there were several regions with the same amount of decceleration that can be more or less clearly seperated: Up to 8 ms the clappers even accelerate. This is explained by an inherent tendency to accelerate. Up to 25 ms the tempo is more or less stable. With more latency they were also able to maintain the clapping, but with increasing difficulty. It was no longer possible with more than 60 ms delay. The authors assume that with 25 ms the environment is more or less natural, as this is also the time for sound to travel a distance of about 8 m.

With greater latency, the persons intentionally or unintentionally used strategies to overcome the latency: This could for example pushing the beat in order to compensate for the decceleration or ignoring the sound of the other. This produces some kind of leader/follower pattern as it also can be found for example with a drummer and a guitar player.

A similar, but more complex experiment was conducted by Chew et al. in the context of their Distributed Immersive Performance Project [4] where they try to create a virtual space where the participants can communicate and play music together as if they are at the same place. In the experiment, two professional piano players play Poulenc's Sonata for Piano Four-Hands, while the MIDI signals from the keyboard where artificially delayed. It was discovered, that they can play together with a delay of about 50 ms. They were conscious of the delay and it was confusing, but they were able to compensate it with some training. However, the players didn't feel comfortable with their own play, because it sounds unsynchronous although it wouldn't be for an external audience. This led to a new technique where the sound of the own instrument was also delayed. This rose the acceptable delay up to 65 ms, but more importantly the players now could enjoy their play, because the heared music as such is synchronous. As Carôt et al. state, it is very difficult to determine a general valid threshold for the maximal acceptable latency [5]. They have evaluated the interplay of drummers and a bass player with various delays. The acceptable delay was very different for different drummers, but in general less latency is accepted the faster the rhythm is. The concrete values range from 60 ms down to 5 ms.

2.2 Musical Interaction with High Latency

Although, a high latency it is not considered as natural, it doesn't prevent musical interaction, but creates a new form of music. Cáceres and Renaud state "In this specific case, the network is considered as medium which bring its own acoustic contribution to the music making process rather than being a link between two spaces." [6]. This has multiple facets: The delay can be

used as an effect generator for echo or reverberation, "making the distance "audible"". Secondly, the different places and their acoustics can be used as musical element. Furthermore, the delay itself can be used to generate music that sounds different, but nevertheless synchronous at different places. One possibilty for this is referred to as feedback locking: The own signal (or an artificial metronome pulse) is feed into the network and back. After the musician locked his own tempo to the feed back signal, the beat is equal to the round trip time (RTT). When the musician at the other end plays synchronous to this signal, the sound of both musicians is shifted by one beat at the origin. For some music this is acceptable and creates new interesting musical impressions.

An impressive demonstation of this effect was the Pacific Rim of Wire [7]. The Stanford New Ensemble and the Stanford Laptop Orchestra (SLOrk) performed a concert at Stanford together with an ensemble of traditional Chinese instruments located in Beijing. They performed Terry Rilley's In C. It is written as a chain of of various patterns where to musicians are free to switch from one to the next. Therefore, each play of the piece sounds different. The tempo of the play was exactly matched to the network delay between Stanford and Beijing (110 ms) and was supported by the SLOrk, whose players could choose from several patterns matched to a common metronome. The players at Beijing played synchronous to this sound while being shifted by an eighth note at Stanford. This creates a "loose-but-synchronous" music as intended by the composer.

2.3 Existing Approaches

One of the first research groups in the topic of networked music performances is the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University [8]. They started with an application called SoundWIRE that can be used to make network quality audible, by transmitting sonar like pings. This lead to the development of a generic stream engine for transmitting audio over the Internet. It does not utilizes any data compression and can therefore only used over connections like the Internet2 backbone with high network bandwidth. It uses TCP in order to guarantee that all packets are delivered. Quality of service (QoS) is used for increasing the quality of the connection. It has a multi-threaded plugin architecture with various input, output and processing plugins. The total latency with this engine is about 200 ms.

In order to reduce the latency, a new engine called JackTrip was implemented by Juan-Pablo Cáceres and Chris Chafe at the CCRMA [9]. It also does not use any data compression, but uses UDP as underlying protocol eliminating the time-intesive retransmission that is inherent to TCP. The danger of packet loss is counteracted by overlapping the packets. This way, a missing packet can be reconstructed by the surrounding packets. The underlying audio engine is the JACK sound system as implied by the name. JackTrip has several threads for audio and networking that are connected through ring buffers with special over- and underrun handling. For example, when data is missing and can not be reconstructed, the system plays back silence or the last received packet. JackTrip was successfully used in several networked music performances like the Pacific Rim of Wire presented in the previous section.

While audio data has a high data rate, the musical information itself (tone pitch, length etc.) can be transmitted with very few data as musical instrument digital interface (MIDI). This is the foundation of Quintet.net [10] by the Hamburg Institute for Microtonal Music and Electronic

Media. The MIDI data can be directly created by appropriate instruments (keyboards etc.), but also classical instruments can work with this system by using a pitch tracker. All data is transmitted to a central server that is controlled by a conductor. The latency is about 200 ms to 1000 ms.

Yet another project is DIAMOUSES by the Technological Educational Institute of Crete [11]. It has a very versatile platform that can be used for many different use cases. Starting with collaborative music rehersal to over concert broadcasts to remote music master classes. A central server can be used for reduced network bandwidth usage, but a peer-to-peer (P2P) connection is supported, too. It can be used with musical instrument digital interface (MIDI) and audio data, has a dedicated portal for arranging jamming sessions and provides audience involvement via digital video broadcast (DVB).

The notable property of Soundjack [12] developed by Alexander Carôt et al. is the possible usage via narrow band networks like digital subscriber line (DSL) connections. It uses a prediction-based, lossy audio codec with low delay (5.3 ms at 48 kHz) and provides error concealment for compensating packet loss. In a jam session between Lübeck and Paris they faced a latency of 33.3 ms including the network delay of 20 ms.

Chapter 3

Fundamentals

This chapter describes the fundamentals that build the basis of the successive considerations. Starting with a comparison of a conventional music session with networked approaches, the characteristic properties of networked music performances and audio on embedded systems are presented.

3.1 Signal Paths

In a conventional environment as depicted in Fig. 3.1, the sound will be recorded and sampled by means of an analog to digital converter (ADC). It is mixed with the data coming from the other sources of sound and converted back to audible sound by a digital to analog converter (DAC). This would also be possible without transferring the data into digital domain, but for the networked approach this is no longer true, because the network can only transfer digital signals.

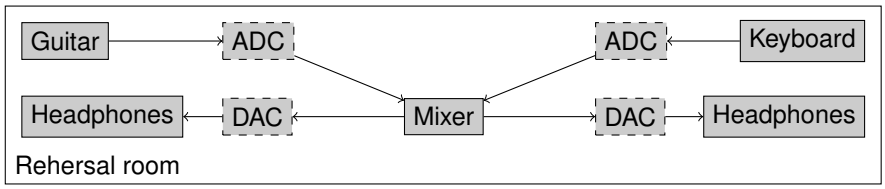


Figure 3.1: *Signal path of conventional environment*

There are two different approaches for network based music distribution [11]. Fig. 3.2 shows a centralized network based approach. All ADC send their sampled data to a server via the network where it is mixed and send back to the DAC. The peer-to-peer (P2P) approach in Fig. 3.3 waives the need for a central server. The sampled data is send to the other parties where it is mixed by a local mixing module. This introduces the need for mixing at every single place, but an individual mix is quite common anyway, even in classical sessions. For example the bass player would like to hear more drums, while the singer would like to hear more guitar and especially himself. The choice of the appropriate approach has to be made individually for each application. The central approach introduces additional latency in the signal path, because the data is not transferred directly, but has to make a detour to the server. On the

other hand, the P2P approach has a higher demand for network bandwidth for more than two participants, because a data packet has to be send multiple times, once for each participant.

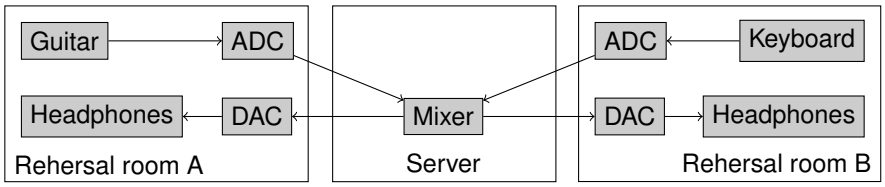


Figure 3.2: *Signal path of centralized network based approach*

While the interface for the musicians is the same in the classical approach compared to the networked approach, the one thing that is really different is the medium of data transmission between the converters and the mixer. While it is a relatively short direct cable connection in the first case (e.g. via AES3) it is the Internet for the latter case. This mainly affects two properties: Latency and data rate. The first describes the time between transmission and reception of the signal. It is important for the relative timing of the musicians. The latter describes the fact that the Internet can handle only a certain amount of data per time. Therefore, a data compressing coding algorithm is advised in order to reduce the required network bandwidth.

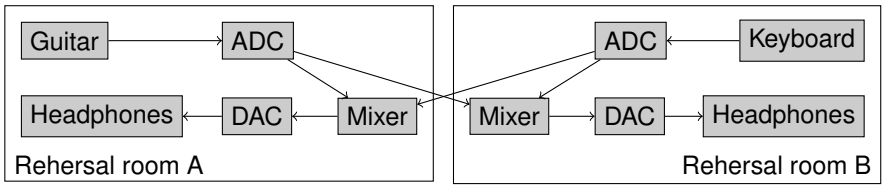


Figure 3.3: *Signal path of peer to peer approach*

3.2 Network Latency

Since the network itself has vast influence on the perceived end-to-end latency, it is advised to invesigate the sources and properties of network latency. In the case of a direct connection via optical fiber, the length of the fiber l_{fiber} and the speed of light c determine the propagation delay

$$\delta_{propagation} \approx \frac{l_{fiber}}{0.65 \cdot c}. \tag{3.1}$$

Even this calculation results in latencies that influence the musical interaction as presented Sect. 2.1. Table 3.1 presents various distances and the corresponding speed-of-light delay. The transmission delays over the Internet were measured by sending an ICMP echo request packet to the homepages of Universität zu Lübeck, Technische Universität München, Université d'Aix et de Marseille and Columbia University, assuming the web servers are located in the respective cities. The round trip delay got by the ping utility was divided by 2 to get the one way delay.

For transmission over the Internet, the latency is even bigger caused by several effects that are inherent to the Internet [13]. First of all, the Internet is not a fully connected network, but relies on central backbones that increase the distance. Secondly, it takes some time to put a

data packet on the line, depending on the packet size and the network bandwidth. Thirdly, the data is not transferred directly, but is passed on via multiple routers where the data is received, analyzed and send to the next router taking some additional time. Furthermore, it is not guaranteed that it can be processed immediately because there might be other data being processed by the router. This introduces even more delay.

	Distance	$\delta_{propagation}$	$\delta_{Internet}$
Rehersal room	6 m	20 ns	-
Large Stage	25 m	83 ns	-
Hamburg - Lübeck	60 km	0.31 ms	2 ms
Hamburg - Munich	600 km	3.08 ms	9 ms
Hamburg - Marseille	1200 km	6.16 ms	14 ms
Hamburg - New York	6000 km	30.8 ms	50 ms

Table 3.1: *Comparison of some approximate line-of-sight distances, corresponding propagation delays, and approximative transmission delays over the Internet*

This gets even worse, because the delay is neither constant nor properly predictable since the exact way of a single data packet as well as the current load of the routers is not predefined. This induces jitter on the received data stream. A mathematical model of the Internet jitter is difficult to obtain [14]. Corlett et al. [15] proposed a shifted gamma function for approximating the probability density function of Internet delay, but that is not universally accepted, too [16].

For this consideration the following simplification on the delay of a packet δ_n is made: There is a minimal delay δ_{min} that is given by the delay on the best path under best conditions. No packet will arrive before this delay. Furthermore, there is a maximal delay δ_{max} . A packet not arriving before this time is considered as lost and has to be handled seperately. This might happen, because an intermediate router has dropped the packet or it just has taken a very long and crowded router.

Since the data transmission over the Internet is packet based, the steady stream of audio data recorded at a sampling rate of f_s has to be splitted into data blocks of size N_F . This data is then optionally coded and send to the Internet. The sequence of the packet sending times $(s_n, s_{n+1}, \ldots)$ complies to a regular pattern described by

$$s_{n+1} = s_n + \frac{N_F}{f_s}. \tag{3.2}$$

This is no longer true for the receiver side, because of the non-deterministic behaviour of the Internet. Under the assumption that the packet send at s_n is not lost, it arrives at

$$\begin{gathered} s_n + \delta_n. \\ \delta_{min} \leq \delta \leq \delta_{max} \end{gathered} \tag{3.3}$$

The sequence of packet reception times $(r_n, r_{n+1}, \ldots)$ is no longer regular. The time intervall between two packet receptions is

$$\begin{aligned} r_{n+1} - r_n &= s_{n+1} + \delta_{n+1} - (s_n + \delta_n) \\ &= \frac{N_F}{f_s} + \delta_{n+1} - \delta_n. \end{aligned} \tag{3.4}$$

The difference between the packet delay $(\delta_{n+1} - \delta_n)$ of two packets is called IP Packet Delay Variation (IPDV) according to [17]. Since δ_n might be different from δ_{n+1}, the regular pattern is destroyed.

3.3 Audio Coding

A raw, stereo, 16 bit, 44.1 kHz audio signal requires a data rate of

$$2 \cdot 16 \,\frac{\text{bit}}{\text{sample}} \cdot 44\,100 \,\frac{\text{sample}}{\text{s}} \approx 1411 \,\frac{\text{kbit}}{\text{s}}. \tag{3.5}$$

In addition, some more network bandwidth is needed for control data (i.e. headers). For comparison: An average digital subscriber line (DSL) connection, as used in many homes, has an upload network bandwidth of 1024 $\frac{\text{kbit}}{\text{s}}$.

This motivates the use of a coding procedure that reduces the required bit rate. It is necessary to distinguish between lossless and lossy coding algorithms. The first exploits the inherent redundancy of the audio signal to reduce the amount of data needed, while using a representation that allows perfect reconstruction of the original signal.

A popular example of a lossless coding algorithm is FLAC. It models the input signal (e.g. by using a linear predictive coding) and encodes the difference to the original signal by means of an entropy encoding. Data compression of about $50\,\% - 60\,\%$ can be reached [18]. For the above example, this would yield a bit rate of about 706 $\frac{\text{kbit}}{\text{s}}$.

The lossy coding on the other hand utilizes the characteristics of the human ear in order to create a representation that sounds similar to the human ear, but uses less data. An example is the fact that the human ear has a frequency dependent loudness threshold under which a sound can not be recognized. Lossy coding algorithms can reach much higher compression ratios. For example HE-AAC was rated excellent in listening tests [19] at bit rates as low as 48 $\frac{\text{kbit}}{\text{s}}$.

A coding method that is tailored to suit the needs of networked music performances is the CELT codec [20]. It is merged with the SILK codec (optimized for Voice over IP (VoIP)) into Opus, a coding algorithm that was standardized in 2012 [21]. Just like many other audio coding algorithms (for example mp3 or Ogg Vorbis), Opus/CELT is block based and uses a modified discrete cosine transform (MDCT). Since MDCT works on windows of data, enough data has to be collected before the transformation can take place.

This coding delay is an important property for this application, because the first sample of a window can not be transmitted before the last sample of a window was recorded. Therefore,

unlike most other codecs, Opus/CELT uses a comparably small window (down to 2.5 ms compared to 50 ms or more). The disadvantage of the small window is the decreased frequency resolution. Special countermeasures are implemented in Opus/CELT to minimize the audible impact of this.

The Opus codec also implements an error concealment that can be used if a data packet is lost, for example because of network congestion. When a packet is lost, the decoder searches for a periodicity of the signal decoded so far and repeats it [21]. The error concealment technique presented in [22] uses an auto-regressive model of the previous data for extrapolating new data. The survey of multiple algorithms in this paper yields that the best algorithm provides good sound quality for a packet loss rate of 2 %. Although, the audio quality considerably depends on the type of music itself.

3.4 Audio on Embedded Systems

> "An embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a specific function." [23]

In the context of this thesis, an embedded system consists of a microprocessor and a bunch of peripherals like audio hardware that is used for a dedicated task. The dedicated task of the developed system is recoding, playback and Internet transmission of music. Often, as in the case of the Raspberry Pi, many peripherals are already integrated into a single integrated circuit (IC). This is referred to as system on chip (SoC). Examples for other embedded systems include digital cameras, anti-lock brakes, dish washers or mobile phones. Especially the last example shows that sound capabilities are quite common for embedded systems.

When implementing sound support for a SoC based embedded system, it might be the case that the SoC already has a dedicated analog sound peripheral. Otherwise, a digital audio interface might be available were a specific integrated audio circuit, called audio codec, can be connected. A fact that emphasizes the relevance of mobile phones for the embedded market is that a major part of the existing audio codecs are especially designed for mobile phones.

On the software side there are different possibilities: The software can run directly on the SoC (also referred to as "bare-metal-coding") or a operating system is used that provides a lot of services for interfacing the hardware and makes the development a lot easier for complex systems. An operating system that is widely used for embedded systems is Linux. It is open-source, so it is easy to look at the source code and modify it in order to optimize the operating system for the needs of the platform. The Linux kernel includes a lot of drivers that can be used for all kinds of peripherals. In the case of a missing driver for a specific peripheral, a dedicated driver can be written. This task is simplified by several existing services and frameworks.

The following sections describe the hardware and software issues when connecting an external audio codec to a SoC running Linux.

3.4.1 Audio Codecs

An audio codec is a special IC dedicated for adding analog audio support to other electrical devices like SoC. It is not to be confused with audio codec as a synonym for a data compressing coding method for audio. It combines an ADC and an DAC. The first transfers analog audio signals (like those captured with microphones) to digital signals that can be processed by digital circuits (like a microprocessors or a digital signal processor (DSP)). The latter transfers the audio back to an audible analog signal. Many audio codecs feature additional components like amplifiers, filters or signal multiplexers.

3.4.2 Digital Interfaces

There are many interface standards for connecting electrical devices. The ones that are important for this thesis are presented in the following sections. A first overview is given by Table 3.2.

	Payload	Typical Data Rate	# Devices	Direction
I^2C	Arbitrary (e.g. control data)	100 $\frac{kbit}{s}$	Up to 1136	bidirectional
I^2S	Audio	1.536 $\frac{Mbit}{s}$	2	bidirectional
HDMI	Video and Audio	3.96 $\frac{Gbit}{s}$	2	mainly unidirectional

Table 3.2: *Comparison of some digital interfaces used in multimedia devices*

3.4.2.1 I^2C

Inter integrated circuit (I^2C) is a general purpose digital bus for transmitting data such as control data between various IC [24]. There are one or multiple masters that can start a transmission and one or multiple slaves that receive a transmission. A transmission can be a read or write operation, thus it is also possible to transmit data from a slave to a master. It uses two data lines. SCK is a clock signal that is generated by a master. SDA is used for transmitting data. It is possible to connect multiple IC to the same bus. Each IC gets an address that is used by to select the IC for the current data transmission.

The lines are connected to open collector outputs and have pull-up resistors to the positive supply making the default state positive. This makes it possible for each IC connected to the bus, to pull the signal down as long as no other IC pulls the line down at the same time. This creates a wired-AND connection, as the state of the signal is the AND function of the individual states "I don't access the line".

Fig. 3.4 shows a I^2C write transfer. It can be seen that the clock has twice the frequency than the data signal and is shifted. This is because of the specification that the data has to be stable while the clock is high and might change while the clock is low. The only exception to this is

the start and stop condition: For indicating the start of a transmission, the data line goes down while the clock stays high. For stopping the transmission, the data line goes up while the clock stays high. It is also possible to initiate a new start signal to indicate a further transmission without an intermediate stop signal.

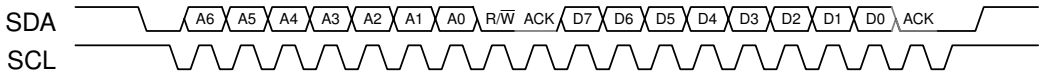


Figure 3.4: *Data format of I^2C write transfer*

After sending a start signal, the write transmission proceeds with the 7-bit address of the slave in question. It follows one low bit to indicate a write transmission. These 8 bits are acknowledged by the slave. The gray line indicates an access of the slave. After that one byte of data follows that is acknowledged, too. The transmission is ended by the stop signal. It is also possible to send multiple data bytes and I^2C also provides a 10 bit adressing mode not shown here.

Figure 3.5: *Data format of I^2C read transfer*

Fig. 3.5 shows the transmission of data from a slave to a master. The first difference is that the $R/\overline{W}$ bit is now high to indicate a read. After the ACK by the slave, a byte of data is send by the slave. This is then acknowledged by the master that thereupon closes the connection.

3.4.2.2 I^2S

Integrated interchip sound (I^2S) transmits digital stereo audio data between different integrated circuits (ICs) within the same device [25]. The data is transmitted in a serial format as depicted in Figure 3.6.

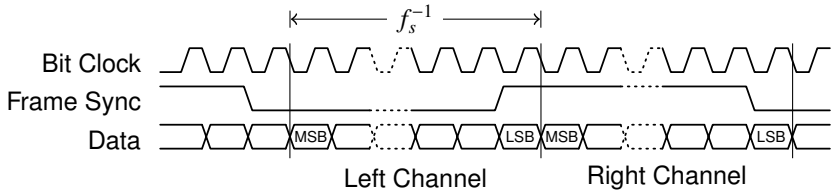


Figure 3.6: *Data format of I^2S*

The single bits of an audio sample are transmitted one after another. The transition from one bit to another is controlled by a dedicated bit clock line. The samples are terminated by the transition of a frame sync clock. The different channel of the stereo signal are transmitted within a single period of the frame sync clock. One during high level of the frame sync and the other during the low level of the signal. Together with the data line, I^2S uses three lines for an unidirectional connection. For a bidirectional connection it is possible to use two triples of lines or to share the clock signals resulting in a minium of four lines.

Although, it is sometimes called I²S bus, it only connects two devices. One is responsible for generating the clocks (referred to as master), while the other (the slave) has to adapt itself to these signals. The frequency of the frame sync signal is exactly the sampling frequency (for example 48 kHz). Since two samples have to fit into a single period, the bit clock depends on the sampling frequency and the bit length of the samples. For 16 bit samples, this leads to a frequency of $2 \cdot 16 \cdot 48\,\text{kHz} = 1.536\,\text{MHz}$. Many audio codecs support other frequencies as long as there are enough bit clock transitions for transmitting the required number of bits.

3.4.2.3 HDMI

The high definition multimedia interface (HDMI) standard is widely used in today's consumer multimedia devices, such as TV sets and receivers. It transmits video and audio data via a bunch of differential digital lines. It also contains an I²C bus for transmitting information about the connected device (display resolution etc.), a wire for hot-plug detection and a consumer electronics control (CEC) interface for transmitting control information (for example volume or channel selection).

3.4.3 Direct Memory Access

Once the data arrived via a digital audio connection, it has to be transferred into the random access memory (RAM) of the computer where it can be processed by the software. Because of the large amount of data (cf. 3.5) and the narrow timing requirements, it is favorable to do this fast and without stressing the central processing unit (CPU) too much. As this is a common requirement for many input and output devices, a technique called direct memory access (DMA) was established. As the name implies, this allows peripherals to directly transfer data from and to the RAM without making use of the CPU. The CPU is only used for setting up the transfer.

During the evolution of the computer, many different modes were arranged. The transfer might be triggered by the software or by the hardware and there exist many different kinds of adressing considering the different data buses (for example ISA or PCI used in personal computers) [26].

For embedded systems and especially for SoC, the most important data bus is the Advanced Microcontroller Bus Architecture (AMBA) [27]. It was invented for the ARM architecture and has the concept of bus masters and bus slaves. Bus masters can directly write to and read from bus slaves. This allows DMA of a peripheral to the RAM as long as it was implemented as bus master. Some peripherals might not be implemented as bus masters. They need the assistance of a dedicated DMA controller that conducts the data transfer for them.

3.4.4 Audio Software

The audio software running on the CPU is responsible for fetching the sampled audio data from the hardware, process it and play audio back to the hardware [28]. It typically consists of (at least) three layers as depicted in Fig. 3.7. The lowest consists of the driver for accessing the

sound card (or another peripheral responsible for audio, such as an I^2S interface). In Linux most audio drivers are combined in the ALSA subsystem inside the kernel.

The ALSA subsystem provides a consistent interface that can be accessed by a library that is part of the middle layer. The library itself can be accessed by an application of the upper layer. The middle layer is not absolutely necessary, but it provides a much more powerful application programming interface (API) that simplifies the development of applications. It is also possible that there is more than one library stacked in the library layer. Each library provides some more functionality, but it might also induce more computational effort and complexity. For Linux these might be Portaudio or Jack. Both provide additional services, such as mixing the signal from different applications which is not possible with plain ALSA.

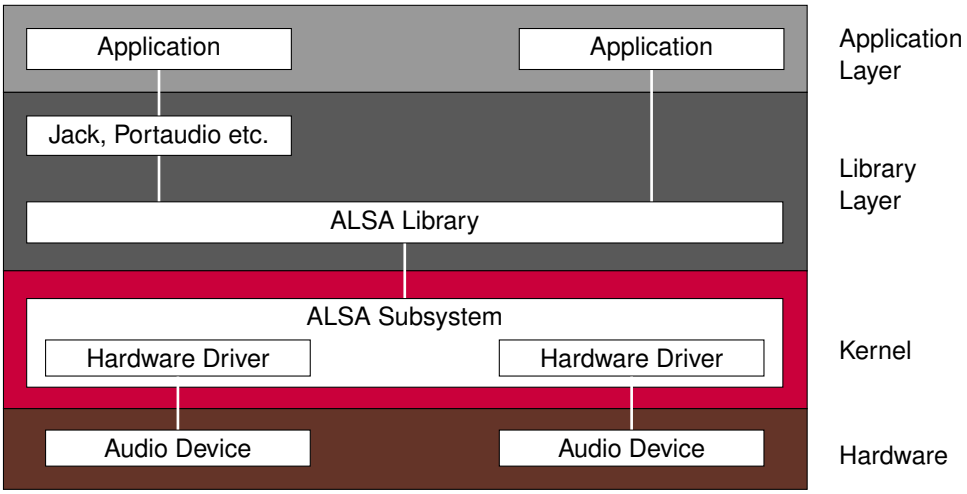


Figure 3.7: *The components involved in providing audio for a Linux based system*

The application has to access the audio library on a regular basis. Either to transmit new audio data in a playback application or to collect recorded audio data in a capturing application. Audio data is transferred in blocks of frames referred to as periods in ALSA. Each frame contains one sample per channel (i.e. two samples per frame for stereo). The size of a period N_F has direct influence on the latency. This is an important property, especially for networked music performances as presented in Section 3.2. The first sample of a period can only be transferred, after the last sample of the period arrived. This leads to a blocking delay of

$$\delta_{block} = \frac{N_F}{f_s}. \tag{3.6}$$

N_F	f_s = 44.1 kHz	f_s = 48 kHz
64	1.5 ms	1.3 ms
128	2.9 ms	2.7 ms
240	5.4 ms	5.0 ms
1024	23.2 ms	21.3 ms
8192	185.8 ms	170.7 ms

Table 3.3: *Blocking delay for different period sizes and sampling frequencies*

Table 3.3 lists some common period sizes, sampling frequencies, and the corresponding blocking delay. This latency increases lineary, if more than one block is buffered at a time, because the application transmits further periods before the playback of the previous was finished or does not collect the recorded periods in time. While smaller blocks and less buffered blocks decrease the latency, it increases the timing requirements for the application. This is because the period between the time when the application was signaled that more data is needed and the time when the audio driver runs out of playable audio data is smaller.

This is a real-time requirement that needs particular attention, especially for low-latency applications on systems with little processing power such as embedded systems. As stated in [28], it is at least advisable, to increase the priority of the audio thread and use a real time scheduler policy such as SCHED_FIFO. The current Linux standard scheduler is the Completely Fair Scheduler (CFS). As the name implies, the main goal is to enable a balance between the the various processes in order to increase the overall performance (in CPU utilization as well as in reactiveness of interactive applications). Although, a audio thread should not be fair in the sense that it should do the processing as fast as possible and at the desired point in time in order to achieve a minimal latency. This will clearly decrease the performance of other processes, but since the audio application is the determining application for the embedded system this is accepted.

3.4.5 Advanced Linux Sound Architecture

Advanced Linux Sound Architecture (ALSA) is the most common framework for handling audio in Linux. It started in the middle of the 1980s as a driver for the Gravis UltraSound cards written by Jaroslav Kysela [29]. Since then, it was expanded to a generic audio driver framework and got included into the official Linux 2.5 kernel in 2002 [30]. Besides audio drivers, Advanced Linux Sound Architecture (ALSA) provides interfaces and services that encapsulate common functionality for audio drivers, for example buffer handling. This leads to a considerable reduction in development effort for writing and maintaining an audio driver.

ALSA is splitted into a library that is accessed by the applications (alsa-lib) and a framework that is part of the Linux kernel. The library provides an API for developing an application with audio support. The full documentation can be accessed at [31], but for getting a small insight, Sourcecode 3.1 represents an application that outputs a sawtooth wave to the loudspeaker. It is only intended for illustration, especially since it does not contain any error handling.

The structure of the code is as follows: Line 10-12 provide the initialization. This includes for example data format, sampling frequency and data transfer mode. There are several modes for accessing the hardware. The one used in this example is the memory mapped mode (MMAP). The data is directly written into the buffer provided by ALSA without the need for an additional buffer. This has the advantage that the data can be generated just in time. Therefore, the latency is reduced. In the subsequent loop, chunks of data are submitted to ALSA. ALSA will merge them to a continous stream of data and transmits it to the hardware. It is of great importance that the stream does not tear off, i.e. that there is enough data available at all times. In the case of too few data for proceeding (referred to as underrun), ALSA will stop and the user will hear a clicking sound. Although, submitting too much data increases the latency between submitting the data and playing the data back. This is the central trade-off for building low-latency applications.

```
1  #include "alsa/asoundlib.h"
2
3  int main(void) {
4    int i,j;
5    snd_pcm_t *handle;
6    snd_pcm_sframes_t frames;
7    snd_pcm_uframes_t offset;
8    const snd_pcm_channel_area_t *areas;
9
10   snd_pcm_open(&handle, "hw:0,0", SND_PCM_STREAM_PLAYBACK, 0);
11   snd_pcm_set_params(handle, SND_PCM_FORMAT_S16,
12           SND_PCM_ACCESS_MMAP_INTERLEAVED, 2, 48000, 1, 500000);
13
14   for(i = 0; i < 200; i++) {
15     if(i == 1)
16       snd_pcm_start(handle);
17
18     frames = 1024;
19     snd_pcm_mmap_begin(handle, &areas, &offset, &frames);
20
21     short *data = (short*)(areas[0].addr
22             +(areas[0].first+offset*areas[0].step)/8);
23
24     for(j = 0; j < frames*2; j++)
25       data[j] = j<<9;
26
27     snd_pcm_mmap_commit(handle, offset, frames);
28     snd_pcm_wait(handle, 1000);
29   }
30
31   snd_pcm_close(handle);
32   return 0;
33 }
```

Sourcecode 3.1: *ALSA application producing a sawtooth wave*

Chapter 4

Audio over IP

Based on the experiences made in previous work (cf. Sect. 2.3), a software was developed that enables music performances via the Internet. Just like JackTrip [9] and most other Audio over IP applications, this software uses UDP as transport protocol via the Internet. This enables lower network latency as it would be caused by the retransmission technique of TCP but induces the need to handle packet loss and packet reordering in the application.

In contrast to the existing approaches, the application developed in this thesis has to take the additional constraints of an embedded system into account. First of all, this includes the reduced computational power. This is especially important, because a data compressing coding algorithm as well as error concealment should be used as in Soundjack [12] in order to reduce the data rate and compensate for missing packets.

4.1 Low-Latency Audio Transmission

In order to provide a minimal end-to-end latency, while considering the low computational power in embedded system, an adaptive receiver technique was developed for this thesis. It is based on buffers that dynamically adapt to the network conditions and packet loss handling with deadline control.

The main issue that has to be faced is the jitter introduced by the network as presented in Sect. 3.2. The jitter causes an unsteady stream of packets arriving at the receiver. In contrast to Internet applications like web browsing, e-mail or file downloads, it is necessary for audio streaming to reassamble these into a steady stream (cf. Sect. 3.4.5). This is done by the use of a data buffer in the form of a queue or a ring buffer. Incoming data is stored at the end of the queue, while the data is retrieved at the front in regular time intervals.

This produces a steady data stream of data again, that will not be cut-off as long as there is always enough data in the queue so that no packet arrives after it should be processed. For bounded jitter (i.e. $\delta_{min} \leq \delta_n \leq \delta_{max}$), this can be assured by delaying the first packet by $\delta_{max} - \delta_{min}$ and continue with regular time steps of $\frac{N_F}{f_s}$. The proof for this can be found in the appendix on page 66.

Although, the requirement of a bounded jitter is too strong for the Internet. There are two causes for that: First of all, not all packets will arrive at all. They might get stuck somewhere and dropped because of an overloaded router. Secondly, even if the packet finally arrives, the delay might be much too long lead to an unacceptable end-to-end latency. Furthermore, this might even lead to a packet outrunning another packet leading to packet reordering. Therefore, all packets with a very high delay and especially packets arriving after their successor (identified by a sequence number) are dropped as if they had never arrived at the receiver. All those lost packets would lead to the buffer running out of data (buffer underrun). Therefore, a special packet loss handling takes place that generates compensation data right before a buffer underrun would take place.

For the sake of completeness, another error condition (buffer overrun) might occur when there is too much data in the queue to hold another packet. Besides dropping the incoming packet, an effective countermeasure is adequate memory allocation. As long as the sampling frequencies are equal[1], the memory consumption is bounded as proven in the appendix on page 67.

4.1.1 Buffer Adjustment

Proper jitter compensation is directly conflicting with a low end-to-end latency, because it introduces additional delay. It is therefore required to find a fair trade-off by using just enough jitter compensation to maintain a satisfactory audio quality. This is achieved by controlling the queue length, i.e. the amount of data in the buffer. The average queue length $\overline{Q}$ is measured and regularily tuned to a target length Q_{target} by inserting or dropping data so that the jitter is suitably absorbed.

For determining the target length Q_{target} as the right average amount of data, the standard deviation of the queue length is measured, too. This is not done continuously, because the standard deviation σ_Q itself is fluctuating even for longer time intervals because of the self-similar nature of the delay distribution [33]. Instead, there is a measurement phase right after the connection was established for calculating σ_Q . In this phase, the queue length is tuned to a high value for providing an undisturbed measurement. After the phase, the target length is calculated as

$$Q_{target} = \beta \cdot \sigma_Q. \tag{4.1}$$

β is a hand-tuned factor for controlling the amount packets of lost packets and therefore the audio quality. The relationship between packet loss and audio quality was investigated in [22].

4.1.2 Packet Loss Handling

In order to prevent a buffer underrun when packets are lost, new data has to be created in that case. Three methods are implemented:

[1] This is not trivial, because the sampling frequencies are derived from high frequency clocks generated for example by temperature dependend quartz circuits. Clock synchronization is out of the scope of this thesis. An interested reader is referred to [32].

1. Silence: Instead of the lost packet, play back silence. This leads to a clicking sound. This is annoying if it happens too often, but might be acceptable if the amount of lost packets is low.

2. Wavetable mode: Use the last packet received and play it back again. The perceived sound depends on N_F , the amount of lost packets and the music itself, but it might be less perceivable than silence (cf. [9]).

3. Error concealment: Use the last packets to extrapolate the lost packet. An existing implementation was presented in Sect. 3.3. A good error concealment might provide good sound quality, but it requires a high computational effort.

The question remains, when to conduct the calculation. One approach as presented in [22] is the preventive error concealment that takes place just after each packet reception. When a packet is missing, the already calculated packet is taken. This approach induces a high computational effort that is not feasible for embedded systems with low computational power. Therefore, it is more adequate to conduct the calculation only for the lost packets. The main principle is presented in Fig. 4.1 for a single stream.

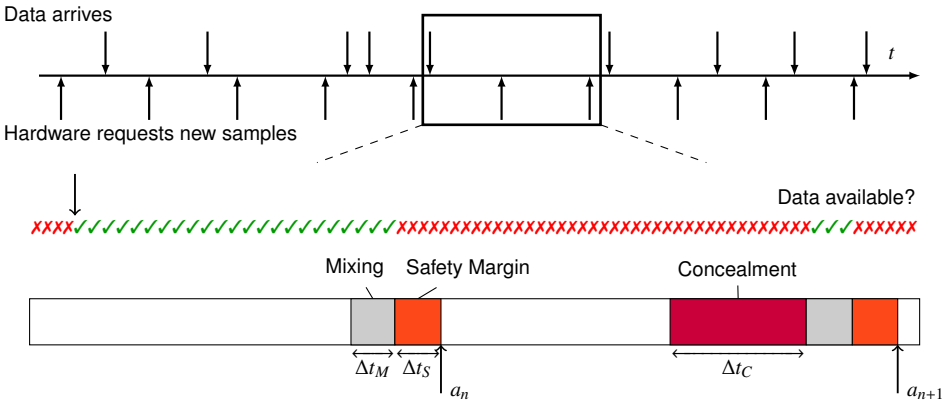


Figure 4.1: *Timing for playback*

The upper time line depicts the packet reception sequence $(r_n, r_{n+1}, \ldots)$ on the upper half and the sequence of requests of the audio hardware $(a_n, a_{n+1}, \ldots)$ on the lower half. Below, a section of this time line is depicted. A bar indicates if there is valid data in the buffer. When there is data in the buffer, it is not necessary to conduct the concealment. The processing can start shortly before the hardware request. When there is no data in the buffer, the processing has to start earlier, because extra time is needed for the concealment. This way a dynamic deadline is maintained to decide when it is no longer feasible to wait if the packet eventually arrives and the processing should start. This requires prior measurement of the time needed for concelment Δt_C and for mixing Δt_M. Furthermore, a safety margin is required for compensating impurities in timing that result from reduced real time capabilities. For the general case of N streams and R packets still missing, the processing has to start when the time until the hardware request $a_n - t$ is

$$\Delta t_P = R \cdot \Delta t_C + N \cdot \Delta t_M + \Delta t_S. \tag{4.2}$$

4.2 Implementation

Based on these considerations, a software was developed that runs on the Raspberry Pi. The core is written in C++11 and embraces about 5000 lines of code. A class diagram of the main classes is depicted in Fig. 4.4. Some helper classes (for example for logging and evaluation) are not shown. The main class that will be instantiated by the main function is JamBerry. It is composed of the list of the connected remotes and two controllers. The capture controller for transferring data from the hardware and the playback controller for transferring the data to the hardware.

The controllers contain the associated interfaces to the network and the audio hardware. Both transfer the data to and from the hardware by means of a subclass of AudioProcessor. For the capture part, this is the downmixer that mixes the stereo signal down to a mono signal. For the playback part, this is the remote mixer that mixes the different streams according to the pan and volume settings. The difference between the controllers is the association to the stream classes, because there is only one encoder stream that is a member of the capture controller, but each remote has an own decoder stream. The decoder streams are members of the remotes and can be accessed via iterating the remote list. The controllers themself are implemented as finite state machines (FSM). They are triggered by the ALSA library via the devices (PlaybackDevice and CaptureDevice). The playback controller is presented in Fig. 4.8. After initialization the stream is started. As soon as there are N_F frames available in the ALSA buffer, the reading process is started.

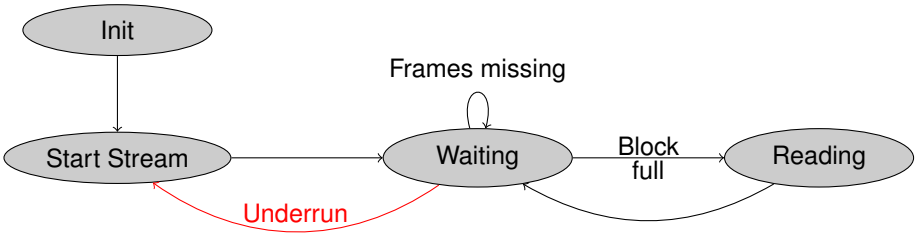


Figure 4.2: *Finite state machine of the capture controller*

The playback controller is slightly more complex because it might be the case, that there is already too much data available in the ALSA buffer and the controller has to wait. If this is ensured, the controller waits until there is not time left and the writing process has to be started in order to avoid a buffer underrun (cf. Sect. 4.1.2).

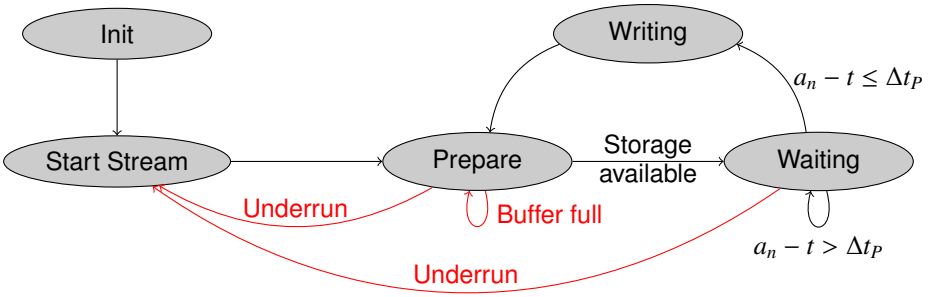


Figure 4.3: *Finite state machine of the playback controller*

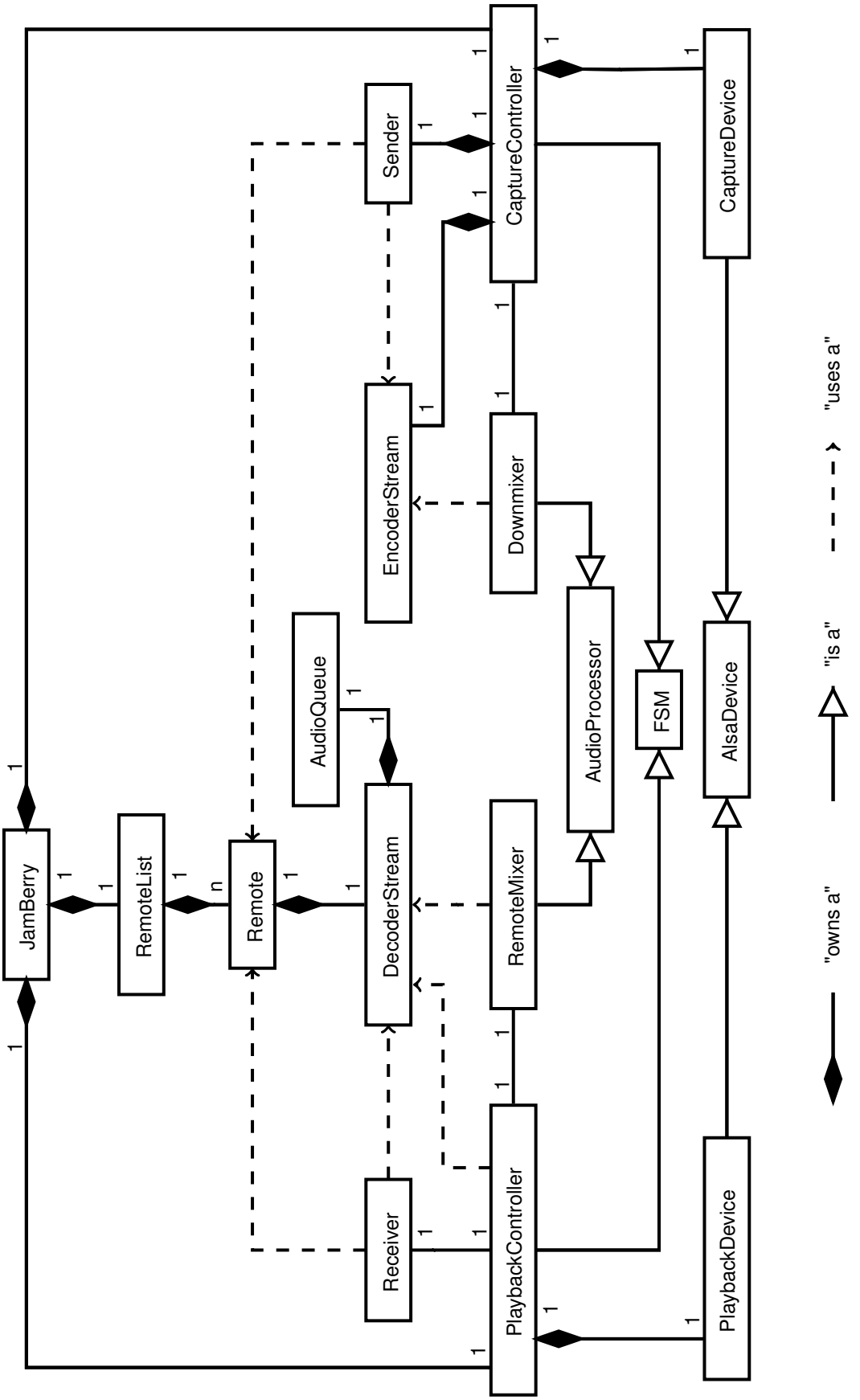


Figure 4.4: *Class diagram of the main classes*

4.2.1 Data flow

This section describes the flow of the audio data from the hardware of the sender to the one of the receiver as depicted in Fig. 4.5. In the beginning, the analog audio is sampled by the ADC of the audio codec and transferred to the Raspberry Pi via the I^2S connection. The ALSA kernel driver will fetch the data and push it into a buffer. From there it is regularily transferred, fetched, and processed by the downmixer controlled by the capture controller. The current version only supports a single encoder stream, so the stereo signal from the hardware has to be mixed down to mono. The extension to multiple channels is easily possible. The downmixer writes the data to a storage inside the encoder stream.

The encoder stream will take the data from there and encodes it into a second storage. Furthermore, it adds a header with information that are important for decoding (for example the frame size and the used codec). The sender finally sends the packet out to the Internet. It uses the list of all connected remotes and sends the packet once for each remote address. The packet is also send to the local address so that it is possible to mix the own signal to the audio output. It is send uncoded, because it is not send via the Internet and thus saves computational effort for decoding the own signal.

The packet is fetched by the receiver at the remote. The sender's address is read from the packet and the receiver chooses the according decoder stream. The decoder stream will analyze the content of the header and chooses the appropriate decoder. Therefore, it is not necessary that all sender have to agree about a single coding configuration or even the buffer size. This is enabled by a frame based audio queue that receives the decoded data.

The capture controller will regularily fetch data from the queue. When data is missing, it will start the concealment for producing replacement data. The data from the different streams is then mixed directly into the buffer of ALSA by means of the mixer. ALSA finally transferrs the data to the audio codec for transferring it back into analog domain.

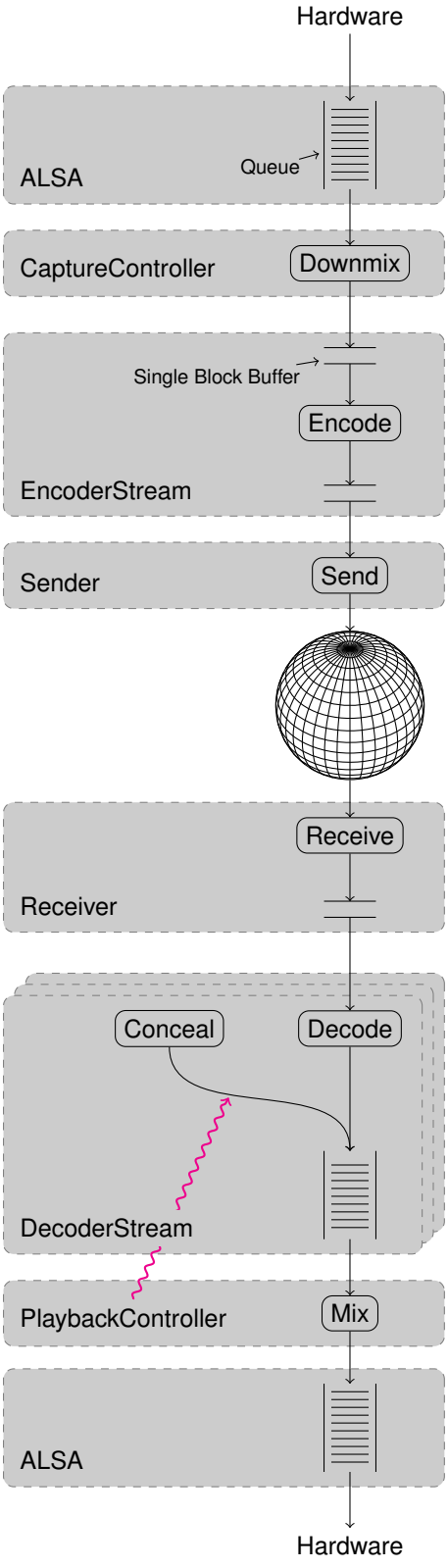


Figure 4.5: *Data flow*

4.2.2 Thread Activities

Several threads control the data flow. These are the playback thread, the receiver thread and the capture thread (includes sending). In compliance with Sect. 3.4.4, they are running with SCHED_FIFO real time priorites and can therefore preempt other processes running on the Raspberry Pi (but not the kernel). The playback thread has the highest priority, followed by the capture thread and the receiver thread.

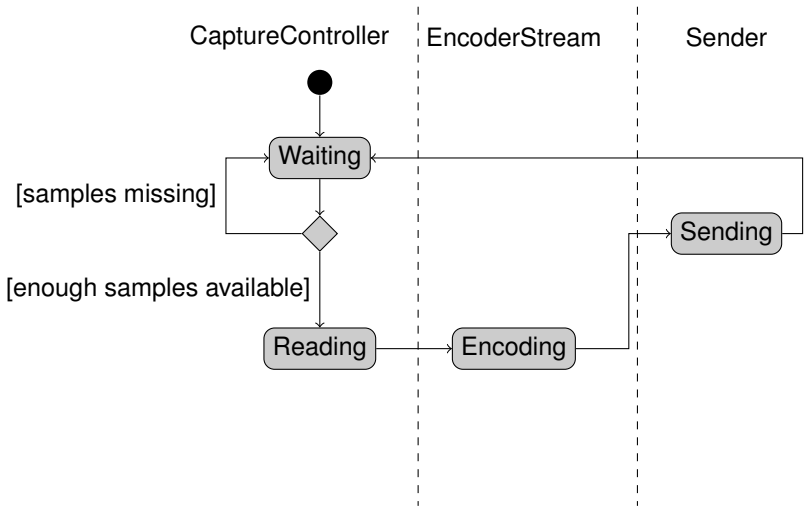


Figure 4.6: *Activity diagram of the capture thread*

The activities of the capture thread are depicted in Fig. 4.6. It blocks until it gets a signal of ALSA. If enough frames N_F are available, it reads the data from the ALSA buffer. The encoder stream will now encode the data and send it via the sender class.

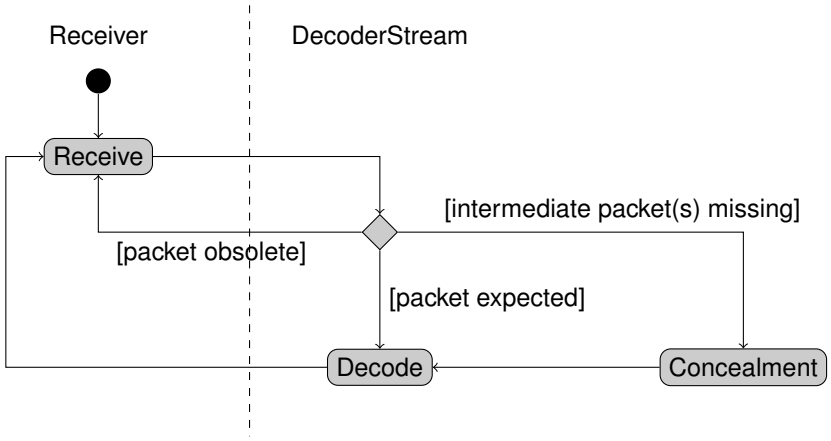


Figure 4.7: *Activity diagram of the receiver thread*

The receiver thread is responsible for receiving the packets from the TCP/IP stack of the kernel and finally store them into the queue. It blocks until a new packet arrives, writes the packet in a buffer and calls the decoder stream. The decoder stream will analyze the sequence number and compare it to the expected sequence number. If the expected sequence number is larger, the packet was already concealed and will be dropped. If it is larger than expected, one or multiple intermediate packets are missing, so the missing packets will be concealed. At least

after the concealment, the expected sequence counter equals the one of the incoming packet and it will be decoded and put into the queue.

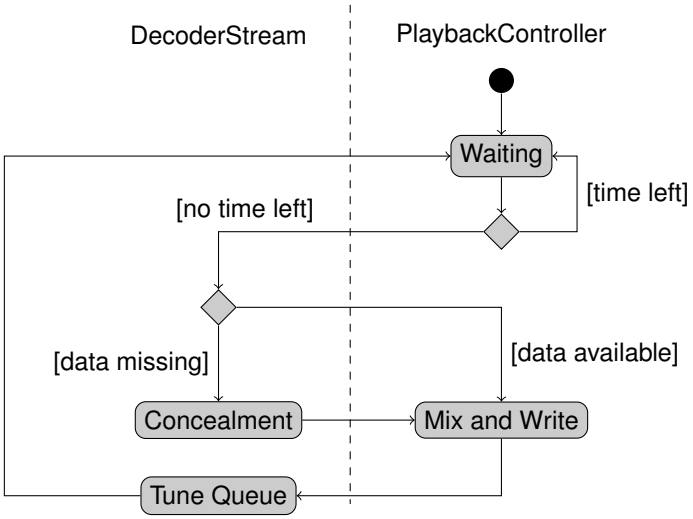


Figure 4.8: *Activity diagram of the playback thread*

The playback thread calculates the current deadline according to Eq. (4.2). When it is woken up, it recalculates the deadline. When there is no time left, it starts the processing. Each decoder stream is instructed to ensure there is valid data in the queue. If there is no valid data, the decoder stream will start the concealment. After all decoder streams are ready, the data is mixed directly into the ALSA buffer according to the current volume and pan settings.

After the data was submitted to ALSA the queue tuning takes place. The current amount of data in the queue is taken to calculate the average queue length. After several iterations (currently 2000), the length of the queue is adjusted according to Sect. 4.1.1.

4.2.3 Graphical User Interface

For controlling the application, a graphical user interface written in Javascript was build. In Fig. 4.9, the data flow is depicted. The user interface runs in a browser and provides the additional advantage, that the control can take place from a remote computer, too. Even multiple connections from various places at the same time are possible. A screenshot of the user interface is shown in Fig. 4.10.

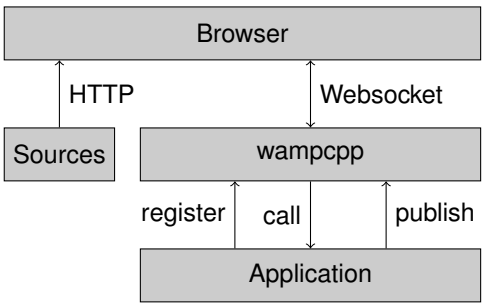


Figure 4.9: *Data flow for graphical user interface*

The user interface is transferred to the browser running on the Raspberry Pi or the one on the remote computer when connecting to the Raspberry Pi. Particular, this is the Javascript code, a Scalable Vector Graphics (SVG) image and a single web page that brings everything together. The proceeding communication takes place over a seperate channel based on the WebSocket protocol that enables a permanent bidirectional connection between the core application and the user interface.

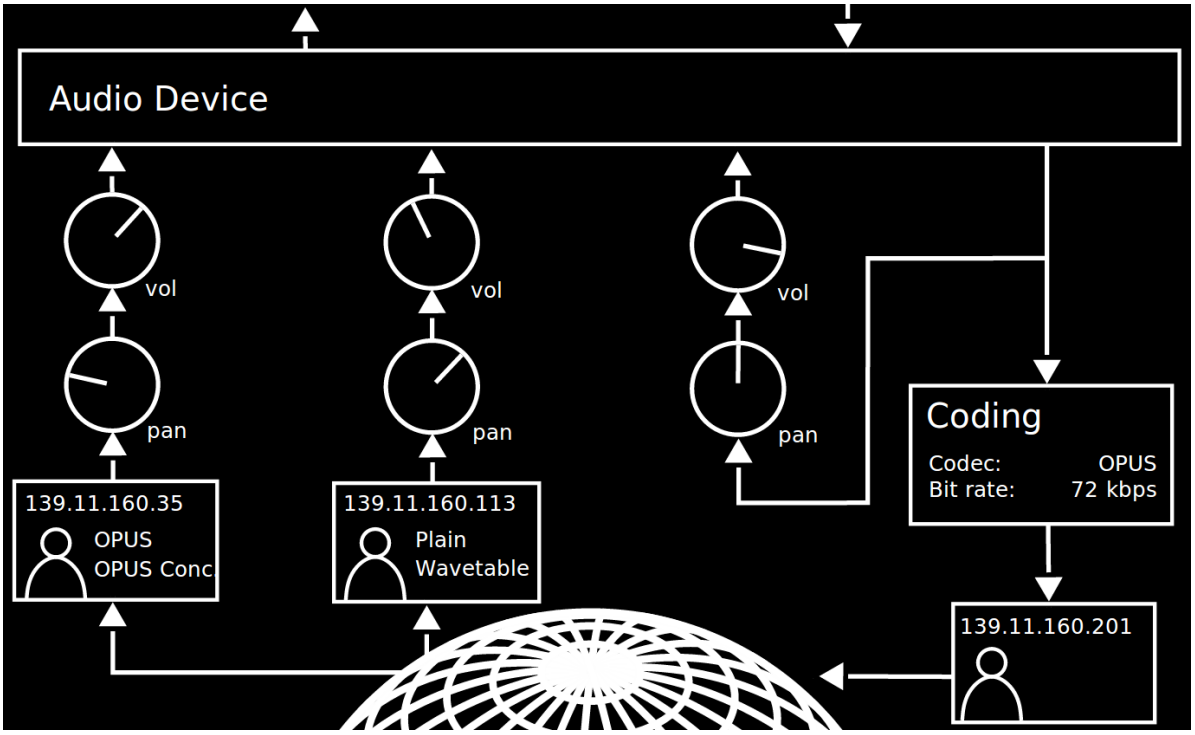


Figure 4.10: *Graphical user interface*

As subprotocol, WAMP v1 [34] is used. It provides two abstractions that make the communication very convenient: The first one is remote procedure call that enables the user interface to call methods of the core application as if it were a local function. The second one is the publish and subscribe design pattern. At the core application as well as the user interface it is possible to subscribe to topic. When something is published to the topic, a callback will be called. For example, when turning a volume knob, the new value is published to the volume topic. The core will receive this event and set the volume appropriately. At the same time, all other connected user interfaces get the same event and can update their display accordingly.

At the Javascript side, the AutobahnJS [35] library is used. For the other side, there was no C++ library before. Therefore, a WAMP library for C++ was developed. It is build upon the existing libraries WebSocket++ [36] and JsonCpp [37]. The latter is used, because WAMP transfers the data in JavaScript Object Notation (JSON) format. C++ classes can link own methods to Uniform Resource Identifier (URI). When the associated remote procedure call is called or it is published to the associated topic, the method will be called. Since the data is transmitted in JSON format and the C++ methods expect the data in primitive data types (such as int, float etc.), templates are used that automatically detect the required types and convert the data accordingly.

Chapter 5

Audio Kernel Driver

An integral task of this master thesis was the development of an audio driver. The system developed is based on a Raspberry Pi. It is a single-board computer build around a Broadcom BCM2835 system on chip (SoC) featuring an ARM11 processor core and a proprietary GPU. Most importantly for this application, it features an integrated interchip sound (I^2S) interface for connecting external audio codecs. As operating system Linux is used that (with a modified kernel) already provides support for most of the peripherals. However, it lacks support for the I^2S peripheral. Some first steps for accessing the I^2S peripheral were made before,[1] but there was no generic audio driver supporting audio output and input. Therefore, it was necessary to implement a kernel driver.

The control of the SoC at such a low-level, as it is necessary for writing a kernel driver, takes place via special register accesses. From the programmers view this looks like using a pointer pointing to an adress within the main memory. This principle is called memory mapped I/O in contrast to isolated I/O, where the input/output (I/O) is accessed via special processor commands. The addresses as seen by the programmer are internally mapped to bus addresses of the Advanced Microcontroller Bus Architecture (AMBA) bus that connect the various components inside the SoC. This is also important for programming a kernel driver, because for some configurations the bus addresses and not the memory mapped addresses have to be used. The concrete mapping of peripherals to addresses can be taken from the datasheet [38]. For example the bus address of the I^2S peripheral is 0x7E203000. It is mapped to 0x20203000 where it can be accessed by the software.

The driver implemented during this thesis belongs to the lowest software layer as presented in Sect. 3.4.4. It was developed as a part of the ALSA subsystem. Since this driver is targeted on an embedded system, it is precisely a part of the ALSA System on Chip (ASoC) layer. This layer provides a helpful abstraction that is common for embedded systems. For example, it splits the sound drivers in multiple parts that can be changed independently, e.g. for using a new audio codec.

This chapter starts with an abstract description of what is necessary to transfer audio data between an application running on the Raspberry Pi and the audio codec. After that, the overall structure of ASoC and the driver are presented and finally the single components of the developed driver are further explained.

[1] Git repository of philpoole: https://github.com/philpoole/snd_pi_i2s_poor_dma

5.1 Abstract Description

For understanding the implementation of the kernel driver, it is necessary to examine how the data has to flow from the audio codec to the random access memory (RAM) where it can be accessed by applications. The simplified data flow is depicted in Fig. 5.1 with continuous lines.

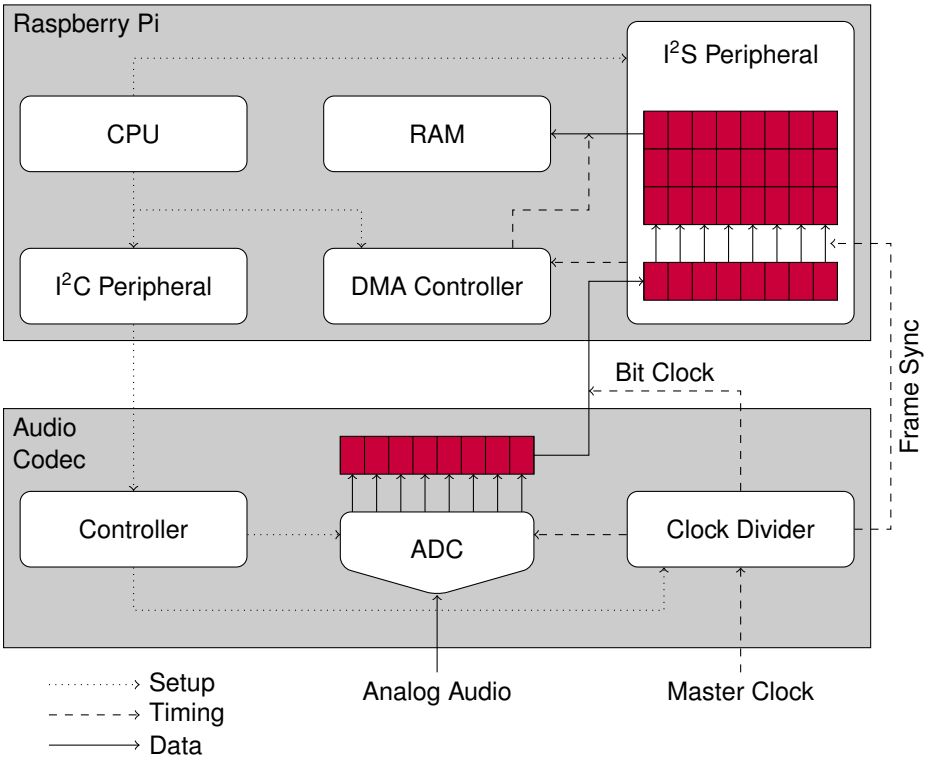


Figure 5.1: *Data and control flow for recording*

The analog audio data is sampled by means of an analog to digital converter (ADC) inside the audio codec. The sampling process is controlled by f_s that is derived from the master clock of the codec. For each sampling interval, a sample is stored in a register. The content of the register is send bit-wise via a I²S connection as presented in Sect. 3.4.2.2. At the receiver, the bits are written into a register again. Several samples are stored until the direct memory access (DMA) controller transfers them to the RAM where they can finally be accessed. The same applies to the way back (not depicted in Fig. 5.1): The DMA controller transfers a block of samples to the I²S peripheral where it is stored until it is sent bit-wise to the audio codec. The data is transferred into analog domain again by a digital to analog converter (DAC) in the audio codec.

Once set up, most parts run automatically without the intervention of the CPU. The CPU only gets a signal when there is new data in the RAM. This is directly forwarded to ALSA that handles the event and controls the data flow to the audio application. For implementing the kernel driver, the main focus is the configuration of the different parts and how to link them together. This is depicted by the dotted lines showing how the central processing unit (CPU) (as executor of the driver software) is responsible for the setup of the components.

Finally, the dashed lines present how clocks and signals are used for controlling the data flow. The master clock is generated internally or externally of the audio codec and has the

highest frequency (for example $256 \cdot 48\,\text{kHz} = 12.288\,\text{MHz}$). It is fed into the clock divider that generates the different clocks needed. These are mainly the clock for the ADC and the associated filters, the bit clock for controlling the serial transmission of the audio data and the frame sync for signalling the boundaries of a single frame. It is also possible to let the receiver generate the bit clock and frame sync. It depends on the concrete hardware and application which method is better (for example which part can provide a more stable clock).

The I²S peripheral will generate a data request (DREQ) signal when there is enough data buffered (this can be adjusted by the driver). As a reaction to that, the DMA controller will start a data transfer to the RAM. In an analogous manner, the I²S peripheral will generate a DREQ signal when it needs more data for the transmission to the audio codec.

5.2 Structure

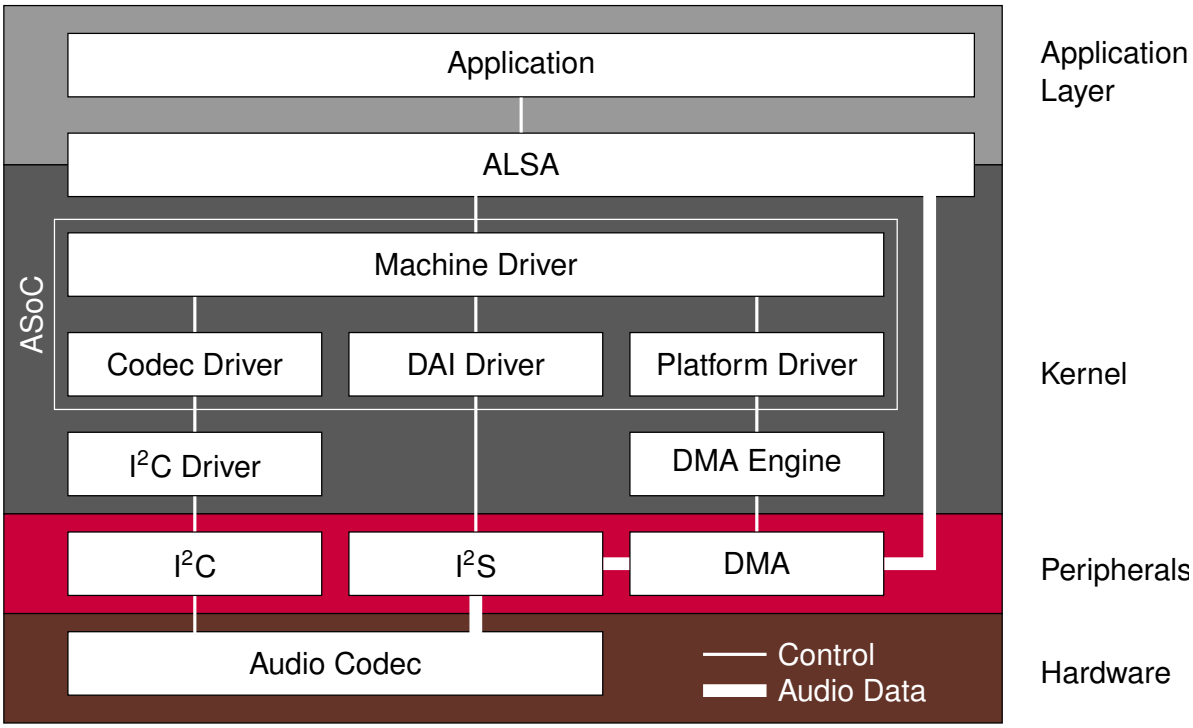


Figure 5.2: *ASoC structure*

The characteristic structure of an audio driver inside of ASoC and the their interaction with the external components is depicted in Figure 5.2. The different parts of the driver are as follows:

Codec Driver It provides control access to the codec. It is for example responsible for initialization, volume or gain. This typically takes place via a serial connection such as inter integrated circuit (I²C). Another common possibility is Serial Peripheral Interface (SPI). The serial interface is not directly accessed, but via an appropriate kernel subsystem.

DAI Driver The digital audio interface (DAI) driver accesses the hardware for writing and reading of the audio data. The DAI might be I²S or a similar format. Settings have to be made according to the concrete data format.

Platform Driver The platform driver is responsible for controlling the data transfer between DAI peripheral and ALSA. Usually (but not always) this takes place via DMA. Since DMA is a technique used by many other drivers, there is another associated kernel subsystem called DMA engine.

Machine Driver The machine driver brings everything together. It contains only the code that is necessary for a specific combination of codec, DAI and platform. Therefore, only this part has to be rewritten if for example a new audio codec should be used.

5.3 DMA Driver

In order to enable fast data transfer from the I^2S peripheral to the RAM, DMA is used. It is not mandatory for accessing the audio hardware but it significantly reduces the CPU load. As presented in Sect. 3.4.3, there are two possibilites for DMA, depending on the peripheral being a bus master or not. The I^2S peripheral inside the BCM2835 belongs to the latter case [38]. Therefore, a DMA controller has to be used. There is already a proprietary application programming interface (API) for the DMA controller inside the Raspberry Pi kernel but it does not fit into the common DMA engine subsystem of the kernel that is used by ASoC. Therefore, the implementation of a wrapper that matches the existing API to the one of the DMA engine subsystem was performed.

The basic operation principle is the submission of tasks. A task is defined as a control block containing source and destination address, data length, and some control flags. These addresses are an example for the case that AMBA bus adresses have to be used rather than memory mapped adresses as mentioned at the beginning of the chapter. This is because the DMA controller directly works on the bus and has no information about the mapping made inside the ARM core. An example for a control flag is the DREQ flag. With this flag set, the task is executed not before the peripheral signals that there is new data available. Another flag indicates that the CPU should be signaled when a transfer is finished. A further data field specifies the address of the next control block. This builds a linked list of tasks that should be conducted one after another.

The DMA subsystem specifies various data transfer modes that determine how the different tasks should be linked. Amongst others, these are slave_sg and dma_cyclic. The sg is an acronym for scatter/gather, meaning that the data coming from a peripheral is scattered to multiple memory positions or gathered from multiple memory positions. For a peripheral it is common to have a single register for reading and writing the data, where the data appears sequentially. In order to build a block of memory, the software might use this mode for instructing the DMA controller to increment the writing address after each reading from the register. After a block is filled, the next task points to the beginning of the next block in memory. The other mode, dma_cyclic, equates to slave_sg, the only difference is that in this mode, the controller starts from the beginning over and over again until it is explicitly stopped. Currently, the driver only supports dma_cyclic mode. It is the most adequate mode for the audio driver, because it implies the least involvement of the CPU.

5.4 Platform Driver

The task of the platform driver is to connect the DMA engine driver to ASoC. This mainly includes setting up the memory that should be used for the DMA according to the audio related settings (for example buffer size). Since both APIs (the one of the DMA engine driver and the one of ASoC) do not depend on the concrete platform, this driver looks nearly the same on many platforms. Therefore, a generic DMA engine driver is included in ASoC that deals with this task[2]. Therefore, the platform driver only contains some settings and a link to the generic DMA engine.

5.5 Codec Driver

The codec itself is configured via a serial interface. It was not necessary to write a codec driver, because there already exist drivers for many popular audio codecs. Due to the modular structure of ASoC, it was easy to use them without changes.

There are two main tasks that involve the codec driver. The first is the setup needed to start the transmission. This includes setting up the sampling rate and the DAI bit format. Furthermore, specific commands are used to enable and disable different parts of the audio codec, thus making the codec to begin sampling data. The second task is the control of parameters while the audio codec is running. These commands, such as volume control, are passed on to ALSA so that they can be controlled by user interfaces such as alsamixer.

5.5.1 I²S Driver

For using an I/O peripheral, it is first of all necessary to establish the physical connection. On the Raspberry Pi, the four lines of the I²S peripheral can be accessed via a pin header. Before they can be used, the port has to be configured to provide the right signals. For SoCs or microcontrollers it is common that a data line coming out of the IC can adopt multiple functionalities. This provides great flexibility while maintaining a lower number of data lines, reducing cost and footprint. For example, the bit clock of the I²S port might also be configured as SDA line of another I²C port.

After that, a sequence of further configurations is necessary as presented in the datasheet [38]. The following steps are necessary for starting a DMA based I²S transmission on a BCM2835:

1. Enable the peripheral.
2. Configure the format and start the clock.
3. Clear the data buffer. Otherwise, old data might be transmitted.
4. Set up the thresholds for generating the DREQ signals and enable them.
5. Set up and start the DMA controller.
6. Start the transfer.

[2] In fact it was just introduced during the time of writing this thesis. The first version of the proposed driver had an own implementation of the interconnection

The audio peripheral of the BCM2835 is very flexible. In fact it is not only a I^{2}S peripheral, but supports also various other serial audio formats. It is possible to configure frame sync period, duty cycle, position and length of the channels with bit clock resolution. Although, there is only a single frame sync for both directions. Therefore, it is not possible to use for example different sampling rates for input and output. This is in contrast to the typical assumption of ALSA where the streams can be configured independently. Additional effort has to be made for ensuring that both streams have the same settings by interrupting the configuration of a second stream in case the first stream was already configured with a different setting.

5.5.2 Clocking

In many cases, the I^{2}S clocks are generated by the audio codec itself. This is the case for the developed hardware, too. It has a seperate clock generator that is controlled via I^{2}C and provides a master clock. The audio codec generates the I^{2}S clocks from the master clock. Although, it might also be the case that the I^{2}S clocks should be created by the BCM2835. This might be because the audio codec can not provide the clock or does not produce the right clock for the desired sampling rate. In that case one has to utilize one of the internal clocks and divide them down to fit the needed frequencies. This is of particular difficulty, because the BCM2835 does not provide a clock having an adequate audio frequency. The frequencies that can be easily accessed are 19.2 MHz and 500 MHz. The frequencies needed for a sampling rate of 48 kHz are the sampling rate itself as frame sync and a multiple of that, for example 1.536 MHz as bit clock as derived in Sect. 3.4.2.2. None of the above frequencies can be divided by an integer divider to reach this frequency.

There are two solutions for this: The most common is the use of a non integer divider. This clearly can not produce a perfect clock, but produces a clock with a varying frequency that has the desired frequency as average. An early patent presenting this technique is [39]. It uses a structure that is similar to the one found in delta-sigma ADCs. For minimizing the undesirable effect, the BCM2835 uses multi-stage noise shaping (MASH) that shifts much of the noise into higher frequencies where it harms less at the cost of an overall increase of noise [40].

The second solution utilizes the fact, that many audio codecs do not require a specific ratio of bit clock to frame sync as long as there are at most enough bit clock cycles within a frame sync cycle to contain the desired word length, and the bit clock frequency is not too high. The state of the data line in the remaining bits does not care as shown in Fig. 5.3. This leads to the uncommon number of 40 bit clock transitions per frame sync period resulting in a bit clock frequency of $40 \cdot 48\,\text{kHz} = 1.92\,\text{MHz}$. This can be generated form the 19.2 MHz clock by an integer divider of 10. Both solutions are implemented and the latter one is chosen for matching sampling rates (e.g. 48 kHz and 96 kHz).

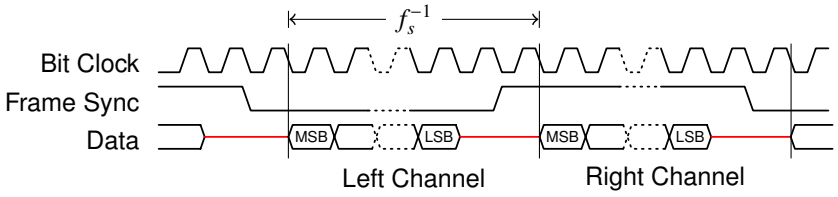


Figure 5.3: *Serial audio transmission with word length ≠ frame sync period*

5.6 Machine Driver

The machine driver is usually very small since it only contains configurations for merging the different parts. The other drivers are written in a very generic fashion for supporting a plurality of configuration options such as the format of the serial audio transmission. The supported configuration options are not necessarily equal for all involved parts so it is the task of the machine driver to choose the appropriate format that all involved parts support.

Furthermore, the machine driver includes configurations that are specific for a concrete hardware. This could be for example the control of a digitally controlled external amplifier or the configuration of the master clock. The latter can be restrictive such as allowing only specific sampling rates that can be derivated from the master clock or it could include the configuration of an external IC used for generating the desired frequency.

Chapter 6

Hardware Design

The device developed during this master thesis is based on a Raspberry Pi. It was chosen because of the small size, the low price, and the manifold extension possibilites. The internals were already presented in the last chapter. This chapter will present the external connections and how the Raspberry Pi is integrated into self developed components to build the complete system. Fig. 6.1 contains a picture of the Raspberry Pi presenting the various connection possibilities.

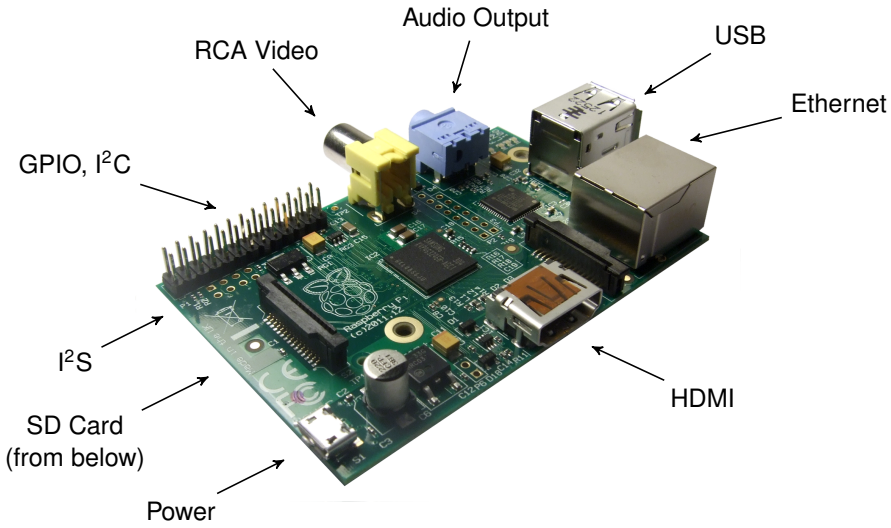


Figure 6.1: *Raspberry Pi*

Unfortunately, the Raspberry Pi features no audio input and the audio output is not connected to a DAC, but a pulse wide modulation (PWM) module. This provides medium quality audio. Therefore, an external audio codec is used for providing audio input and output. It is designed to provide stereo line-level analog connections. This is adequate for stationary environments (such as a rehersal room) where a musician uses mixing consoles for mixing analog signals from multiple microphones and instruments into a single stereo audio signal.

Although, the prototype that was developed for this thesis should also be useable as a standalone device for presenting the Audio over IP technology without the need of external components. It should be possible to connect a guitar and a microphone directly to the device

as well as headphones for audio output. Therefore, a preamplifier and a headphone amplifier are connected to the audio codec.

The interaction of a user with an embedded system is possible in many ways. The most elementary one is the use of push buttons and indicator lamps. The disadvantage is the reduced flexibility, because adding a button requires redesign of the chassis and electronics. Since the Raspberry Pi provides interfaces like a personal computer, the connection of an external display and common input devices (keyboard and mouse) is possible and provides much more flexibility because a graphical user interface can be implemented in software. It is also possible to use a graphical user interface by an external device such as a personal computer, mobile phone or tablet computer. While such a device is present more often than not, it provides the additional advantage of being a remote control. The place where the signals have to be handled (for example the stage) can be distance of the place of user interaction (for example at the front of house).

In order to provide maximum flexibility while waiving the absolute need of external components, a dual approach is used: A graphical user interface was developed as presented in Sect. 4.2.3 that can be accessed by an external computer. The same interface can be accessed by a touchscreen integrated in the embedded device.

6.1 Hardware overview

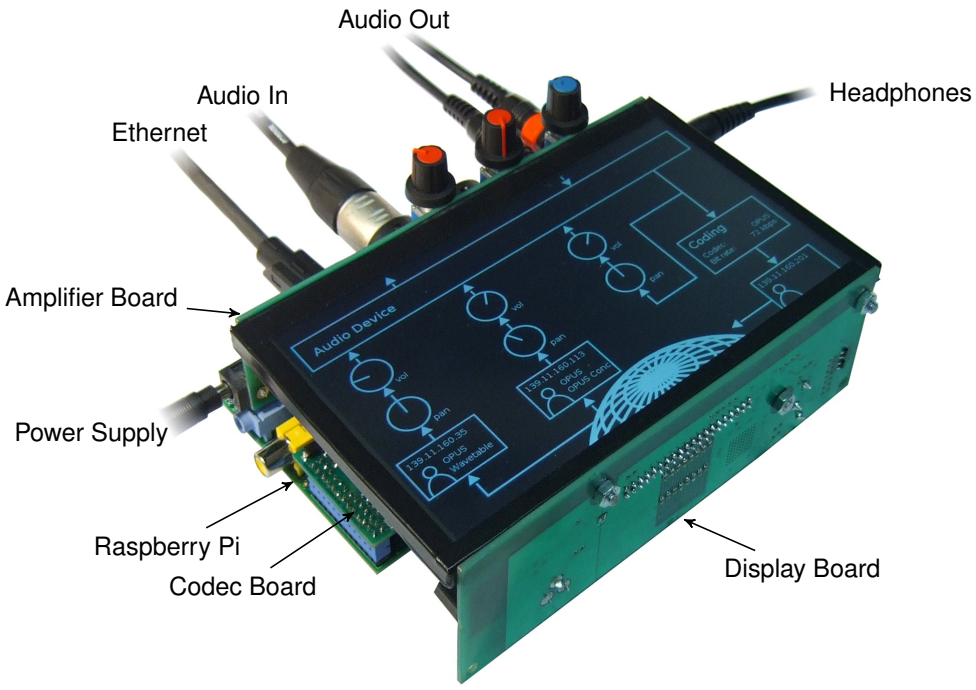


Figure 6.2: *Overall system*

The overall system is depicted in Fig. 6.2, while Fig. 6.3 is schematic representation of the electronic subassemblies and their interconnections. The system is composed of three self-developed circuit boards, the Raspberry Pi and the touchscreen. The core in terms of functionality as well as spatial arrangement is the Raspberry Pi. An opening in the amplifier board

enables direct access to the Ethernet connector as well as to the USB connectors. External input devices can be connected to the USB connectors, but this is normally not needed since the touchscreen can be used for interfacing the system.

The codec board, featuring the audio codec and a generator for the audio clock, is stacked onto the Raspberry Pi and can also be used without the amplifier board for testing purposes or for building up a stripped down version of the device. Perpendicularly connected to the codec board is the amplifier board. A 9 pin D-sub connector provides electrical as well as mechanical connection. The amplifier board provides the amplifiers and audio connectors for typical audio equipment such as XLR connectors for microphones and TRS connectors for guitars and line-level components. A stereo mini TRS connector receives the plug for the headphones. Furthermore, the power supply for the system is located on this board.

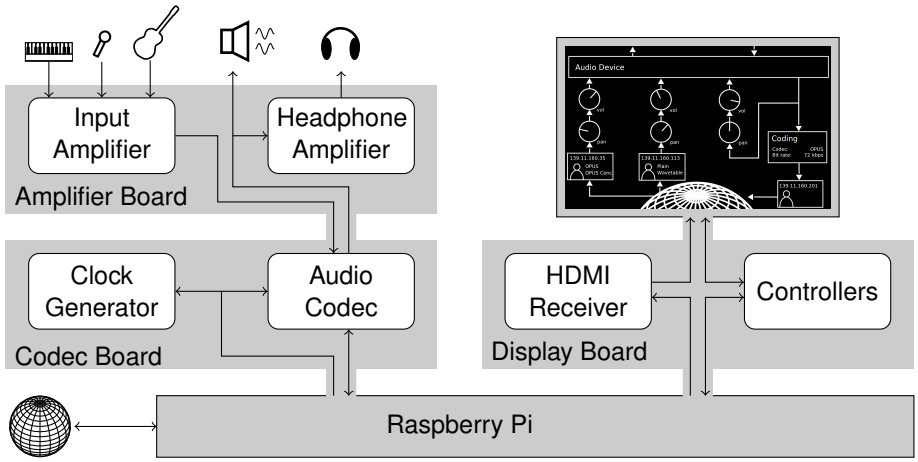


Figure 6.3: *Hardware overview*

The display board is aligned parallel to the amplifier board by means of threaded rods for mechanical stability. The touchscreen lies upon the upper ones and is clamped by the circuit boards. Power supply for the display board (and finally the touchscreen) comes directly from the amplifier board, while the entire communication is conducted via a HDMI cable. While HDMI conventionally only transmits video (and possibly audio) content, it provides a I^2C line typically used for identifying the connected display. This one is diverted to carry the control data to the touchscreen and backlight controllers and touch measurements back to the Raspberry Pi.

6.2 Codec board

The audio codec used on the developed codec board depicted in Fig. 6.4 is the CS4270 by Cirrus Logic. Sample rates up to 192 kHz with oversampling from 32× up to 128× and a maximum of 24 bits per sample are supported. It provides stereo audio input and output and various digital interface formats, in particular I^2S. The supply voltage and thereby the high voltage level of the digital interface ranges from 3.3 V to 5 V. The first one matches the digital signals of the Raspberry Pi. The digital and the analog part of the CS4270 are driven

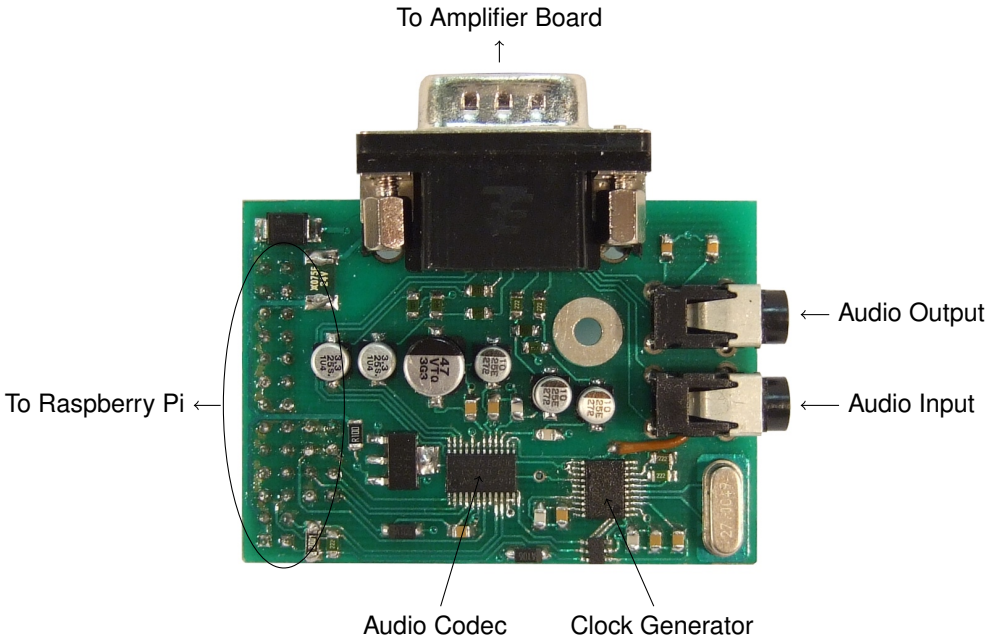


Figure 6.4: *Codec board*

by different power supplies. The digital part gets the power directly from the 3.3 V line of the Raspberry Pi, while the power for the analog part is generated by a linear voltage regulator from the 5 V line in order to reduce noise.

The CS4270 is available in a 24-pin TSSOP package and is therefore solderable by hand. This is important, because a reflow oven for automatic soldering was not available. Surprisingly, this criterion was very restrictive: Most of the more complex IC available today are provided in packages such as BGA or QFN. This has certainly many advantages for industrial products, first of all the smaller footprint, but it considerably complicates the production of prototypes.

The external connections of the codec are build according to the datasheet of the CS4270 and depicted in the appendix on page 69. The analog input and output connections will be examined more closely. Fig. 6.5 shows the connection of a single output channel.

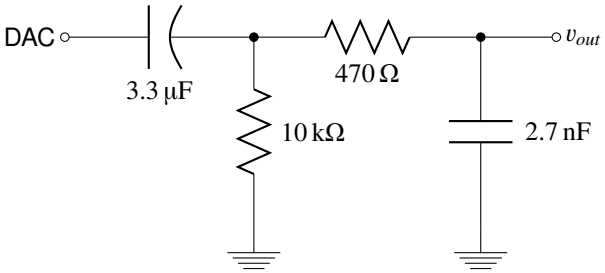


Figure 6.5: *Audio output connection*

The 3.3 µF capacitor and the 10 kΩ resistor build a high-pass filter with cut-off frequency

$$f_c = \frac{1}{2\pi \cdot R \cdot C} = \frac{1}{2\pi \cdot 10\,\mathrm{k\Omega} \cdot 3.3\,\mu\mathrm{F}} = 4.82\,\mathrm{Hz} \tag{6.1}$$

for suppressing direct current (DC). The other components build an anti-imaging low pass filter. Since the CS4270 uses oversampling, the requirements are not very strict. The cut-off frequency of this filter is

$$f_c = \frac{1}{2\pi \cdot 470\,\Omega \cdot 2.7\,\mathrm{nF}} = 125.4\,\mathrm{Hz}. \tag{6.2}$$

This allows for a sampling frequency of 4 kHz with $L = 32\times$ oversampling. In this case, the cut-off frequency should be

$$f_c = L \cdot f_s - \frac{f_s}{2} = 32 \cdot 4\,\mathrm{kHz} - \frac{4\,\mathrm{kHz}}{2} = 126\,\mathrm{kHz}. \tag{6.3}$$

Since the normal operation mode of the final device uses 48 kHz sampling frequency at $128\times$ oversampling, the cut-off frequency could be higher, but it is not necessary, because the attenuation of the filter at $\frac{f_s}{2}$ is only

$$|H\,(2\pi \cdot 24\,\mathrm{kHz})| = \frac{1}{\sqrt{1+(\omega RC)^2}} = \frac{1}{\sqrt{1+(2\pi \cdot 24\,\mathrm{kHz})^2\,(RC)^2}} = -0.156\,\mathrm{dB}. \tag{6.4}$$

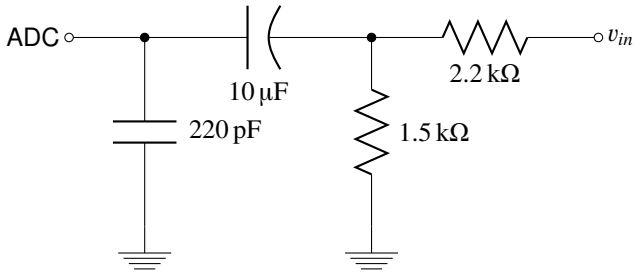


Figure 6.6: *Audio input connection*

The connection to the audio input is depicted in Fig. 6.6. Similar to the output connection, it attenuates low frequencies (for blocking DC) and high frequencies (against aliasing). With a 3.3 V supply voltage, the full-scale input voltage is, according to the datasheet, $0.56 \cdot 3.30\,\mathrm{V} = 1.80\,\mathrm{V} \approx 0.64\,\mathrm{V}_{rms}$. Since line-level signals of $1.55\,\mathrm{V}_{rms}$[1] should be applied, the signal has to be divided down. The divider build up by the resistors is

$$\frac{2.2\,\mathrm{k\Omega} + 1.5\,\mathrm{k\Omega}}{1.5\,\mathrm{k\Omega}} \approx 2.5. \tag{6.5}$$

The source impedance seen by the ADC is the parallel connection of the resistors with a resistance of

$$\frac{2.2\,\mathrm{k\Omega} \cdot 1.5\,\mathrm{k\Omega}}{2.2\,\mathrm{k\Omega} + 1.5\,\mathrm{k\Omega}} = 892\,\Omega. \tag{6.6}$$

[1] There are various standards for line-level voltages. This one is the one used for german broadcasting (ARD) and is relatively high compared to consumer audio equipment.

This value is smaller than 1 kΩ as required by the audio codec. Finally, the input impedance as seen from the outside is

$$2.2\,\mathrm{k\Omega} + 1.5\,\mathrm{k\Omega} = 3.7\,\mathrm{k\Omega}. \tag{6.7}$$

For providing the clock for the audio codec, an external clock generator, the MAX9485, is used. It generates the required clock (for example $256 \cdot 48\,\mathrm{kHz} = 12.288\,\mathrm{MHz}$) from a 27 MHz quartz crystal. An additional DAC is mounted on the board for fine-tuning the sampling frequency.

6.3 Amplifier board

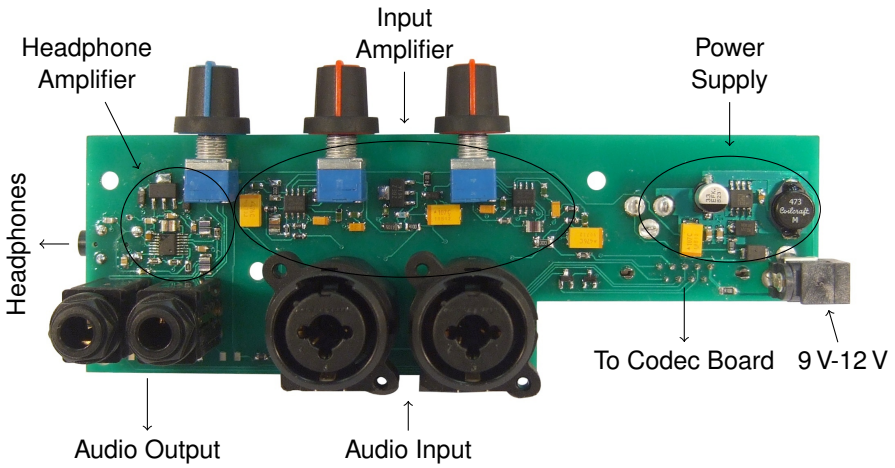


Figure 6.7: *Amplifier board*

The amplifier board in Fig. 6.7 provides various connectors for external audio equipment. Two combined XLR/TRS connectors can receive plugs from microphones, guitars, keyboards and other instruments as well as signals coming from an external mixer. Since they all might have different signal levels, an analog preamplifier amplifies the signals to line-level for feeding them into the codec board. The output signal can be directly accessed via TRS connectors and is fed into a headphone amplifier, too.

A coaxial power connector receives the plug from an external AC adapter providing a voltage of 9 V up to 12 V. The power is fed into multiple voltage regulators providing power for the various components. A LM2675M switching regulator provides power for the Raspberry Pi and the codec board. Two LM1117MPX linear regulators provide 7 V and 5 V for the amplifiers. The use of two different regulators for 5 V results from the characteristic of a switching regulator and the digital components on the Raspberry Pi to produce noise that would influence the amplifier output quality. On the other hand, a switching regulator is more efficient than a linear regulator and is therefore used for powering the Raspberry Pi.

The input amplifier uses a two-stage non-inverting operational amplifier circuit. A single stage is depicted in Fig. 6.8. The operational amplifier is an OPA2134 by Texas Instruments.

Since an audio signal is a AC signal with voltages above and below zero, an additional negative voltage source would be needed for powering the operational amplifier. For simplicity of the

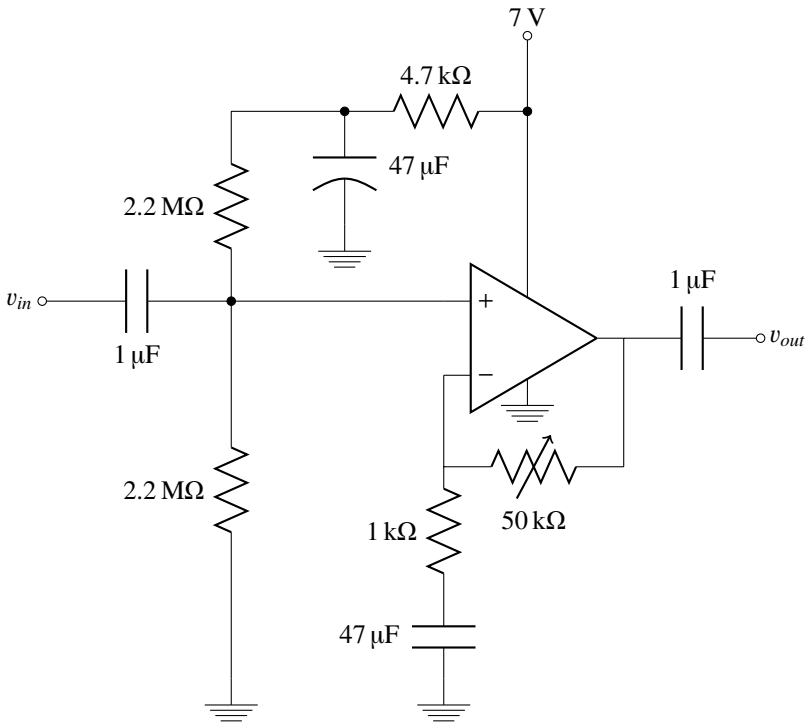


Figure 6.8: *Single stage of input amplifier*

power supply a different approach was used: The (AC coupled) audio signal gets a voltage offset of 3.5 V by using resistors with high resistance (2.2 MΩ). This signal is fed into the positive input of the operational amplifier. The potentiometer in the feedback loop can be adjusted to provide an amplification of 0 dB up to

$$G = 1 + \frac{R_{FB}}{R_G} = 1 + \frac{50\,\text{k}\Omega}{1\,\text{k}\Omega} \approx 34\,\text{dB}. \tag{6.8}$$

This would also amplify the voltage offset, driving the operational amplifier into saturation. Therefore, a capacitor is used for the connection of the feedback loop to ground. For DC, the ideal resistance of the capacitor is infinite, resulting in an amplification of 0 dB regardless of the potentiometer. By chaining two of those stages, an overall amplification of 68 dB can be archived. This high amplification is especially needed for microphones.

For the headphone amplifier, a MAX13331 is used. This IC contains two Class AB headphone amplifiers that can deliver up to 135 mW into 35 Ω headphones. The amplifier is set to a constant amplification of 20 dB. A potentiometer builds up a voltage divider to attenuate the incoming signal. The line-out TRS connectors can also deliver this attenuated signal or the original signal, depending on the assembly. For providing minimum noise at silence, the CS4270 and the MAX13331 both feature MUTE circuits. The CS4270 provides two digital signals, one for each channel, to indicate if audio output is present. The MAX13331 features a single digital input for shutting the amplifier down. For combining the two signals into one, the circuit in Fig. 6.9 is used. With at least one channel not muted, the resistor pulls the signal up and enables the headphone amplifier. When both channels are muted, the MOSFETs start conducting and the signal is pulled low. The amplifier is disabled.

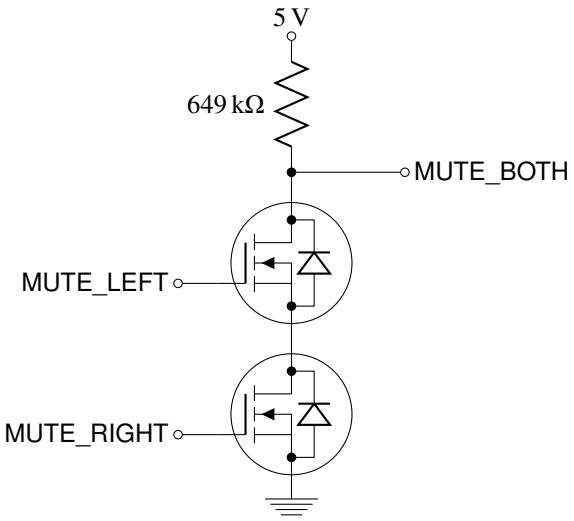


Figure 6.9: *Mute circuit*

6.4 Display Board

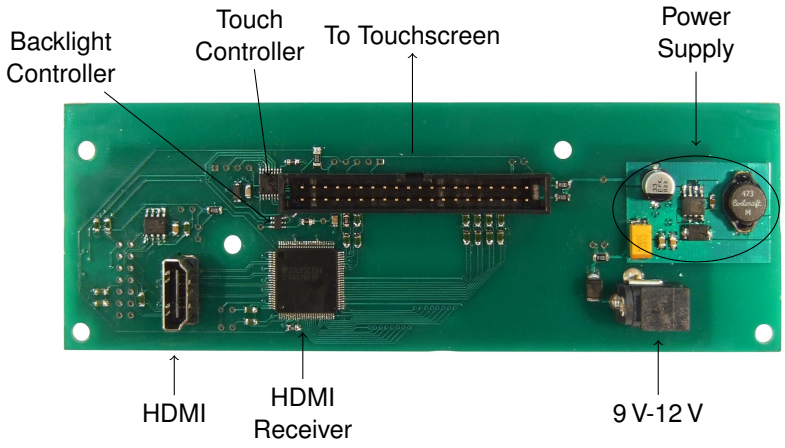


Figure 6.10: *Display board*

The display that is accessed by the display board depicted in Fig. 6.10 is a A13-LCD7-TS by Olimex. It consists of a 7" Thin Film Transistor (TFT) touchscreen AT070TN92 by INNOLUX and a circuit board developed by Olimex. This circuit board has two functionalities that simplifies the development of a prototype. First of all, the AT070TN92 is connected via a flexible flat cable (FFC). This requires a special connector, while the Olimex board provides pin headers that can be accessed by ordinary jumper cables or a ribbon cable. Secondly, the board provides multiple voltage regulators that are necessary for generating the various voltages needed for driving a TFT. Only a single 3.3 V power source is needed for driving the Olimex board. It is generated by a LM2675M switching regulator that is powered directly by the input power supply of the device that is connected through the amplifier board. This switching regulator provides the power for the other components on the display board, too.

First of all, this is the TFP401A by Texas Instruments. It is used for translating the HDMI signals to the digital signals of the display. The use of a TFP101A (a predecessor of the TFP401A) for

driving a TFT display of an embedded system (a BeagleBoard) was previously done by Till Harbaum [41]. The output signal consists of three 8 bit parallel busses (one for each color: Red, green and blue). They change from one pixel to the next, triggered by a dedicated clock line. For assigning the pixels to a rectangular grid, a HSYNC and a VSYNC line seperate the pixels into rows and finally full frames. A further DE line indicates blank times (for example before a frame starts). These signals (28 in total) are directly fed into the display.

A SCDT line indicates when there is no input signal (especially when the Raspberry Pi is shut down). It is used to disable the power supply of the display. For controlling the backlight of the display, a MCP4726 DAC is used. It can be controlled via I^2C and adjusts the feedback voltage of the backlight driver located on the Olimex board.

For enabling touch input, a resistive touch panel is used. It consists of two layers of resistive material. When the user presses the panel, the layers have contact and a position depended resistance can be measured. This is done by means of a AD7879-1W touchscreen controller by Analog Devices. It internally provides median and averaging filters for filtering noise out that would lead to a jittering mouse pointer. The touch panel layers are directly connected to the touchscreen controller that conducts the measurements in regular intervals which can be read out via I^2C.

As mentioned in Sect. 6.1, there is no dedicated I^2C cable connection between the Raspberry Pi and the display board. Instead, the I^2C connection inside of the HDMI cable is used. The traditional use case (to inform about the display properties) is carried out by a 24LC64 EEPROM, but is only included for compatibility with other embedded systems, because the Raspberry Pi does not necessarily depend on this information, because it can also provided via a configuration file. The use of the HDMI I^2C connection requires a small modification of the Linux kernel, because it is usually not accessed by the CPU. The intended functionality can be achieved anyway.

A further characteristic complicates the interface. The voltage level of this I^2C connection is 5 V, but the touchscreen and backlight controllers should be interfaced with 3.3 V. The backlight controller supports I^2C signals up to 5.5 V, but this requires a higher supply voltage. This might lead to a analog voltage that is higher than 3.3 V with accidentially wrong configuration, finally leading to a possible hardware defect of the backlight circuit. The touchscreen controller does not support a higher supply voltage. The only alternative would be another controller with higher supply voltage, but the range of alternative touchscreen controllers in hand-solderable packages (cf. Sect. 6.2) is small. A TSC2003 by Texas Instruments was tested, but the lack of median filtering leads to less touch precision.

Instead, a level shifting technique was used as presented in [42]. It utilizes two MOSFET transistors as shown in Fig. 6.11. In this circuit, a 2N7002 MOSFET is used. I^2C devices operating at 5 V are to be connected on the right side, devices operating at 3.3 V are to be connected on the left side. The connection is the same for the SCL as well as the SDA line.

This approach is possible because of the use of open collector outputs and pull-up resistors for I^2C as presented in Sect. 3.4.2.1. During idle state, no device is pulling down the line and the pull-up resistors on both sides pull the line up to the respective high level. The voltage between gate and source (V_{gs}) of the MOSFET is approximately zero (slightly higher because of the residual resistance of the open collector outputs). Therefore, the MOSFET channel is closed and no current is flowing. Different voltage levels at source and drain are possible.

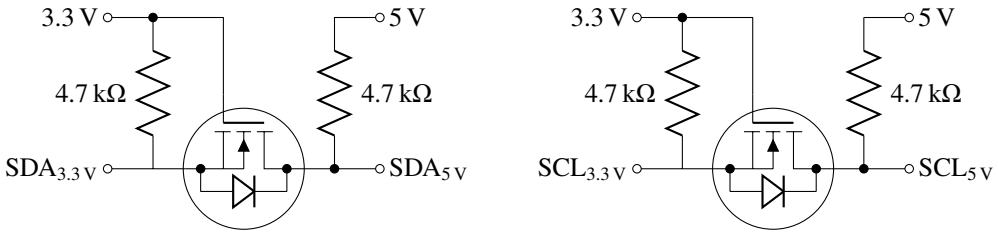


Figure 6.11: *I^2C level shifting technique*

When a device on the left side pulls down the line, V_{gs} rises to 3.3 V. The MOSFET starts conducting and current flows, pulling down the line on the right side. Both sides are now at low level as intended by the 3.3 V device. The same outcome results, when a device on the right side pulls down the line. The effect is a slightly different. Since, the Drain and the Source are connected via the body diode, current flows from the left to the right side when the voltage on the right side is smaller. This increases V_{gs} and finally the voltage is pulled down to low level on the left side, too.

Chapter 7

Evaluation

7.1 Software

7.1.1 Evaluation System

For evaluating the reaction of the system to various network conditions, an evaluation system as depicted in Fig. 7.1 is set up. A single Raspberry Pi is used as sender as well as receiver for eliminating the influence of non synchronized clocks. It sends the generated packets over the Internet to a server that sends the received packets back to the Raspberry Pi. The overall delay is measured by means of a sound generator, generating clicks in regular intervalls, and an oscilloscope.

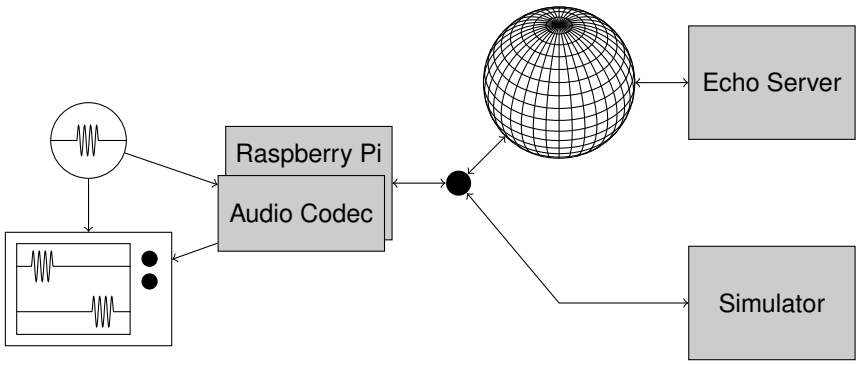


Figure 7.1: *Software evaluation system*

The echo server is located in a computing center in Nürnberg, approximately 450 km away from the Raspberry Pi in Hamburg. This connection is used for determining a typical network condition. Since the data has to travel back and forth before playing it back, this corresponds to a jam session with a musician 900 km away (e.g. Hamburg - Milan in Italy). It was observed that these conditions do not only change on a small scale (e.g. from packet to packet), but also on a larger scale (e.g. depending on the time of day). Therefore, a simulator is build to eliminate the large scale dependency for making the measurements more comparable among each other.

This simulator is a software running on a personal computer attached to the Raspberry Pi via a single network switch. It receives packets and sends them back after a delay determined by an adjustable random number generator. No packet is send back before the previous packet was send, because there was no packet reordering detected for the Nürnberg connection. The probability density function of the random number generator is a shifted gamma function as proposed in [15]. The probability density function is defined as

$$P(x|k,\theta,s) = \frac{1}{\Gamma(k)\theta^k}\,(x-s)^{k-1}\,e^{-\frac{x-s}{\theta}}. \tag{7.1}$$

The parameters k and θ determine the shape and scale of the distribution and s shifts the probability density function on the x-axis. Additionally, an adjustable amount of packets d are randomly dropped. This simulates packets that get lost for example because of a congested router.

7.1.2 Network Latency

As a first step, the network delay to Nürnberg is analyzed and the simulator is set up accordingly. For this purpose, each packet is equipped with a time stamp at the time of sending and reception. The difference between these two time stamps corresponds to the network delay. Fig. 7.2 shows the resulting delay distribution for a measurement of 25 seconds.

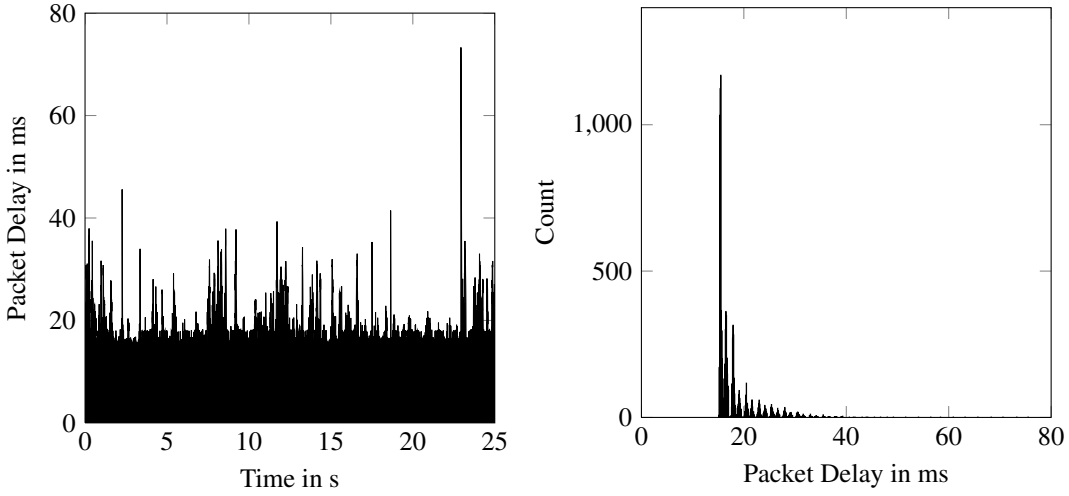


Figure 7.2: *Exemplary network delay distribution for connection to Nürnberg*

The average delay of this distribution is 18 ms with a standard deviation of 4 ms. In order to fit to this connection, the simulator was set up with a mean of 2 ms and a variance of 9.5 ms^2 for the unshifted gamma distribution and it is shifted by 14 ms. This results in the parameters $k = \frac{8}{19}$, $\theta = \frac{19}{4}$, and $s = 14$. A packet loss factor of $d = 0.098\%$ was chosen that matches the number of entirely lost packets in the measured connection.

The distribution of the delay with the simulator at these settings is shown in Fig. 7.3. The histogram shows a similar shape and the delay keeps within the same limits. Although, the real delay shows more burstiness. There are times where most packets have a higher delay (e.g. around 8 s), while there are times where the jitter is much smaller than in average (e.g.

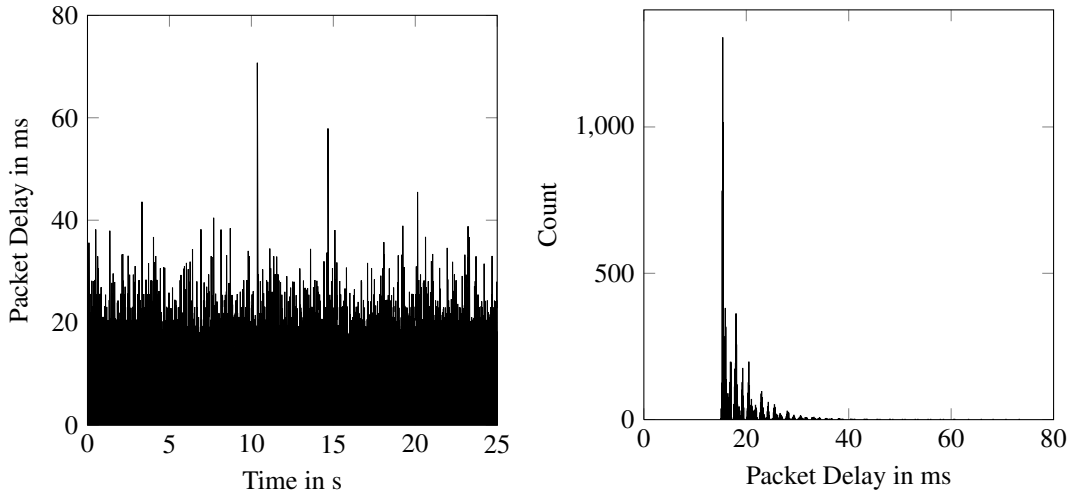


Figure 7.3: *Exemplary network delay distribution of the simulator*

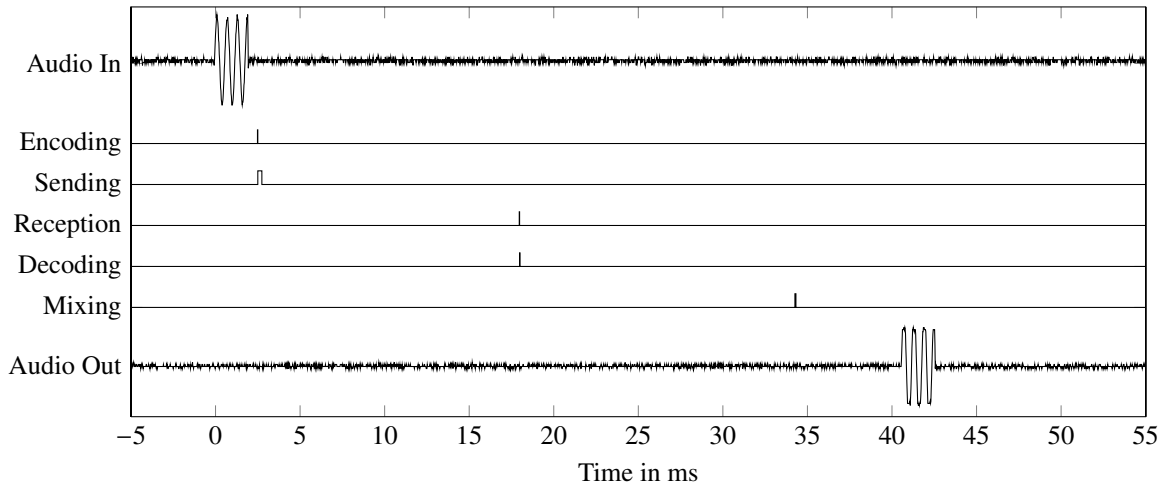
around 20 s). The latter case has no audible impact (besides a possible imprecise adjustment), but the first case might lead to packet loss bursts where several consecutive packets are lost, leading to a longer pause in audio data. It induces additional difficulties for error concealment, because more data has to be predicted at once. This effect is not considered in the simulation so far.

7.1.3 Journey of an Impulse

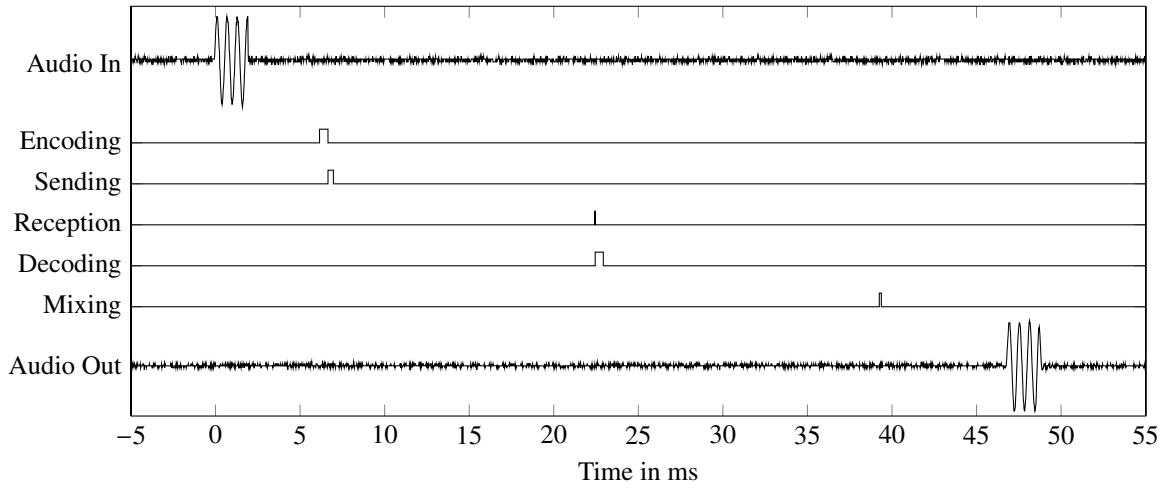
For measuring the influcence of the network conditions and software parameters on the overall audio latency, the time shift of the audio data was analyzed by means of an oscilloscope. As an audio signal a short sine burst was used, because it can be easily recognized and measured in time domain and (in contrast to a DC pulse) is not affected by AC coupling of the audio hardware. Additionally, the software marks a packet, when a certain level is exceeded for indicating the sine burst packet. This makes it possible to recognize when the burst is processed in a software module. Thereupon, a GPIO pin is set to high and back to low after the processing.

The results are visualized in Fig. 7.4. In the example without coding, the overall latency is slightly more than 40 ms. The network delay (the time from sending until reception) is about 15 ms and the packet is delayed about 16 further milliseconds before it is mixed and send to the audio hardware. The times before encoding and after mixing depend on the block size and the relative position of the first burst slope to the beginning of the block. Therefore, for this experimental setup, there is an inherent timing variance in the magnitude of a block length. This implies that the exact time duration of the recording or playback process can not be measured. Although, the time between mixing and playback is distinctly longer than the time of recording, because of the safety margin (cf. Sect. 4.1.2). By further improving the real time behaviour of the system, this time duration might be reduced.

In order to reduce the audio data rate, the Opus coding algorithm can be enabled. This reduces the bit rate from 768 $\frac{\text{kbit}}{\text{s}}$ down to 72 $\frac{\text{kbit}}{\text{s}}$. For the measurement with enabled Opus coding, the overall latency increased to 47 ms. This is partly induced by the additional time needed for error concealment (about 3 ms). This implicitly increases the target queue length, because



7.4.1: Without coding algorithm, $\beta = 3$, $N_F = 120$



7.4.2: With Opus coding algorithm, $\beta = 3$, $N_F = 240$

Figure 7.4: *Input and output signal and digital signals indicating the current action*

the concealment decreases the mean time until processing. Furthermore, the CPU utilization considerably increases because of the error concealment and the Opus coding algorithm. The last fact is reflected in the broader impulses for encoding and decoding. For a single packet, the overall amount of computation seems to be small, compared to the overall latency, but it has to be beared in mind, that there are many transmissions running in parallel (in intervals determined by N_F) and other software components (e.g. the kernel) have to find time for computation, too. Therefore, it is necessary to increase N_F in order to reduce the CPU utilization. This depencency will be further investigated in the next section.

7.1.4 Influences on Overall Latency

For the following figures, the previously described experiment was repeated with various settings. The default setting is $\beta = 3$ and $N_F = 120$. While using the Opus coding algorithm, $N_F = 240$ is chosen. The simulator settings specified above are used for measuring the overall latency. Furthermore, the percentage of concealed packet was measured over a time span of

one minute. For estimating the CPU utilization, the Linux program "top" was used.

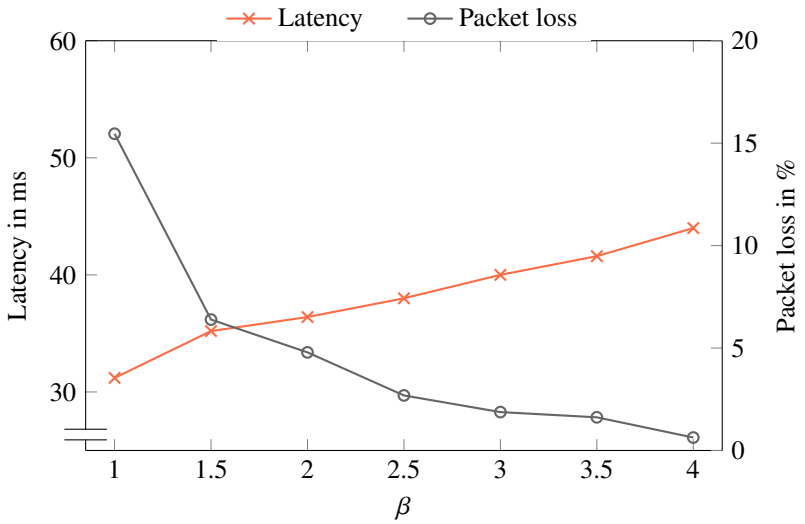


Figure 7.5: *Latency and packet loss against packet loss tuning factor without coding algorithm*

The first Fig. 7.5 shows the influence of the packet loss tuning factor β on the overall latency and the packet loss. A higher β increases the target queue length and therefore the overall latency. On the other hand, the packet loss decreases because more network jitter is tolerated. A packet loss of 2 % was selected as appropriate threshold according to the evaluation in [22]. Therefore, $\beta = 3$ was chosen as optimal parameter.

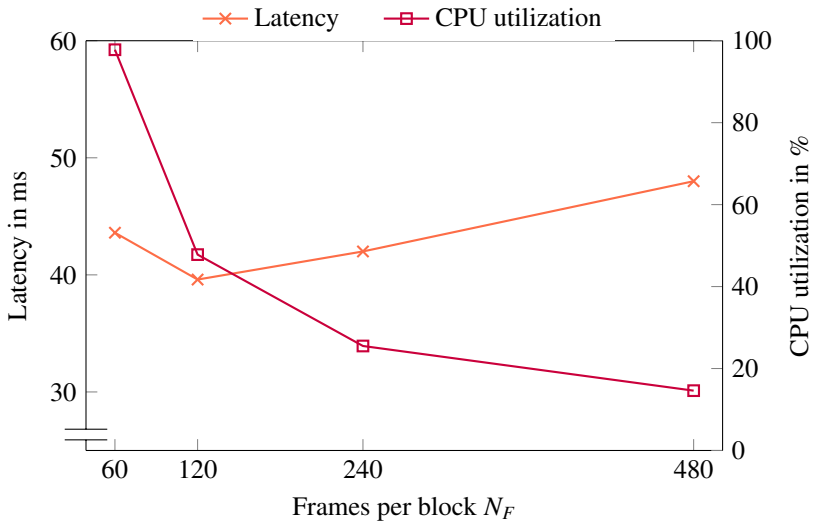


Figure 7.6: *Latency and CPU utilization against frames per block without coding algorithm*

A further setting that influences the latency is the number of frames per block N_F as presented in Fig. 7.6. All in all, less frames per block induce a lower blocking delay and therefore a lower overall delay. On the other hand, the CPU utilization increases, because more frames have to be handled within the same time span. Finally, at $N_F = 60$, the CPU utilization leads to congestion of the system. Therefore, packets might not be handled in time and this induces higher jitter, finally leading to a higher overall latency again. $N_F = 120$ was chosen for minimal latency at acceptable CPU load.

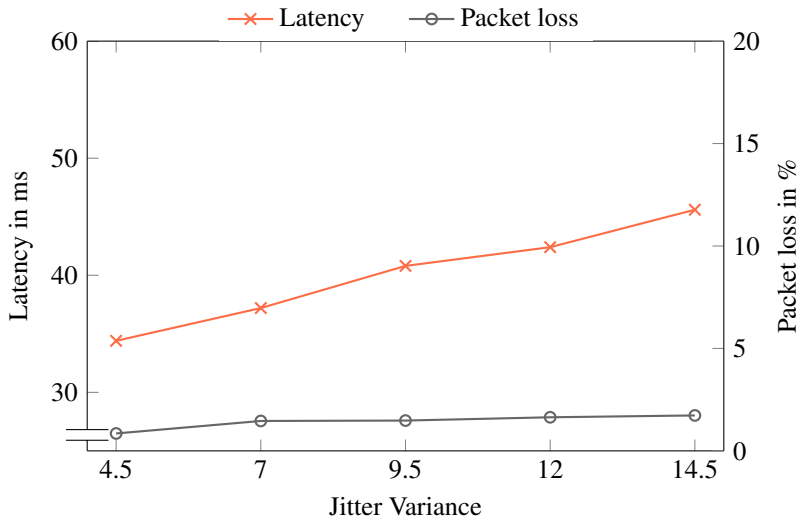


Figure 7.7: *Latency and packet loss against simulated jitter variance without coding algorithm*

After determining the optimal settings for the original network scenario, the influence of other network scenarios was investigated. Fig. 7.7 shows the effect of various variance settings of the simulator. As intended by the algorithm, the packet loss stays nearly constant while the system adapts itself to the modified conditions by increasing the overall latency.

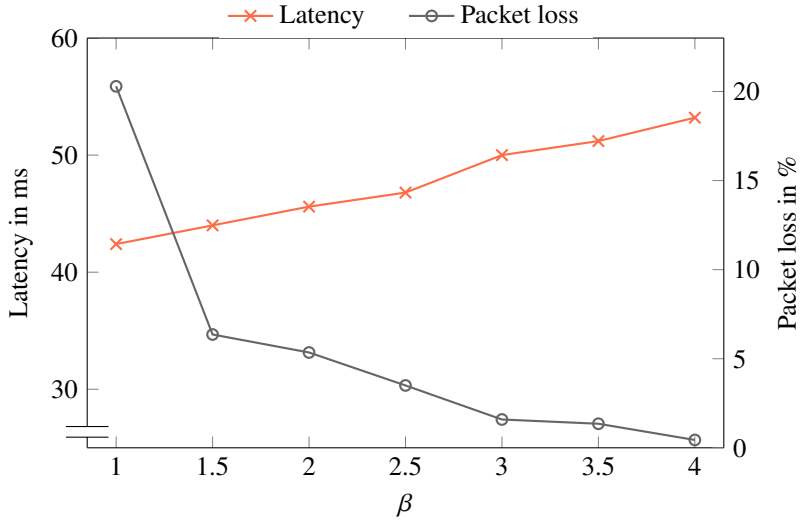


Figure 7.8: *Latency and packet loss against packet loss tuning factor with Opus*

By enabling the Opus coding, the overall latency increases as presented in the previous section. This is observable in Fig. 7.8, too. However, the relation between the packet loss tuning factor and the packet loss does not change significantly. The same $\beta = 3$ was chosen.

Fig. 7.9 confirms the assumption of an increased CPU utilization. While, for $N_F = 120$, the CPU utilization is about 50 % for the non-coding case, it leads to a highly unstable system at full load for the Opus case. Therefore, $N_F = 240$ was chosen, reducing the CPU utilization to a tolerable magnitude.

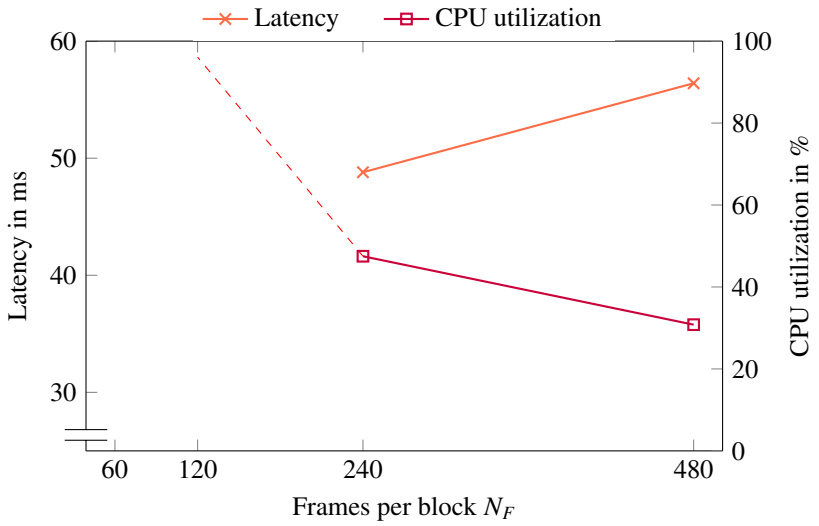


Figure 7.9: *Latency and CPU utilization against frames per block with Opus*

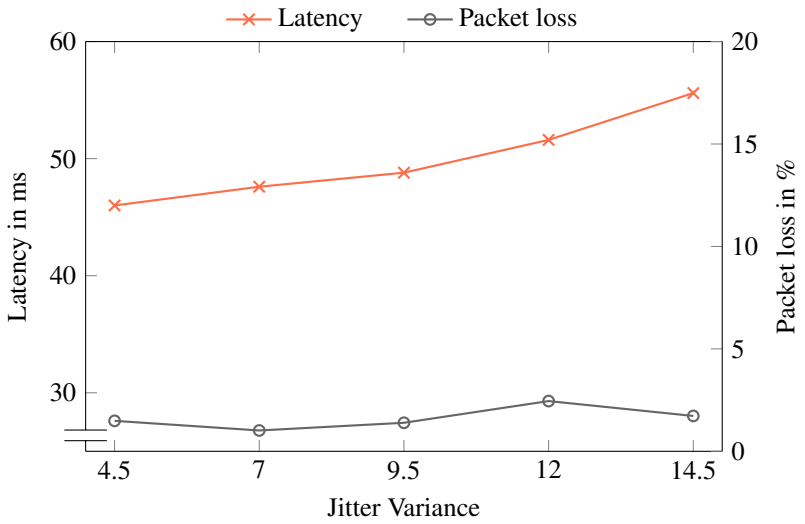


Figure 7.10: *Latency and CPU utilization against simulated jitter variance with Opus*

Finally, the effect of modified jitter was investigated in Fig. 7.10. The overall latency increases with the jitter and the packet loss stays low. Although, the system does not guarantee the exact compliance to the 2 % mark, because it is an open loop system.

7.2 Hardware

In order to evaluate the audio quality of the developed hardware, several measurements are conducted with an UPV Audio Analyzer by Rohde & Schwarz. When not stated otherwise, the default settings were used. The voltages in this section are root mean square (RMS) voltages. The measurements follow the same principle: Audio is generated and fed into the component. The output is measured and metrics can derived by comparing input and output signal assuming the input signal is ideal. The following metrics are investigated:

Total Harmonic Distortion (THD) describes the amount of distortions induced by non-linear behaviour. It is calculated by dividing the power of the harmonics of a signal by the power of the source signal.

Signal to Noise Ratio (SNR) is a measure that describes the noise power relative to the signal. Noise might be induced for example by digital components or power supplies. The upper bound of the SNR is determined by the resolution of the ADC.

Frequency Response describes the amplification or attenuation of a signal as a function of the frequency. Audio devices might damp higher or lower frequencies (for example, because they are AC coupled).

7.2.1 Audio Output

For the evaluation of the audio output, it was necessary to provide audio in digital domain. Therefore, sound files with a 1 kHz sine wave were generated at different levels. Since they are digital signals, the level is measured in relation to full scale output (all bits high). This equates to 0 dBFS. The same measurements were taken for the codec board as well as for the existing audio output of the Raspberry Pi. In Fig. 7.11 the THD values of the Raspberry Pi PWM module and the codec board are compared at various levels. The best THD that can be reached is -88 dB at -6 dBFS for the codec board. For louder sounds, some more harmonics arise. Although, even then it is much better than values for the existing audio output.

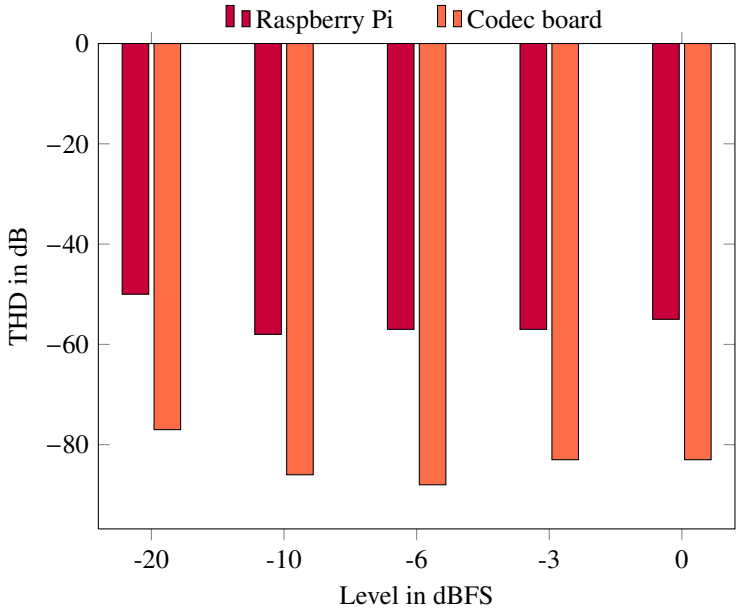
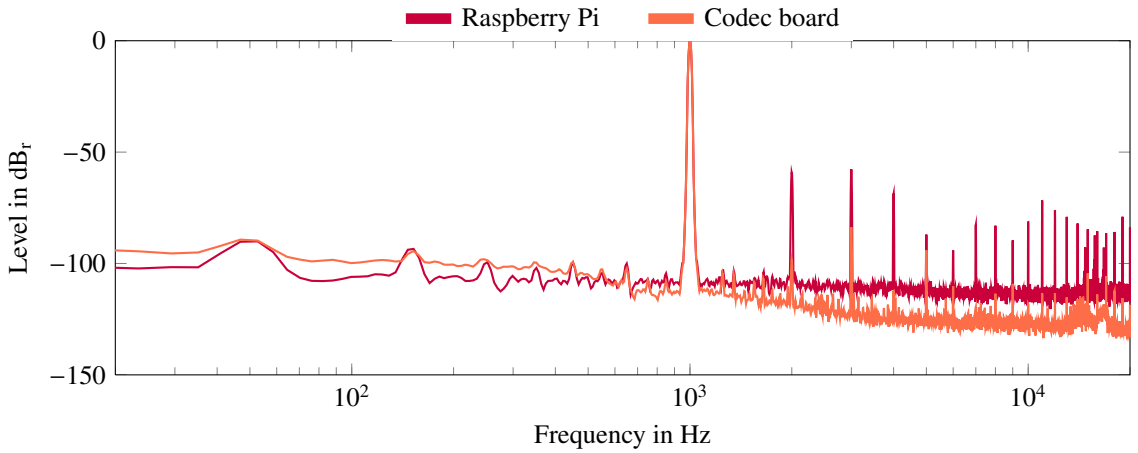
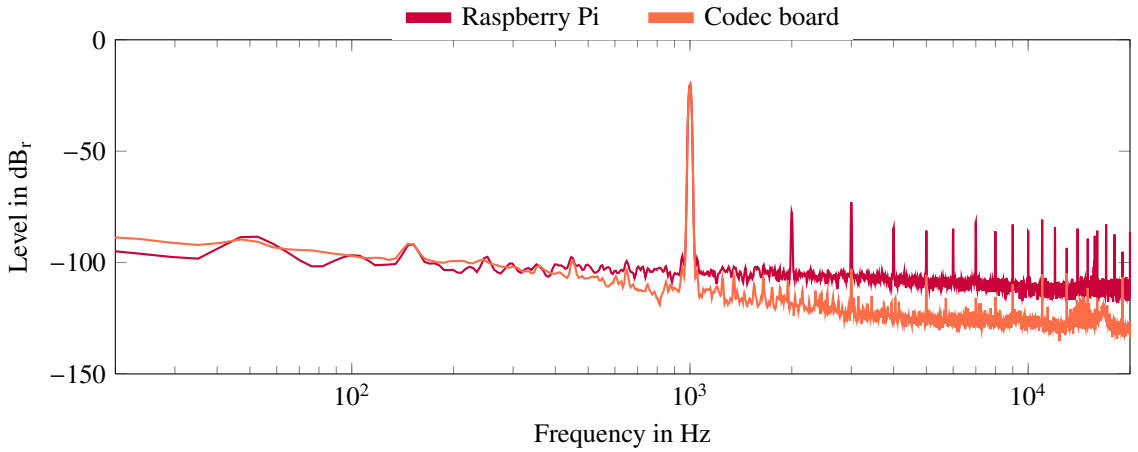


Figure 7.11: *THD comparison of Raspberry Pi and codec board*

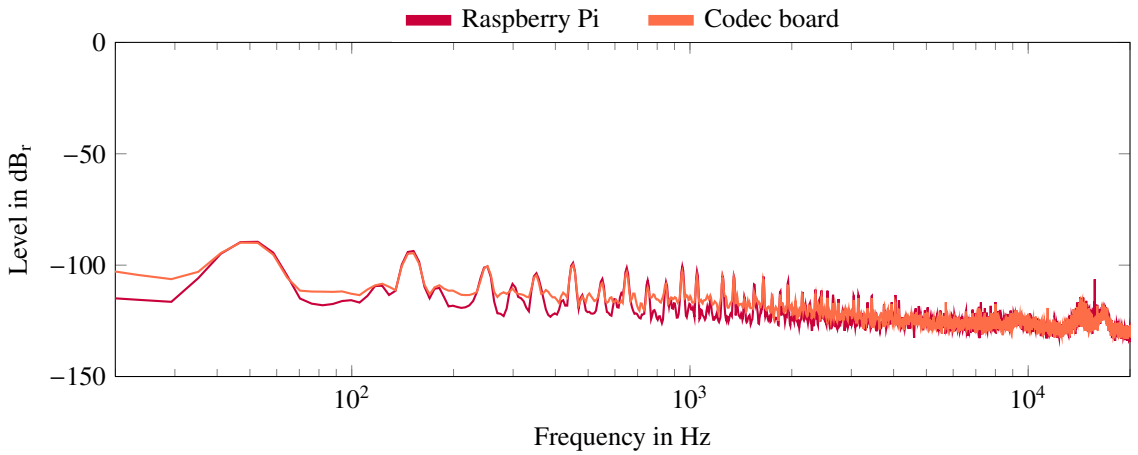
For a better insight in these characteristics, Fig. 7.12 depicts the resulting frequency spectra. The output level is measured relative to the maximum output (i.e. 0 dBr = 776 mV = -2.2 dBV). Apparently, the harmonics are much smaller for the codec board. Notably, the noise floor of the codec board is lower for higher frequencies but higher for low frequencies. It is important to note that the ALSA volume slider of the Raspberry Pi PWM module supports higher values than 0 dB (up to 4 dB). The measurements are referred to the 0 dB setting. For a higher setting, the measurements get even worse because clipping takes place.



7.12.1: For 0 dBFS



7.12.2: For −20 dBFS



7.12.3: For silence

Figure 7.12: *Frequency spectrum of existing audio output of the Raspberry Pi while playing back a* 1 kHz *sine wave*

7.2.2 Audio Input

For the audio input, the other direction was taken. Analog signals were played back and recorded as digital files. These files were analyzed afterwards. Since the audio output is designed for a maximum input level of 1.55 V, a 1 kHz sine wave with this level was played back. It results in a digital signal of 0 dBFS. The frequency spectrum is depicted in Fig. 7.13. The corresponding THD is −91 dB. For calculating the SNR, some silence was recorded and analyzed. The measured level is −93 dBFS corresponding to a SNR of 93 dB for a full scale signal. It is 5 dB smaller than the theoretical quantization noise for 16 bit of 98 dB.

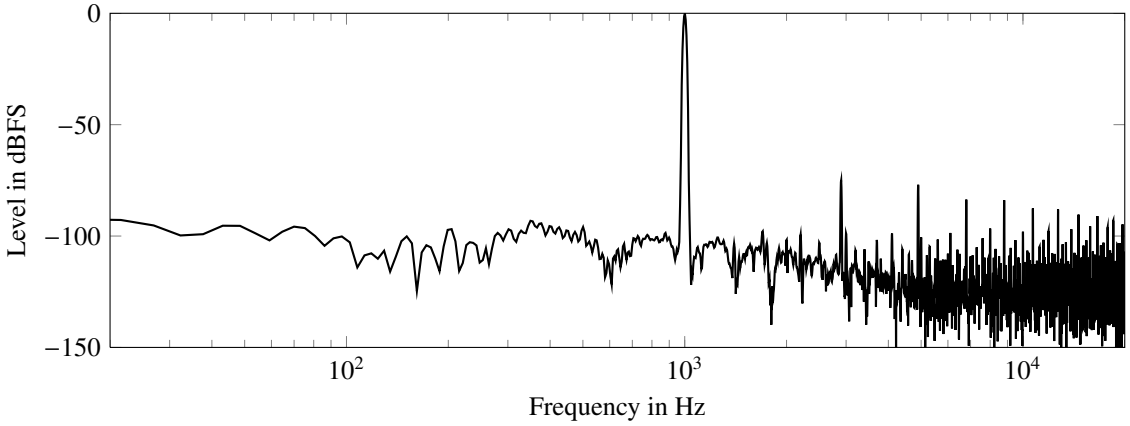


Figure 7.13: *Input frequency spectrum of codec board with* 1 kHz *sine wave input*

7.2.3 Headphone Amplifier

The measurements for the headphone amplifier were conducted completely in the analog domain. For those, the codec board was disconnected from the amplifier board and the analog sound was fed directly into the D-Sub connector of the amplifier board. The first measurement in Fig. 7.14 verifies the operation of the amplifier by showing a higher output than input level. For example, an input signal of −40 dBV results in an output signal of −24 dBV resulting in an amplification of 16 dB. Apparently, the targeted amplification of −20 dB is not quite reached. Although, listening tests have shown, that it is fairly loud for studio headphones and far too loud for earplug headphones. Another finding emphazies the fact that the resistors for setting the gain were choosen too high: The amplifier does not produce levels higher than 13 dB and is therefore overdriven at an input level above −7 dBV. Unfortunately, the audio codec delivers levels up to −2.2 dBV. This limit is even more obvious in Fig. 7.16 where the THD is plotted against the input voltage. At the point where the amplifier is overdriven, the signal is clipped and the harmonic distortions are much higher. Although, when the level is slightly reduced to −6 dBFS, the audio quality is quite good with a THD of −84 dB and a SNR of 82 dB.

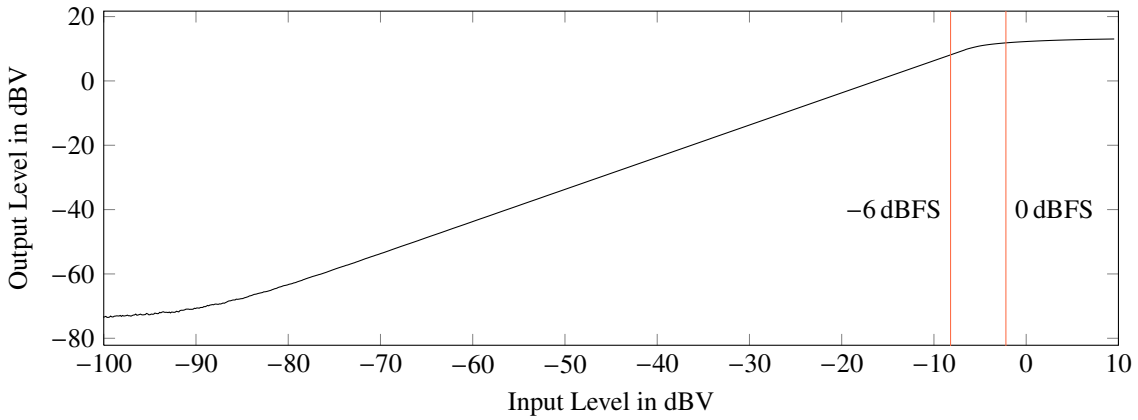


Figure 7.14: *Output level of headphone amplifier plotted against input level*

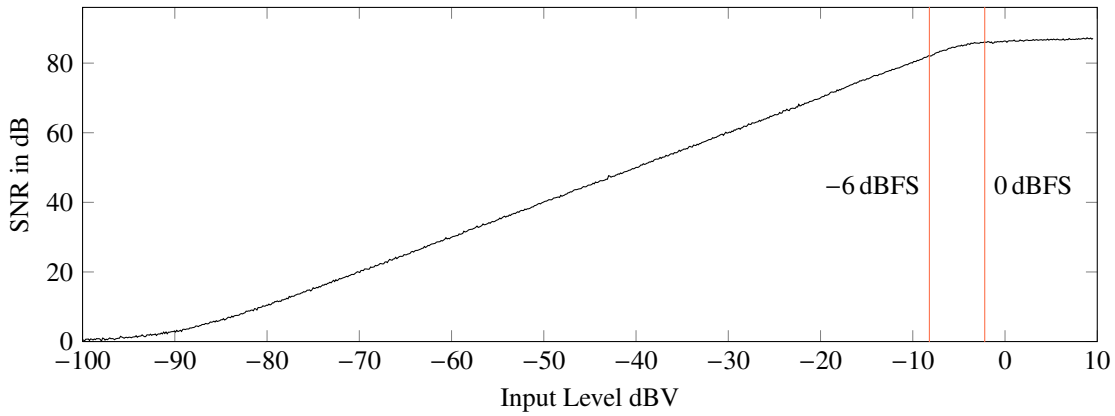


Figure 7.15: *SNR of headphone amplifier plotted against input level*

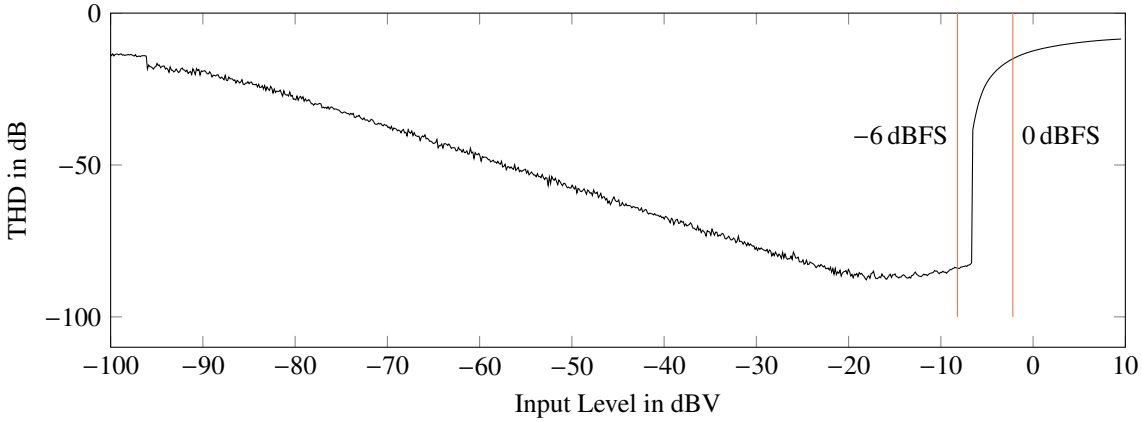


Figure 7.16: *THD of headphone amplifier plotted against input level*

7.2.4 Input Amplifier

Since the input amplifier should support a wide range of audio equipment, a high maximum gain is required, while lower gain settings should be suitable, too. The measurement was counducted like the one for the headphone amplifier. Since multiple gain settings are possible, the measurement was repeated for gains of 68 dB, 51 dB, 34 dB, 17 dB, and 0 dB. The various output levels are plotted against the input level in Fig. 7.17. Of course, the amplification increases, but noise is amplified, too. This is visible by looking at lower input levels: Except for the 0 dB case, the curves start to rise not before approximately −80 dBV. This threshold is even more apparent in Fig. 7.18 that shows the SNR and in Fig. 7.19 that shows the THD. For better comparison of the gain settings, Table 7.1 lists the gain settings and the corresponding input level that is required for full scale input. For example, for a guitar that is directly connected to the amplifier, a gain of about 17 dB would be choosen, because it provides a level of about 200 mV. Therefore, the SNR is about 66 dB and the THD is about −81 dB for this example.

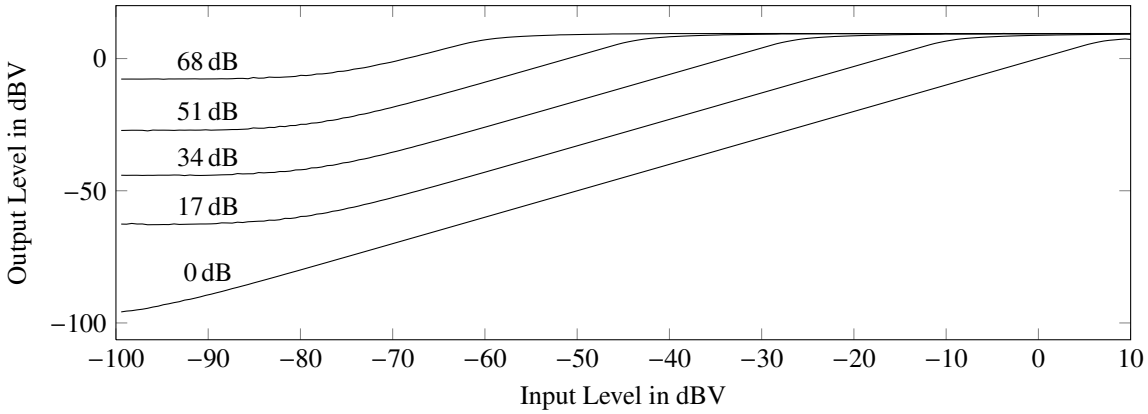


Figure 7.17: *Output level of input amplifier plotted against input level*

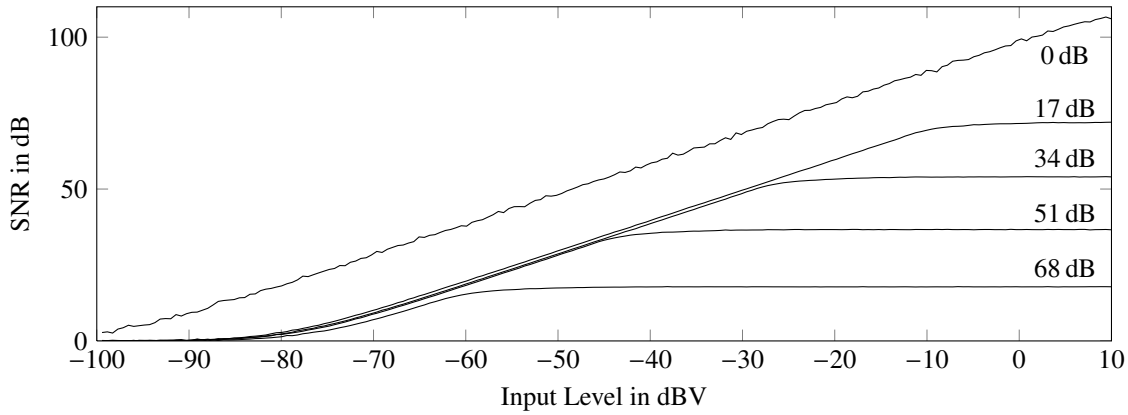


Figure 7.18: *SNR of input amplifier plotted against input level*

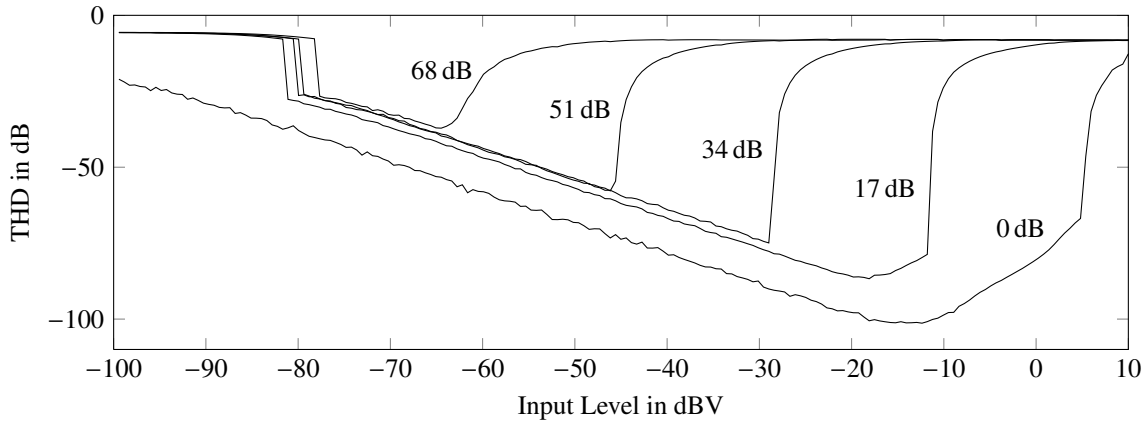


Figure 7.19: *THD of input amplifier plotted against input level*

Gain	Input Level for 0 dBFS		SNR	THD
	Logarithmic	Linear		
0 dB	3.8 dBV	1.55 V	102 dB	−70 dB
17 dB	−13.2 dBV	219 mV	66 dB	−81 dB
34 dB	−30.2 dBV	30.9 mV	48 dB	−74 dB
51 dB	−47.2 dBV	4.37 mV	31 dB	−57 dB
68 dB	−64.2 dBV	617 µV	12 dB	−37 dB

Table 7.1: *Comparison of SNR and THD at different gain settings for full scale output.*

7.3 Frequency Response

The frequency responses are measured by playing back a sine wave with a fixed level, but variable frequency and plotting the output level in relation to the input level. For the headphone amplifier, an input level of −8.12 dBV was choosen to not overdrive the amplifier. The plot in Fig. 7.21 shows a high pass behaviour with a cut-off frequency of 135 Hz. Smaller output capacitors could be choosen for supporting lower frequencies.

For the audio codec the input level is 1.55 V = 3.8 dB. In total, the system attenuates the signal by −6 dB, because the full scale output is 776 mV. The input amplifier was driven with the settings from Table 7.1 and these measurements result in the frequency responses in Fig. 7.22. No significant frequency dependent attenuation takes place for the audio codec as well as for the input amplifier.

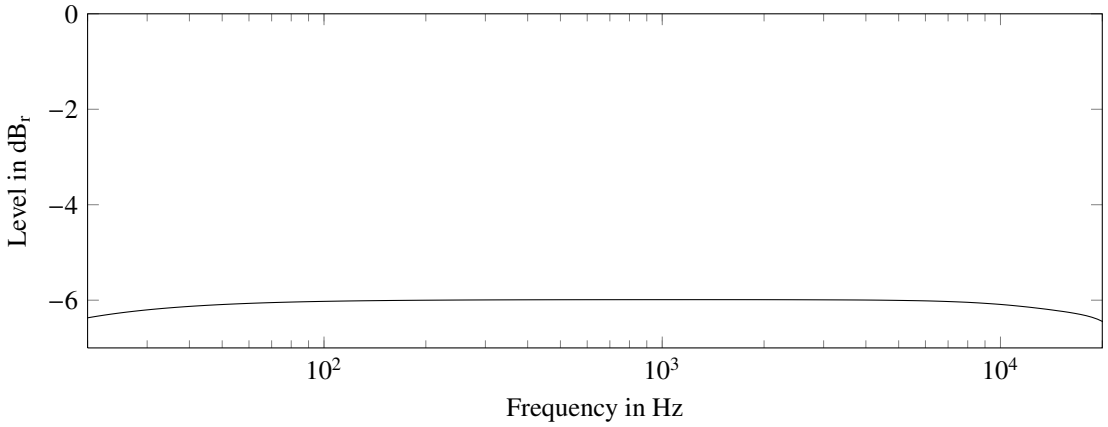


Figure 7.20: *Frequency response of audio codec board for* 3.8 dBV

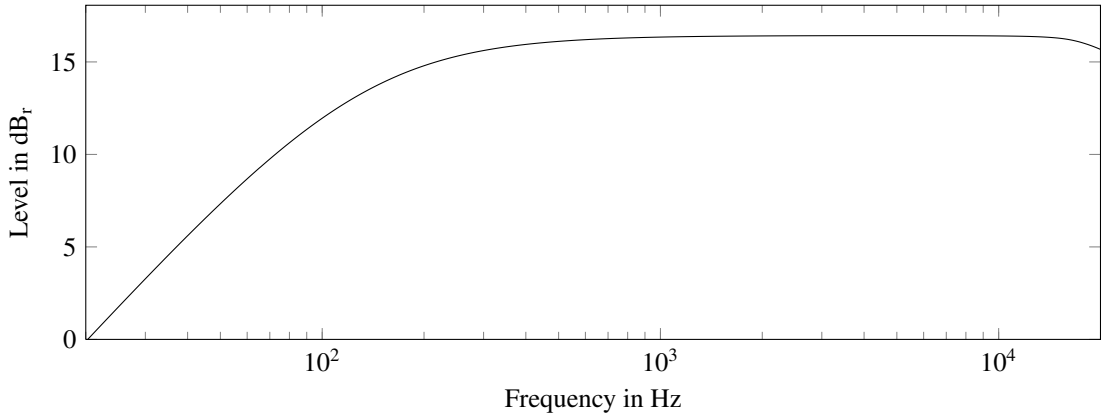


Figure 7.21: *Frequency response of headphone amplifier for* −8.12 dBV

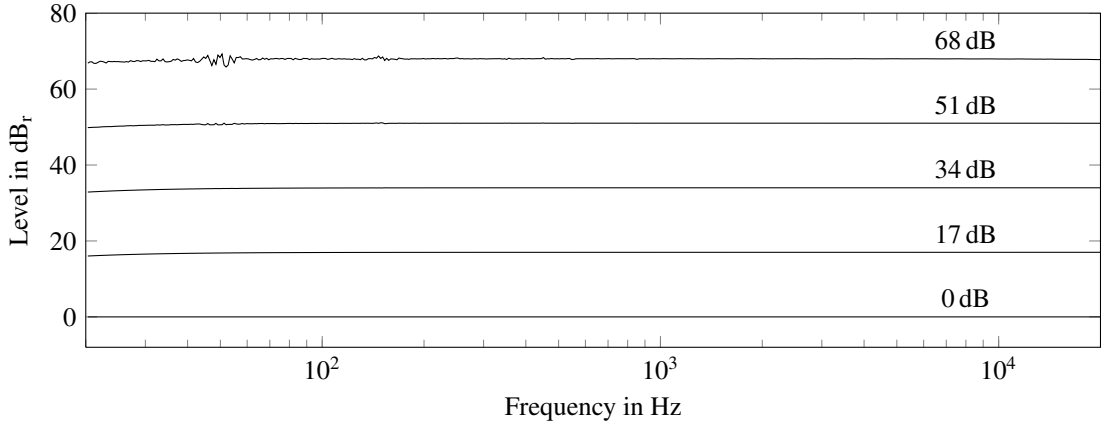


Figure 7.22: *Frequency response of input amplifier*

Chapter 8

Conclusion

Networked music performances provide great potentials for uniting musicians over large distances. Furthermore, research in this direction enables other use cases of high quality and low latency Audio over IP such as medical remote assistance. Yet, the inherent characteristics of the Internet induce some important issues. First of all, the transmission delay over the Internet is large and has a high jitter. Secondly, packets might get lost. And last but not least, uncompressed high quality audio transmission requires a high network bandwidth.

These issues have to be faced when developing a system for networked music performances. Of particular importance for musical interaction is the amount of end to end latency. When the audio of the other musicians is delayed too much relative to the own play, it is increasingly difficult to play in synchronity. Values above 60 ms are hard to compensate, but even for lower delay, it requires some practice to get used to it.

In this thesis, an embedded system providing support for networked music performances was developed. Transmission of uncompressed audio is supported as well as it utilizes the Opus coding algorithm for reducing the data rate by a factor of about 10. In order to compensate the network delay, a queue is used for buffering the incoming audio data. In contrast to a fixed queue length as used in many systems, the queue length is dynamically adapted to the measured network jitter. Since complete compensation of the jitter is not feasible, a certain amount of lost packets is accepted and added to the number of lost packets that is lost in the Internet anyway. The audible effect of the lost packets is reduced by a technique called error concealment that calculates replacement data.

This is provided by a software that is integrated in an embedded system based on a Raspberry Pi. Several additions were added to this ARM based single-board computer for building up the complete system. Since the Raspberry Pi does not provide a proper analog audio interface, an extension board featuring a CS4270 audio codec was build. The communication to the Raspberry Pi takes place via an I^2S connection. For using this connection it was required to write a Linux kernel driver that supports the I^2S peripheral of the Raspberry Pi.

The embedded system was further enhanced by the development of an amplifier board that supports a wide range of instruments with an adjustable gain of at most 68 dB. Furthermore, a headphone amplifier is integrated as well as support for line-level signals. For waiving the need of further external components, a touchscreen is integrated for providing a user interface.

For interfacing the touchscreen via the HDMI interface of the Raspberry Pi yet another board is build. The user interface can also be accessed by an external web browser via a WebSocket connection. Altogether, all components needed for enabling a networked music performance are integrated into a single device.

The assessment of the audio components shows a good overall audio performance. Although, a weakness in the gain setting of the headphone amplifier was discovered that slightly reduces the range of output levels. For evaluating the Audio over IP software, a typical Internet connection over a total line-of-sight distance of about 900 km was analyzed. A simulator was build that complies to the discovered parameters. It is shown that the system achieves an overall latency of about 40 ms to 50 ms for packet loss rates of about 2 %. This amount of lost packets can be handled quite good by state of the art error concealment algorithms, but the computational effort is quite high. The results show that an adaptive adjustment of the queue length is beneficial and performs as intended. A open-loop control as used in this thesis is useful, but requires hand tuning of parameters and is therefore not universally applicable when conditions change (for example a new error concealment is used).

This might be improved by using a closed-loop control with the packet loss as reference. A much simpler approach would be to outsource the control to the user. Starting from an initial value that is determined by the presented algorithm, the musician gets a slider that provides the possibility to balance the perceivable latency against the audio quality induced by the packet loss. This has the additional advantage, that the effect of the latency as well as the audio quality of the error concealment distinctly depends on the type of music.

Further work should be done to improve the computational complexity of the error concealment and the coding algorithm to make it more suitable for the low computational power of the embedded system. The evaluation shows, that this would permit shorter block lengths and finally lower latency. It is also sensible to evaluate the error concealment with respect to packet loss burst since the evaluation of the network conditions show a highly variable nature than can not be properly described by a single packet loss factor.

The software system should be further optimized in terms of real time behaviour in order to lower the times reserved for processing and finally achieving lower overall latency. Using a realtime kernel should improve the behaviour, although a first test has shown no significant improvements. Further investigations are needed for precise identification of disturbing activities.

Regarding the device itself, some adjustments should be done in order to reach even better audio quality. The concrete actions should include using a smaller gain for the headphone amplifier as well as optimizing the output stage of the headphone amplifier with respect to low frequencies. Furthermore, a chassis should be build in order to be prepared for rugged stage applications.

Bibliography

[1] D. Akoumianakis and C. Alexandraki, "Understanding networked music communities through a practice-lens: Virtual 'Tells' and cultural artifacts," in *Proceedings of the International Conference on Intelligent Networking and Collaborative Systems (INCOS'10)*, Thessaloniki, Greece, Nov. 2010.

[2] C. Alexandraki and D. Akoumianakis, "Exploring new perspectives in network music performance: The DIAMOUSES framework," *Computer Music Journal*, vol. 34, no. 2, Jun. 2010.

[3] C. Chafe, J.-P. Caceres, and M. Gurevich, "Effect of temporal separation on synchronization in rhythmic performance," *Perception*, vol. 39, no. 7, Apr. 2010.

[4] E. Chew, R. Zimmermann, A. A. Sawchuk, C. Papadopoulos, C. Kyriakakis, C. Tanoue, D. Desai, M. Pawar, R. Sinha, and W. Meyer, "A second report on the user experiments in the distributed immersive performance project," in *Proceedings of the Open Workshop of the MUSICNETWORK*, Vienna, Austria, Jul. 2005.

[5] A. Carôt, C. Werner, and T. Fischinger, "Towards a comprehensive cognitive analysis of delay influenced rhythmical interaction," in *Proceedings of the International Computer Music Conference (ICMC'09)*, Montreal, Quebec, Canada, Aug. 2009.

[6] J.-P. Cáceres and A. B. Renaud, "Playing the network: the use of time delays as musical devices," in *Proceedings of the International Computer Music Conference (ICMC'08)*, Belfast, North Ireland, Aug. 2008.

[7] J.-P. Cáceres, R. Hamilton, D. Iyer, C. Chafe, and G. Wang, "To the edge with china: Explorations in network performance," in *Proceedings of the International Conference on Digital Arts (ARTECH'08)*, Porto, Portugal, Nov. 2008.

[8] C. Chafe, S. Wilson, A. Leistikow, D. Chisholm, and G. Scavone, "A simplified approach to high quality music and sound over IP," in *Proceedings of the COST-G6 Conference on Digital Audio Effects (DAFx-00)*, Verona, Italy, Dec. 2000.

[9] J.-P. Cáceres and C. Chafe, "JackTrip: Under the hood of an engine for network audio," *Journal of New Music Research*, vol. 39, no. 3, Nov. 2010.

[10] G. Hajdu, "Quintet.net - A quintet on the internet," in *Proceedings of the International Computer Music Conference (ICMC'03)*, Singapore, Oct. 2003.

[11] C. Alexandraki, P. Koutlemanis, P. Gasteratos, N. Valsamakis, D. Akoumianakis, G. Milolidakis, G. Vellis, and D. Kotsalis, "Towards the implementation of a generic platform for networked music performance: The DIAMOUSES approach," in *Proceedings of the International Computer Music Conference (ICMC'08)*, Belfast, North Ireland, Aug. 2008.

[12] A. Carôt, U. Krämer, and G. Schuller, "Network music performance (NMP) in narrow band networks," in *Proceedings of the AES Convention*, Paris, France, May 2006.

[13] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. Prentice Hall, 2011.

[14] V. Paxson and S. Floyd, "Wide area traffic: The failure of Poisson modeling," *IEEE/ACM Transactions on Networking (ToN)*, vol. 3, no. 3, Jun. 1995.

[15] A. Corlett, D. Pullin, and S. Sargood, "Statistics of one-way internet packet delays," in *Proceedings of the Internet Engineering Task Force (IETF'02)*, Minneapolis, Minnesota USA, Mar. 2002.

[16] "Meeting report of the IP performance metrics group," in *Proceedings of the Internet Engineering Task Force (IETF'02)*, Minneapolis, Minnesota USA, Mar. 2002.

[17] C. Demichelis and P. Chimento, "IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)," RFC 3393 (Proposed Standard), Internet Engineering Task Force, Nov. 2002.

[18] J. Coalson. (2013, Jan.) flac hompage: Comparison. Xiph.Org Foundation. [Online]. Available: https://xiph.org/flac/comparison.html

[19] "Subjective listening tests on low-bitrate audio codecs," EBU Project Group B/AIM (Audio in Multimedia), Tech. Rep. EBU Tech 3296-2003, Jun. 2003.

[20] J.-M. Valin, T. Terriberry, C. Montgomery, and G. Maxwell, "A high-quality speech and audio codec with less than 10-ms delay," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 18, no. 1, Jan. 2010.

[21] J. Valin, K. Vos, and T. Terriberry, "Definition of the Opus Audio Codec," RFC 6716 (Proposed Standard), Internet Engineering Task Force, Sep. 2012.

[22] M. Fink, M. Holters, and U. Zölzer, "Comparison of various predictors for audio extrapolation," in *Proceedings of the International Conference on Digital Audio Effects (DAFx'13)*, Maynooth, Ireland, Sep. 2013.

[23] M. Barr, *Programming Embedded Systems in C and C++*. O'Reilly Media, Inc., 1999.

[24] "I^2C-bus specification and user manual," NXP Semiconductors, User manual UM10204, Oct. 2012.

[25] "I^2S bus specification," Philips Semiconductors, Specification, Jun. 1996.

[26] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. O'Reilly Media, Inc., 2005.

[27] "AMBA AXI and ACE Protocol Specification," ARM Limited, Specification, Feb. 2013.

[28] R. Lorriaux, "Real-time audio on embedded linux," in *Proceedings of the Embedded Linux Conference (ELC'11)*, San Francisco, CA, USA, Apr. 2011.

[29] J. Kysela. (1998, Feb.) The Linux Ultra Sound Project. [Online]. Available: http://internet.perex.cz/~perex/ultra/index.html

[30] ALSA project. (2013, Jul.) Main Page News. [Online]. Available: http://www.alsa-project.org/main/index.php/Main_Page_News

[31] ALSA project. The C library reference. [Online]. Available: http://www.alsa-project.org/alsa-doc/alsa-lib

[32] A. Carôt and C. Werner, "External latency-optimized soundcard synchronization for applications in wide-area networks," in *Proceedings of the AES Regional Convention*, Tokio, Japan, Jul. 2009.

[33] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Transactions on Networking (ToN)*, vol. 2, no. 1, Feb. 1994.

[34] "WAMP - the WebSocket application messaging protocol," Tavendo GmbH, Specification. [Online]. Available: http://wamp.ws/spec

[35] AutobahnJS library. Tavendo GmbH. [Online]. Available: http://autobahn.ws/js

[36] WebSocket++. Zaphoyd Studios. [Online]. Available: http://www.zaphoyd.com/websocketpp

[37] B. Lepilleur. JsonCpp - JSON data format manipulation library. [Online]. Available: http://jsoncpp.sourceforge.net

[38] "BCM2835 ARM Peripherals," Broadcom Corporation, Datasheet, 2012.

[39] P. S. Gaskell, N. J. R. King, E. Breakenridge, and M. J. Ball, "Fractional-n frequency synthesizer using delta-sigma modulator in divider control circuit," U.S. Patent 5 079 521, Jan., 1992.

[40] M. Kozak and I. Kale, "A pipelined noise shaping coder for fractional-N frequency synthesis," *IEEE Transactions on Instrumentation and Measurement (TIM)*, vol. 50, no. 5, Oct. 2001.

[41] T. Harbaum. DVI2PAR. [Online]. Available: http://harbaum.org/till/dvi2par/index.shtml

[42] "Level shifting techniques in i^2c-bus design," NXP Semiconductors, Application note AN10441, Jun. 2007.

Mathematical Proofs

Proof of Intentional Packet Delay

Proof that it is possible to generate a continuous stream of data from a packet stream afflicted with bounded jitter by delaying the first packet and continue with regular time steps. Given the time sequence of packet generation $(s_n, s_{n+1}, \ldots)$ that follows

$$s_{n+1} = s_n + \frac{N_F}{f_s}. \tag{8.1}$$

The packet generated at s_n arrives at

$$s_n + \delta_n \tag{8.2}$$

$$\delta_{min} \leq \delta_n \leq \delta_{max}. \tag{8.3}$$

The first packet is intentionally delayed by δ_{max} - δ_{min} and processed at

$$r_0 = s_0 + \delta_0 + \delta_{max} - \delta_{min}. \tag{8.4}$$

Subsequent packets are are processed in a regular sequence $(r_n, r_{n+1}, \ldots)$ that follows

$$r_{n+1} = r_n + \frac{N_F}{f_s}. \tag{8.5}$$

Show that the processing never takes place before the packet arrives, i.e. for all n holds

$$r_n \geq s_n + \delta_n. \tag{8.6}$$

Step 1: Proof by induction

$$r_n = s_n + \delta_0 + \delta_{max} - \delta_{min}. \tag{8.7}$$

Basis: (8.4)

$$r_0 = s_0 + \delta_0 + \delta_{max} - \delta_{min}. \tag{8.8}$$

Inductive Step: Given induction hypothesis

$$r_n = s_n + \delta_0 + \delta_{max} - \delta_{min} \tag{8.9}$$

holds. Then follows from (8.5)

$$\begin{aligned} r_{n+1} &= s_n + \delta_0 + \delta_{max} - \delta_{min} + \frac{N_F}{f_s} && (8.10)\\ &= s_{n+1} - \frac{N_F}{f_s} + \delta_0 + \delta_{max} - \delta_{min} + \frac{N_F}{f_s} && (8.11)\\ &= s_{n+1} + \delta_0 + \delta_{max} - \delta_{min}. \quad \square && (8.12) \end{aligned}$$

Step 2: Estimation

$$\begin{aligned} (8.3) &\Rightarrow \delta_n \leq \delta_0 + \delta_{max} - \delta_{min} && (8.13)\\ r_n &= s_n + \delta_0 + \delta_{max} - \delta_{min} \geq s_n + \delta_n. \quad \square && (8.14) \end{aligned}$$

Proof of Maximum Storage

Proof of an upper boundary for the number of packets waiting for processing at the receiver, namely

$$r_n \leq s_{n+i} + \delta_{n+i} \qquad \forall i \geq \left\lceil \frac{(2 \cdot \delta_{max} - \delta_{min}) \cdot f_s}{N_F} \right\rceil. \tag{8.15}$$

Given as proven above

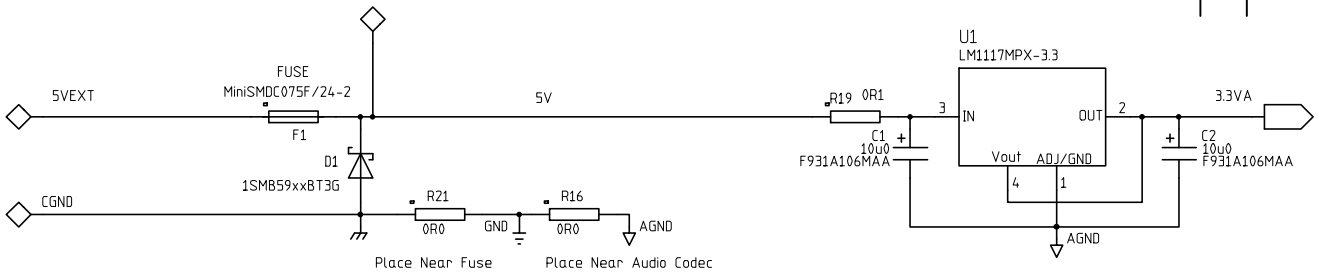
$$\begin{aligned} & r_n = s_n + \delta_0 + \delta_{max} - \delta_{min} && (8.16)\\ \overset{(8.3)}{\Rightarrow}\ & r_n \leq s_n + 2 \cdot \delta_{max} - \delta_{min} && (8.17)\\ \Rightarrow\ & r_n \leq s_n + \frac{(2 \cdot \delta_{max} - \delta_{min}) \cdot f_s}{N_F} \cdot \frac{N_F}{f_s} && (8.18)\\ \Rightarrow\ & r_n \leq s_n + i \cdot \frac{N_F}{f_s} \qquad \forall i \geq \left\lceil \frac{(2 \cdot \delta_{max} - \delta_{min}) \cdot f_s}{N_F} \right\rceil && (8.19)\\ \overset{(8.1)}{\Rightarrow}\ & r_n \leq s_{n+i} \qquad \forall i \geq \left\lceil \frac{(2 \cdot \delta_{max} - \delta_{min}) \cdot f_s}{N_F} \right\rceil && (8.20)\\ \Rightarrow\ & r_n \leq s_{n+i} + \delta_{n+i} \qquad \forall i \geq \left\lceil \frac{(2 \cdot \delta_{max} - \delta_{min}) \cdot f_s}{N_F} \right\rceil. \quad \square && (8.21) \end{aligned}$$

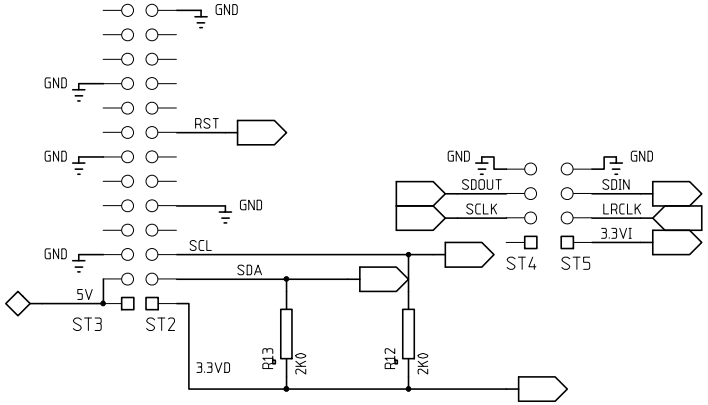
Schematics

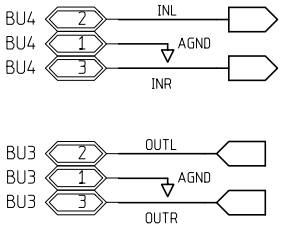
DWG. NO.	SH.	REV

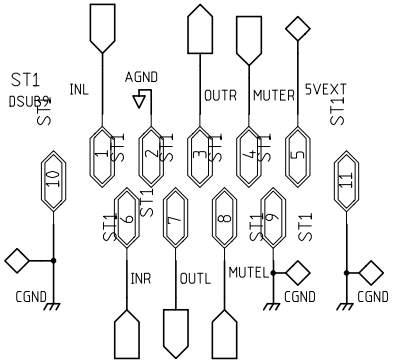
REVISIONS

REV	DESCRIPTION	DATE	APPROVED





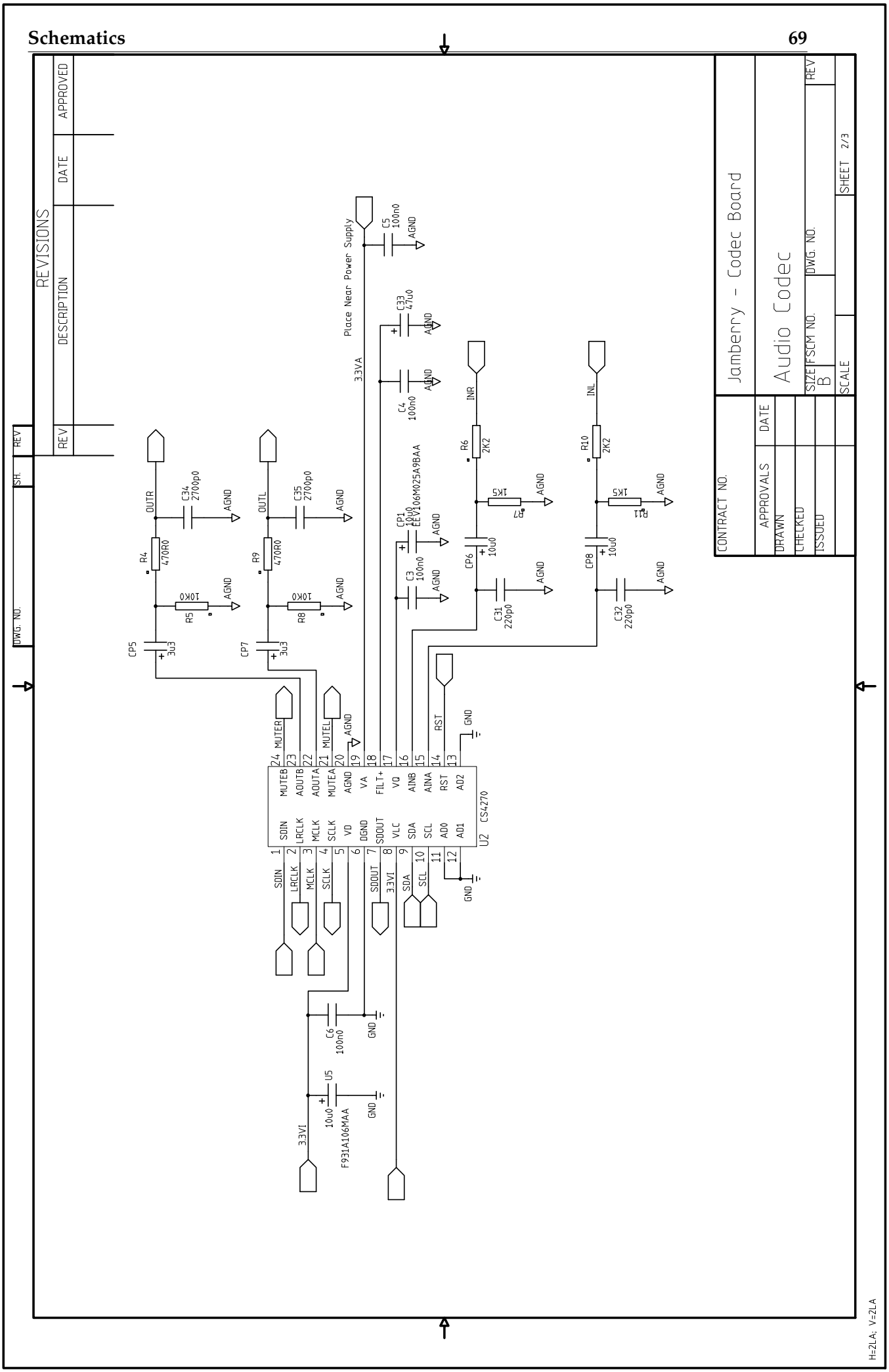




CONTRACT NO.	Jamberry - Codec Board
APPROVALS / DATE	Connectors
DRAWN	SIZE B / FSCM NO. / DWG. NO. / REV
CHECKED	SCALE / SHEET 1/3
ISSUED	

68

H=2LA; V=2LA

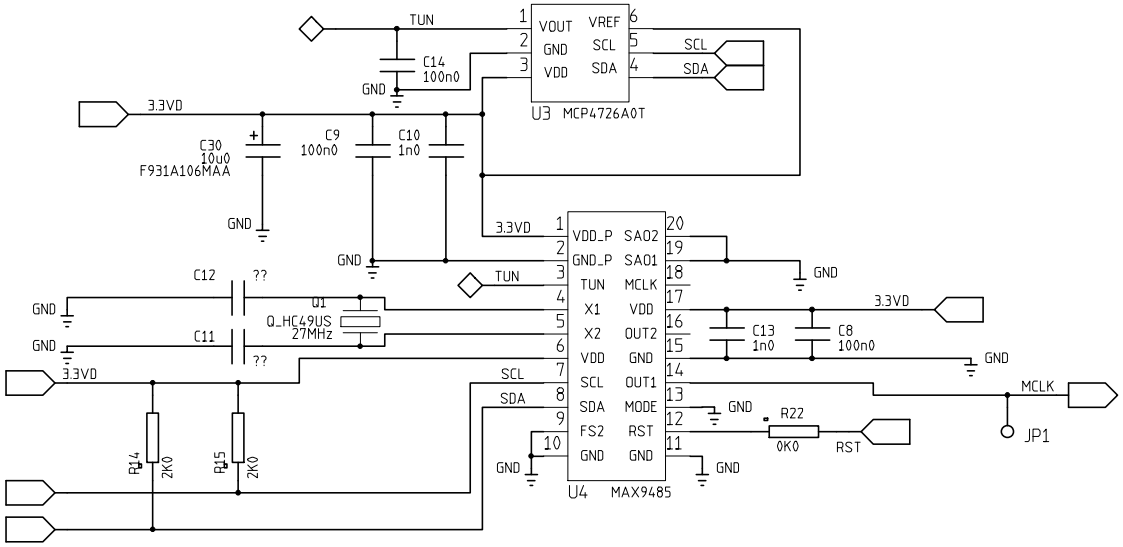

REVISIONS
REV
DESCRIPTION
DATE
APPROVED
SH.
REV
DWG. NO.
Jamberry - Codec Board
Audio Codec
CONTRACT NO.
APPROVALS
DATE
DRAWN
CHECKED
ISSUED
SIZE
B
FSCM NO.
DWG. NO.
REV
SCALE
SHEET 2/3
Place Near Power Supply
3.3VA
3.3VI
AGND
GND
OUTR
OUTL
INR
INL
R4 470R0
R5 10K0
R9 470R0
R8 10K0
R6 2K2
R7 1K5
R10 2K2
R11 1K5
C34 2700p0
C35 2700p0
C5 100n0
C33 47u0
C4 100n0
CP1 10u0 EEV106M025A9BAA
C3 100n0
CP5 3u3
CP7 3u3
CP6 10u0
CP8 10u0
C31 220p0
C32 220p0
C6 100n0
U5 10u0 F931A106MAA
U2 CS4270
SDIN
LRCLK
MCLK
SCLK
SDOUT
SDA
SCL
RST
MUTEB
AOUTB
AOUTA
MUTEA
AGND
VA
FILT+
VQ
AINB
AINA
AD2
AD1
AD0
VLC
DGND
VD
MUTER
MUTEL
H=2LA; V=2LA

Schematics

DWG. NO.	SH.	REV

REVISIONS

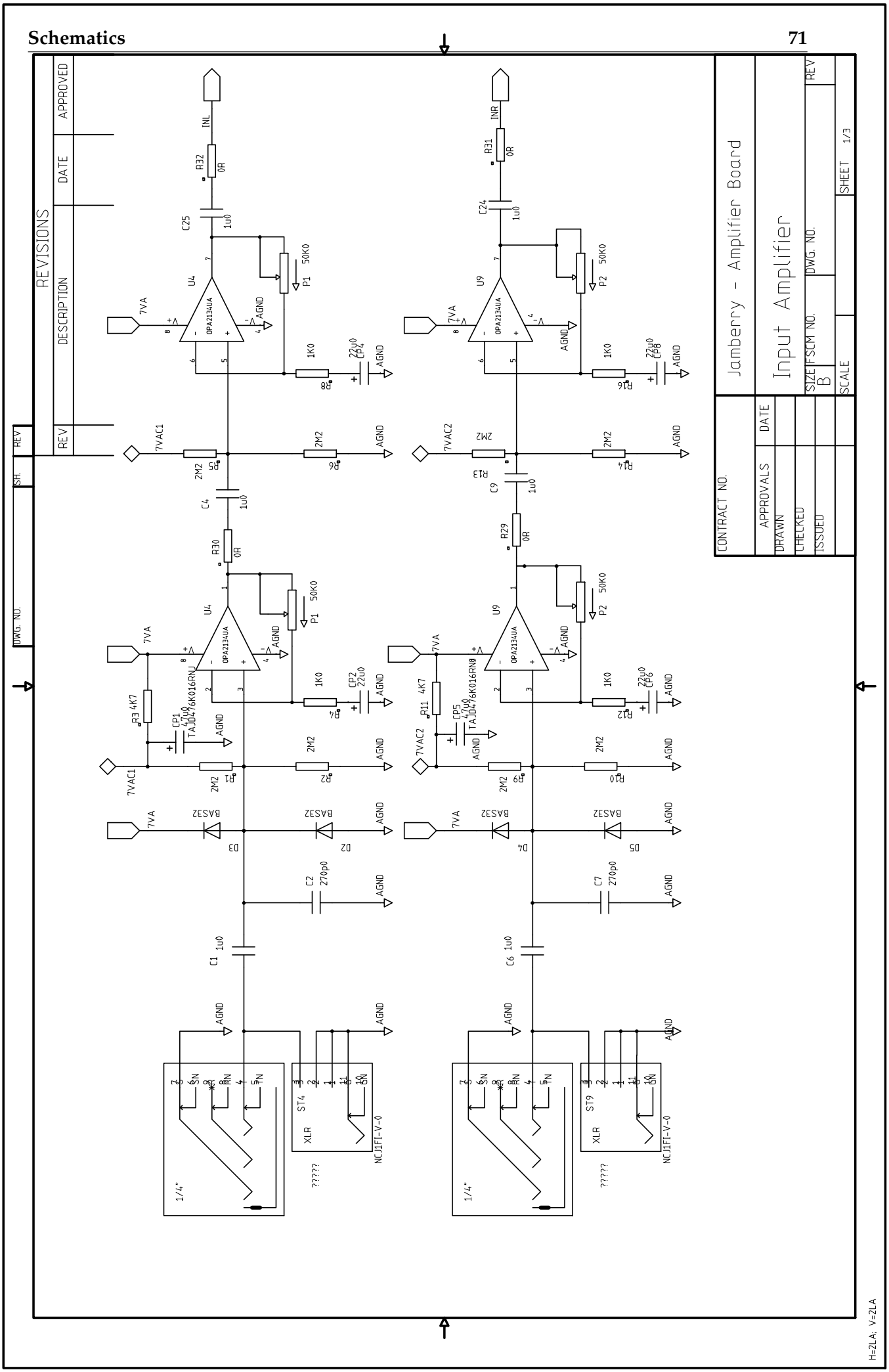
REV	DESCRIPTION	DATE	APPROVED



CONTRACT NO.		Jamberry - Codec Board			
APPROVALS	DATE	Clock Generator			
DRAWN					
CHECKED		SIZE B	FSCM NO.	DWG. NO.	REV
ISSUED		SCALE		SHEET 3/3	

70

H=2LA; V=2LA

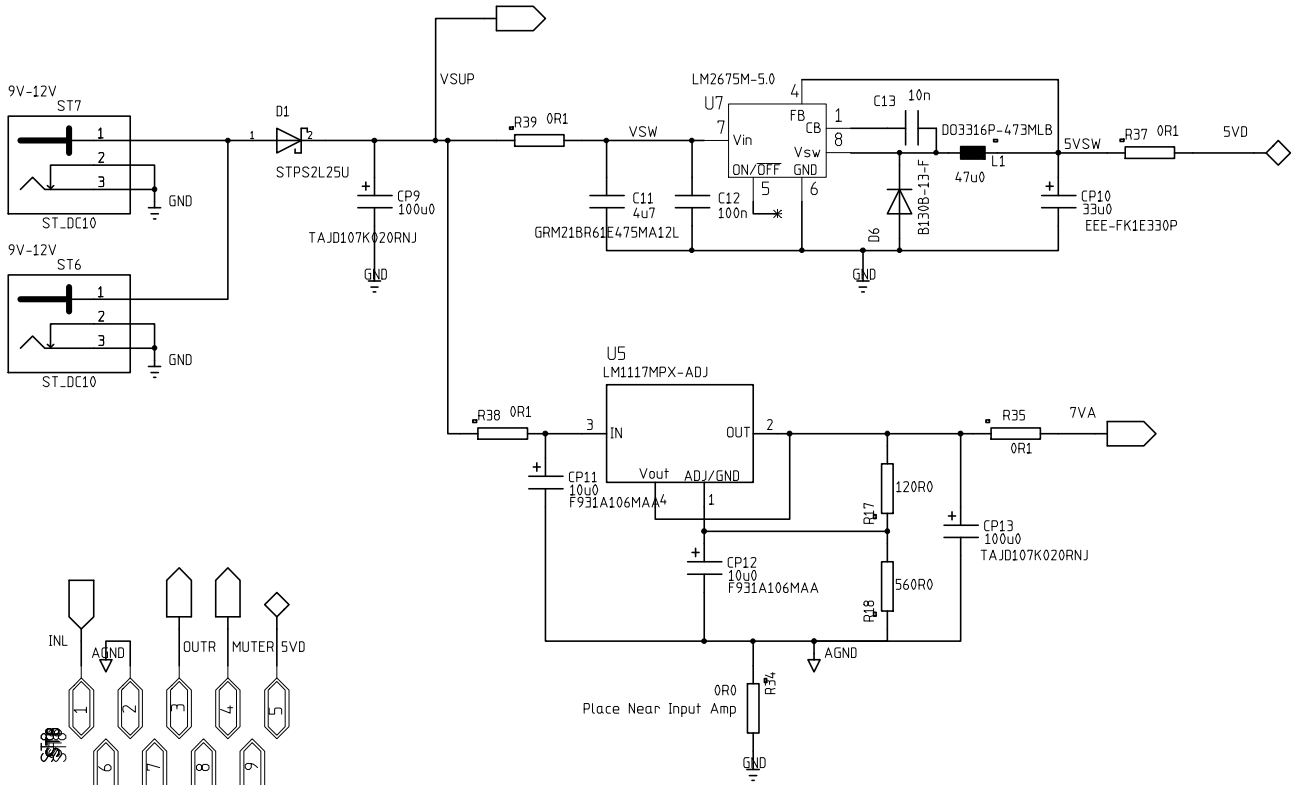

REVISIONS
REV
DESCRIPTION
DATE
APPROVED
SH.
REV
DWG. NO.
Jamberry - Amplifier Board
Input Amplifier
CONTRACT NO.
APPROVALS
DATE
DRAWN
CHECKED
ISSUED
SIZE
B
FSCM NO.
DWG. NO.
REV
SCALE
SHEET 1/3
INL
INR
R32 0R
R31 0R
C25 1u0
C24 1u0
U4
U9
OPA2134UA
7VA
AGND
P1 50K0
P2 50K0
1K0
CP4 22u0
CP8 22u0
7VAC1
7VAC2
2M2
R5
R6
R13
R14
C4 1u0
C9 1u0
R30 0R
R29 0R
R3 4K7
R11 4K7
CP1 47u0 TAJD476K016RNJ
CP5 47u0 TAJD476K016RNJ
CP2 22u0
CP6 22u0
R1
R2
R9
R10
BAS32
D3
D2
D4
D5
C2 270p0
C7 270p0
C1 1u0
C6 1u0
1/4"
XLR
ST4
ST9
NCJ1FI-V-0
?????
H=2LA; V=2LA

Schematics

DWG. NO.	SH.	REV

REVISIONS

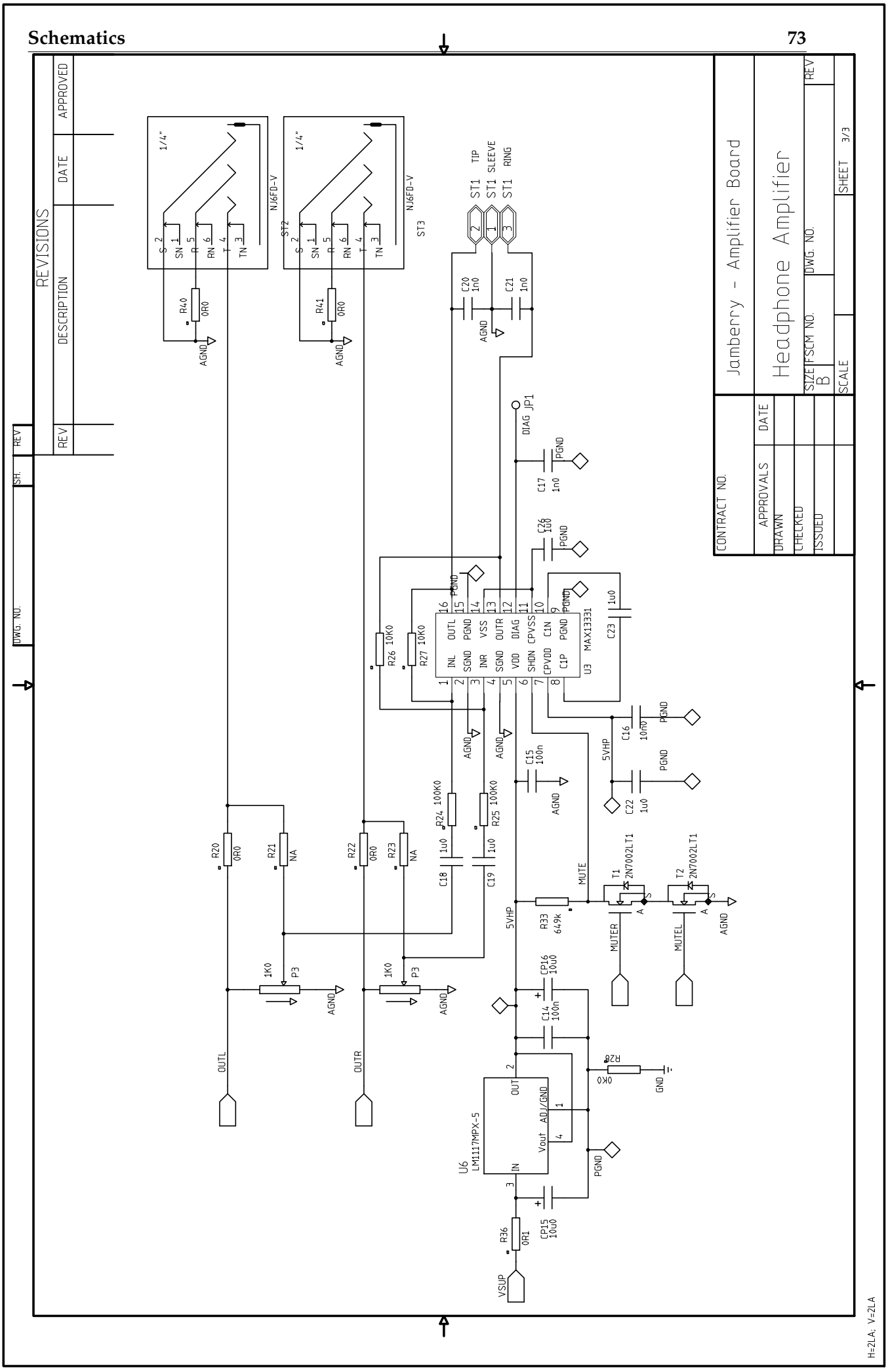
REV	DESCRIPTION	DATE	APPROVED



CONTRACT NO.	Jamberry - Amplifier Board
APPROVALS / DATE	Power Supply and D-Sub
DRAWN	SIZE B / FSCM NO. / DWG. NO. / REV
CHECKED	SCALE / SHEET 2/3
ISSUED	

72

H=2LA; V=2LA

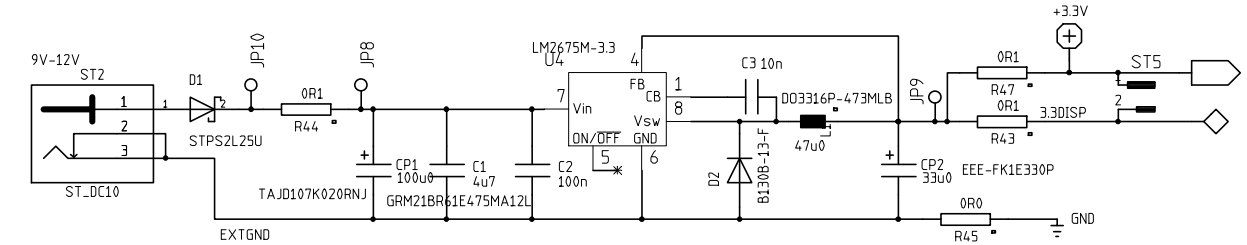

REVISIONS
REV
DESCRIPTION
DATE
APPROVED
Jamberry - Amplifier Board
Headphone Amplifier
SIZE B
FSCM NO.
DWG. NO.
REV
SHEET 3/3
SCALE
CONTRACT NO.
APPROVALS
DATE
DRAWN
CHECKED
ISSUED
DWG. NO.
SH.
REV
1/4"
NJ6FD-V
ST1 TIP
ST1 SLEEVE
ST1 RING
ST2
ST3
R40 0R0
R41 0R0
AGND
C20 1n0
C21 1n0
DIAG JP1
C17 1n0
PGND
C26 100n
U3 MAX13331
INL OUTL PGND VSS OUTR DIAG CPVSS C1N
SGND INR SGND VDD SHDN CPVDD C1P PGND
R26 10K0
R27 10K0
C23 1u0
C16 10n0
C22 1u0
5VHP
C15 100n
R20 0R0
R21 NA
R22 0R0
R23 NA
R24 100K0
R25 100K0
C18 1u0
C19 1u0
MUTE
T1 2N7002LT1
T2 2N7002LT1
R33 649k
MUTER
MUTEL
1K0 P3
CP16 10u0
C14 100n
OUTL
OUTR
R28 0K0
GND
U6 LM1117MPX-5
OUT ADJ/GND Vout IN
CP15 10u0
R36 0R1
VSUP
H=2LA. V=2LA

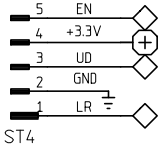
Schematics

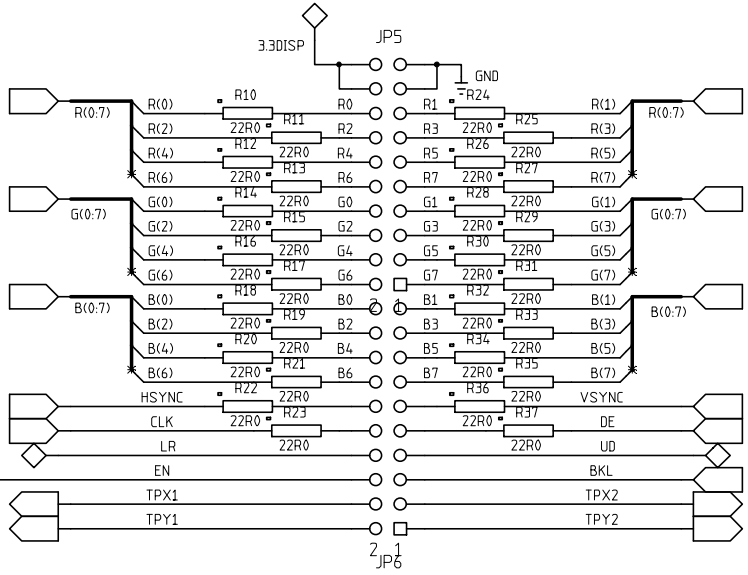
DWG. NO.	SH.	REV

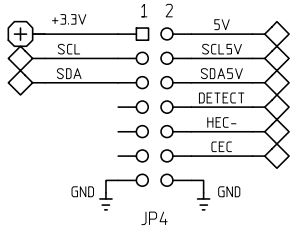
REVISIONS

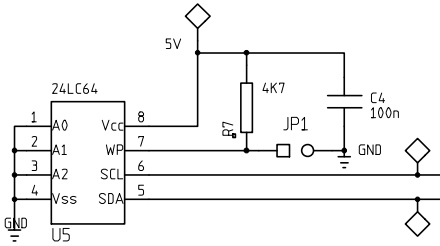
REV	DESCRIPTION	DATE	APPROVED

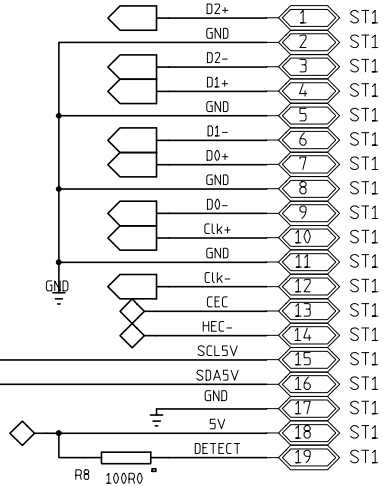


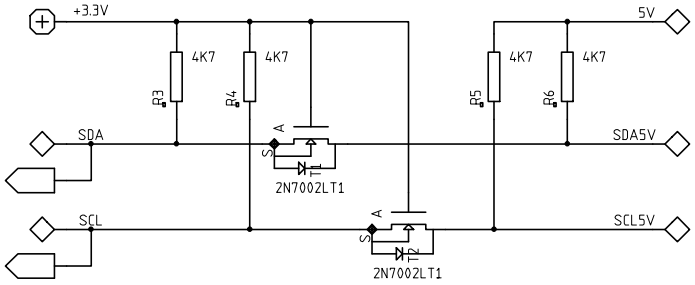








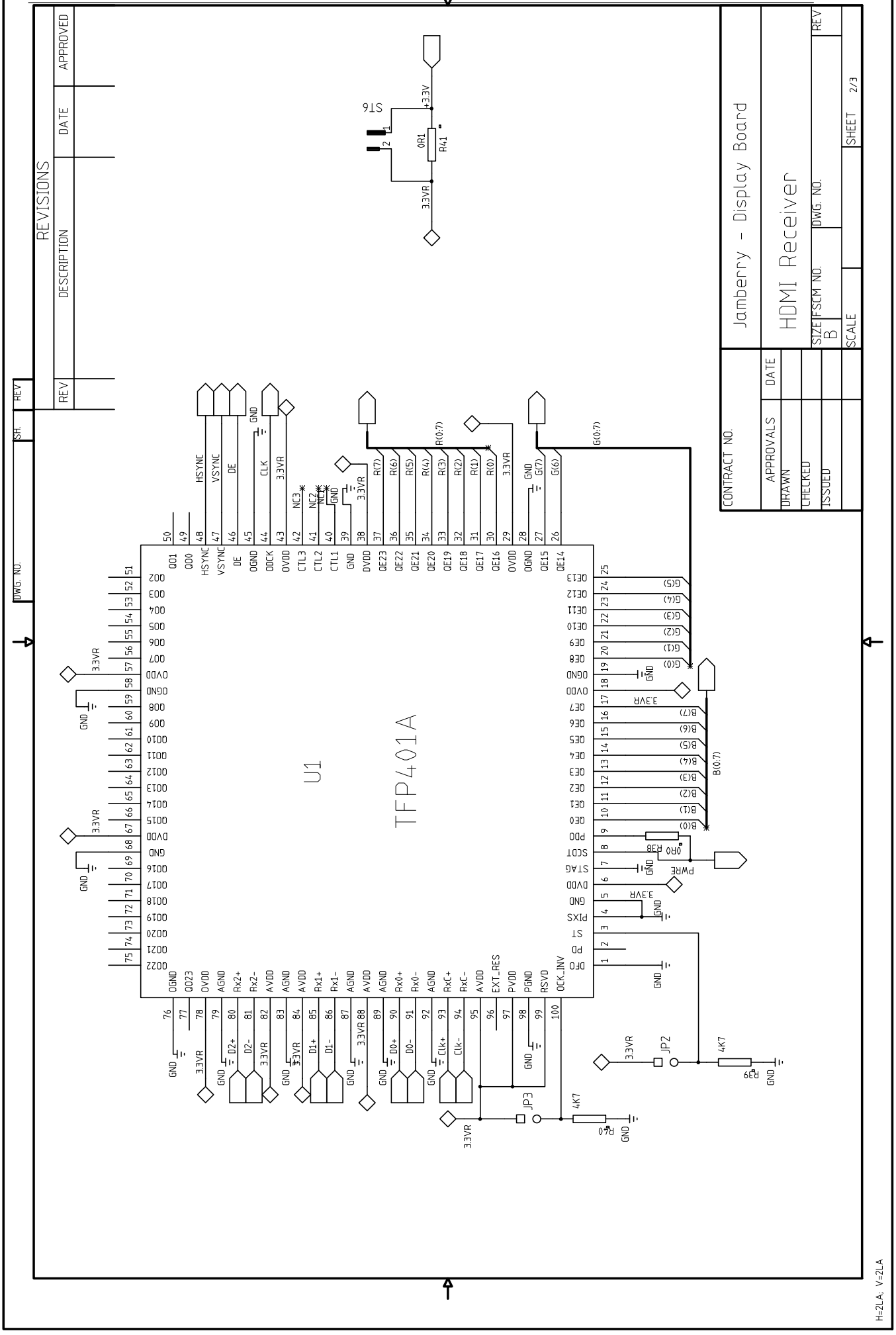




CONTRACT NO.	Jamberry - Display Board
APPROVALS / DATE	Connectors and Supply
DRAWN	SIZE B / FSCM NO. / DWG. NO. / REV
CHECKED	SCALE / SHEET 1/3
ISSUED	

74

H=2LA; V=2LA

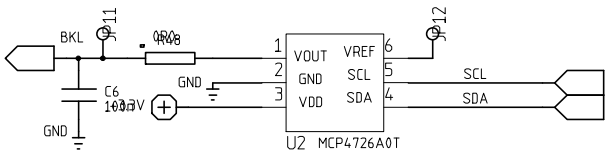

REVISIONS
REV
DESCRIPTION
DATE
APPROVED
Jamberry - Display Board
HDMI Receiver
CONTRACT NO.
APPROVALS
DRAWN
CHECKED
ISSUED
SIZE B
SCALE
DWG. NO.
REV
SHEET 2/3
U1
TFP401A
ST6
R41
0R1
3.3VR
GND
HSYNC
VSYNC
DE
CLK
R(0:7)
G(0:7)
B(0:7)
R(7)
R(6)
R(5)
R(4)
R(3)
R(2)
R(1)
R(0)
G(7)
G(6)
G(5)
G(4)
G(3)
G(2)
G(1)
G(0)
B(7)
B(6)
B(5)
B(4)
B(3)
B(2)
B(1)
B(0)
NC3
NC2
NC1
CTL3
CTL2
CTL1
R38
0R0
PWRE
JP2
JP3
4K7
R39
R40
D0+
D0-
D1+
D1-
D2+
D2-
Clk+
Clk-
Rx0+
Rx0-
Rx1+
Rx1-
Rx2+
Rx2-
RxC+
RxC-
H=2LA, V=2LA

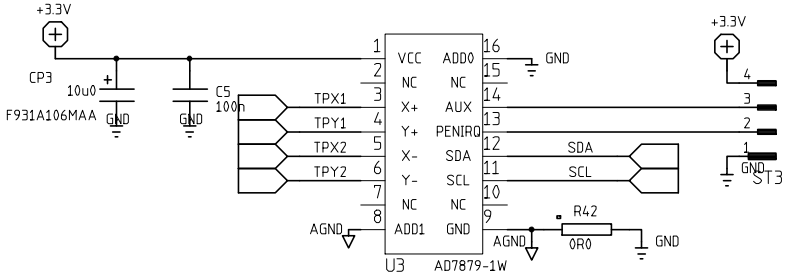
Schematics

DWG. NO.	SH.	REV

REVISIONS

REV	DESCRIPTION	DATE	APPROVED





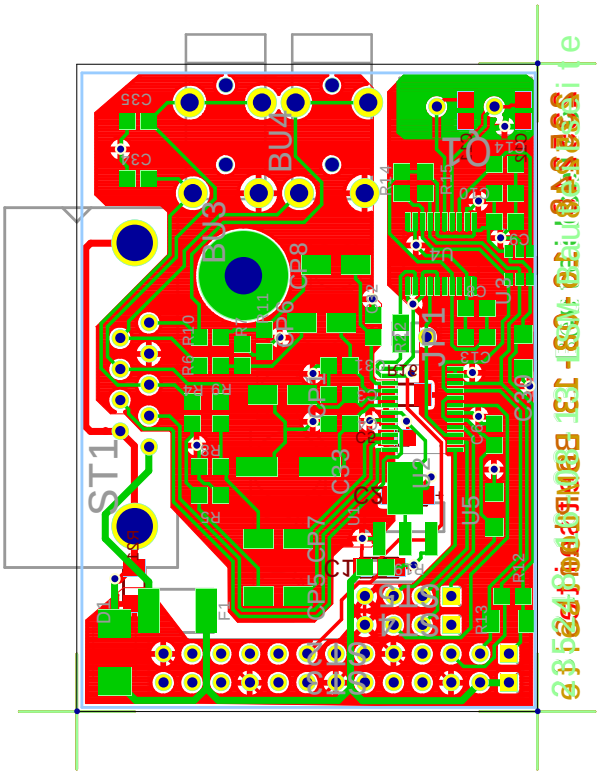
CONTRACT NO.		Jamberry - Display Board			
APPROVALS	DATE	Controllers			
DRAWN		SIZE B	FSCM NO.	DWG. NO.	REV
CHECKED		SCALE		SHEET 3/3	
ISSUED					

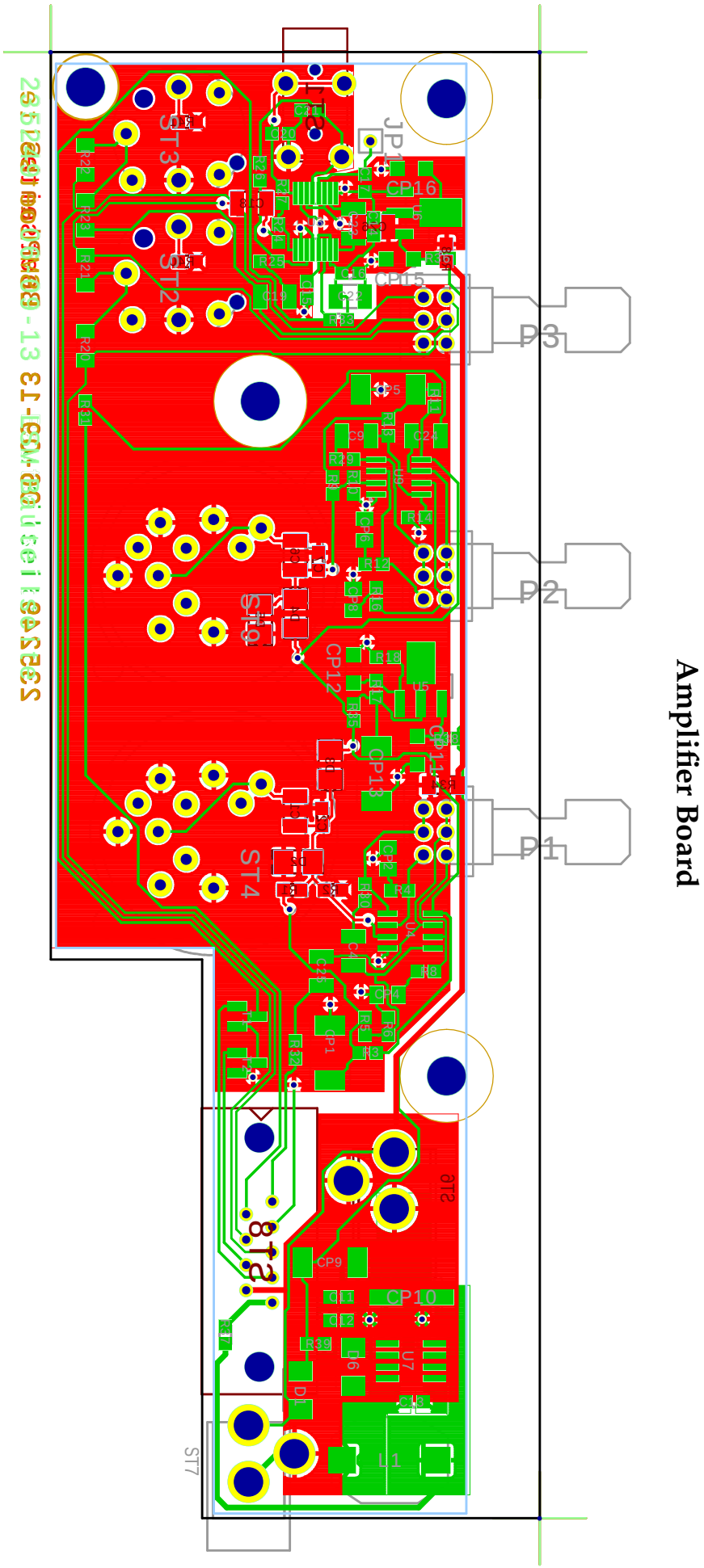
76

H=2LA; V=2LA

PCB Layouts

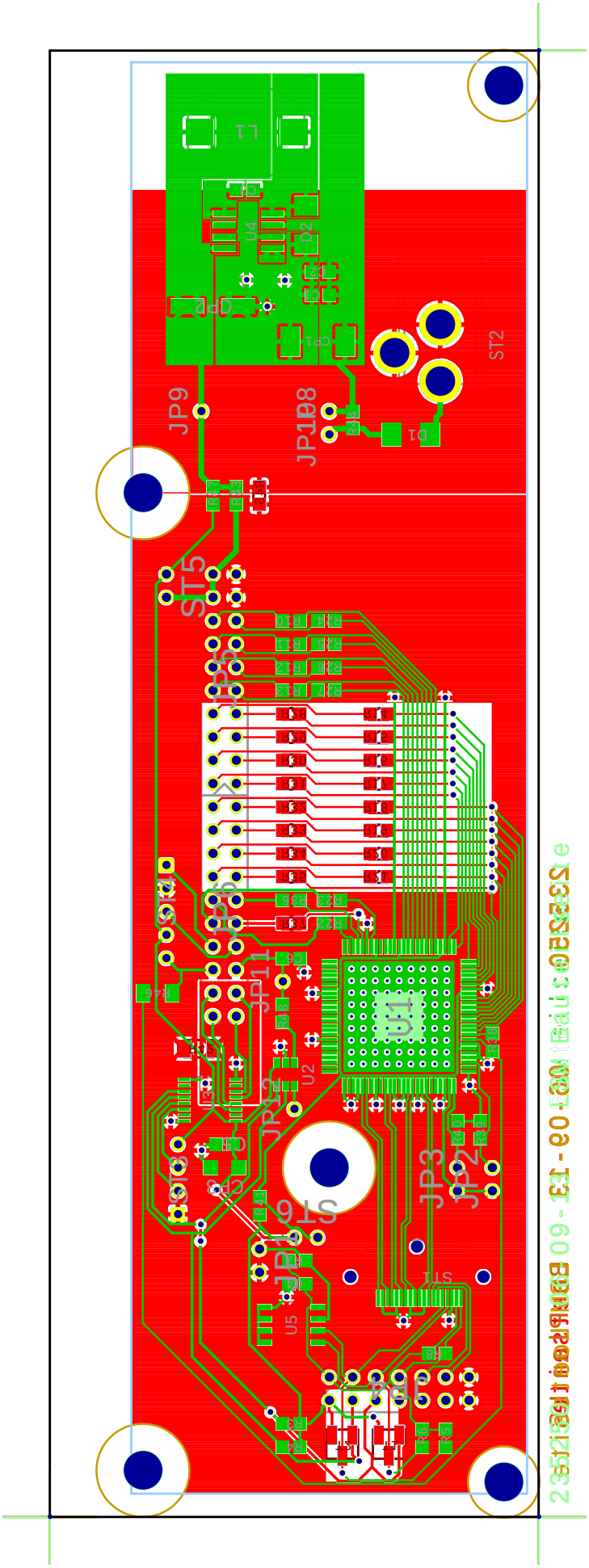
Codec Board





Amplifier Board

Display Board



Bill of Materials

Codec Board

Type	Name	Value	Manufacturer	Part No.	Qty
Fuse	F1	1.5 A	TE Connectivity	MiniSMDC075F/24-2	1
Resistor	R19	0R1	KOA Speer	SR732ATTER100F	1
Resistor	R20	0R0	KOA Speer	RK73Z2ATTD	1
Resistor	R5, R8	10K0	KOA Speer	SG73P2ATTD1002F	2
Resistor	R6, R10	2K2	KOA Speer	SG73S2ATTD2201F	2
Resistor	R7, R11	1K5	KOA Speer	SG73P2ATTD1501F	2
Resistor	R4, R9	470R0	KOA Speer	SG73P2ATTD4700F	2
Resistor	R12, R14, R15	2K0	KOA Speer	SG73S2ATTD2201F	3
Resistor	R16, R22	0R0	KOA Speer	RK73Z2BTTE	2
Resistor	R13	2K0	Panasonic	ERJ-8ENF2001V	1
Capacitor (Cer.)	C34, C35	2700p0	AVX	08055C272K4T2A	2
Capacitor (Cer.)	C3, C4, C5, C6, C8, C9, C14	100n0	AVX	08053C104KAT2A	7
Capacitor (Cer.)	C10, C13	1n0	Vishay	VJ0805A102KXAPW1BC	2
Capacitor (Cer.)	C31, C32	220p0	Vishay	VJ0805A221JXJCW1BC	2
Capacitor (Cer.)	C11, C12	18p0	Vishay	VJ0805A180GXACW1BC	2
Capacitor (Elec.)	CP1, CP6, CP8	10u0	Kemet	EEV106M025A9BAA	3
Capacitor (Elec.)	CP5, CP7	3u3	Panasonic	EEE-1EA3R3NR	2
Capacitor (Elec.)	C33	47u0	Panasonic	EEE-TQV470XAP	1
Capacitor (Tnt.)	C1-C2, U5, C30	10u0	AVX	F931A106MAA	4

Type	Name	Value	Manufacturer	Part No.	Qty
Zener-Diode	D1	5.6 V	ON Semiconductor	1SMB5919BT3G	1
Regulator	U1		Texas Instruments	LM1117MPX-3.3/NOPB	1
Crystal	Q1		Citizen	HC49US-27.000MABJB	1
Audio Jack	BU3-BU4			Lumberg 1503 08	2
Audio Codec	U2		Cirrus Logic	CS4270-CZZ	1
DA-Converter	U3		Microchip	MCP4726A0T-E/CH	1
Clock Generator	U4		Maxim Integrated	MAX9485EUP+	1
Pin Header	ST2-ST3		FCI	87606-313LF	1
Pin Header	ST4-ST5		FCI	87606-804LF	1
Pin Header	ST1		TE Connectivity	3-1634580-2	1

Amplifier Board

Type	Name	Value	Manufacturer	Part No.	Qty
Resistor	R1, R2, R5, R6, R9, R10, R13, R14	2M2	Vishay	CRCW08052M20FKEA	8
Resistor	R3, R7, R11, R15	4K7	KOA Speer	SG73P2ATTD4701F	4
Resistor	R4, R8, R12, R16	1K0	KOA Speer	SG73P2ATTD1001F	4
Resistor	R17, R19	120R0	KOA Speer	RK73H2ATTDD1200F	2
Resistor	R18	560R0	KOA Speer	SG73S2ATTD5600F	1
Resistor	R20, R22	0R0	KOA Speer	RK73Z2ATTD	2
Resistor	R24, R25	100K0	Panasonic	ERJ-6ENF1003V	2
Resistor	R26, R27	10K0	Panasonic	ERJ-6ENF1002V	2
Resistor	R28	0R0	KOA Speer	RK73Z2BTTE	1
Resistor	R33	649K	Panasonic	ERJ-6ENF6493V	1
Potentiometer	P1, P2	50K0	Bourns	PTD902-2015K-A503	2
Potentiometer	P3	1K0	Bourns	PTD902-1015K-A102	1

Type	Name	Value	Manufacturer	Part No.	Qty
Capacitor (Film)	C1, C4, C6, C9, C18, C19, C22, C23, C24, C25	1u0	Cornell Dubilier	FCA1210C105M-G2	8
Capacitor (Cer.)	C11	4u7	Murata	GRM21BR61E475MA12L	1
Capacitor (Cer.)	C12, C14, C15	100n0	AVX	08053C104KAT2A	3
Capacitor (Cer.)	C13, C16	10n0	Murata	GRM216R71H103KA01D	2
Capacitor (Cer.)	C17, C20, C21	1n0	Vishay	VJ0805A102KXAPW1BC	3
Capacitor (Cer.)	C2, C7	270p0	Vishay	VJ0805A271KXXCW1BC	2
Capacitor (Tnt.)	CP1, CP3, CP5, CP7	47u0	AVX	TAJD476K016RNJ	4
Capacitor (Tnt.)	CP9, CP13	100u0	AVX	TAJD107K020RNJ	2
Capacitor (Elec.)	CP10	33u0	Panasonic	EEE-FK1E330P	1
Capacitor (Tnt.)	CP11, CP12, CP15, CP16	10u0	AVX	F931A106MAA	4
Capacitor (Tnt.)	CP2, CP4, CP6, CP8	22u0	Kemet	T491A226K010AT	4
Diode	D6		Diodes Inc.	B130B-13-F	1
Diode	D1		STMicroelec.	STPS2L25U	1
Regulator	U5		Texas Instruments	LM1117MPX-ADJ/NOPB	1
Regulator	U6		Texas Instruments	LM1117MPX-5.0/NOPB	1
Regulator	U7		Texas Instruments	LM2675M-5.0	1
Inductor	L1		Coilcraft	DO3316P-473MLB	1
Connector	ST2-ST3		Neutrik	NJ6FD-V	2
Connector	ST4, ST9		Neutrik	NCJ9FI-V-0	2
Connector	ST6-ST7		Switchcraft	RAPC722X	2
Connector	ST1		Lumberg	1503 08	1
Amplifier	U3		Maxim Integrated	MAX13331GEE/V+	1
Pin Header	ST8		TE Connectivity	3-1634222-2	1

Type	Name	Value	Manufacturer	Part No.	Qty
Diode	D2-D5		NXP	BAS32L,115	4
Amplifier	U4, U9		Texas Instruments	OPA2134UA	2

Display Board

Type	Name	Value	Manufacturer	Part No.	Qty
Resistor	R1, R38, R42	0R0	KOA Speer	RK73Z2ATTD	3
Resistor	R3, R4, R5, R6, R7, R37, R40	4K7	KOA Speer	SG73P2ATTD4701F	7
Resistor	R41	0R1	KOA Speer	SR732ATTER100F	1
Resistor	R10-R37	22R0	KOA Speer	SG73P2ATTD22R0F	28
Resistor	R8	100R0	KOA Speer	SG73P2ATTD1000F	1
Capacitor (Cer.)	C1	4u7	Murata	GRM21BR61E475MA12L	1
Capacitor (Cer.)	C2, C4, C5, C6	100n0	AVX	08053C104KAT2A	4
Capacitor (Cer.)	C3	10n0	Murata	GRM216R71H103KA01D	1
Capacitor (Elec.)	CP2	33u0	Panasonic	EEE-FK1E330P	1
Capacitor (Tnt.)	CP3	10u0	AVX	F931A106MAA	1
Capacitor (Tnt.)	CP1	100u0	AVX	TAJD107K020RNJ	1
Diode	D2		Diodes Inc.	B130B-13-F	1
Diode	D1		STMicroelec.	STPS2L25U	1
Transistor	T1-T2		ON Semiconductor	2N7002LT1G	2
Regulator	U4		Texas Instruments	LM2675M-3.3/NOPB	1
EEPROM	U5		Microchip	24LC64T-I/SN	1
Inductor	L1		Coilcraft	DO3316P-473MLB	1
Connector	ST1		Molex	47659-1000	1
Connector	ST2		Switchcraft	RAPC722X	1
HDMI Receiver	U1		Texas Instruments	TFP401APZP	1
Touch Controller	U3		Analog Devices Inc.	AD7879-1WARUZ-RL7	1

Type	Name	Value	Manufacturer	Part No.	Qty
DA-Converter	U2		Microchip	MCP4726A2T-E/CH	1
Connector			TE Connectivity	1658620-9	2
Ribbon Cable			Amphenol	135-2801-040FT	1
Connector			TE Connectivity	499252-1	2
Connector			Kobiconn	171-3215-EX	2
HDMI Cable			SpeaKa	HDMI 0,3 m	1

List of Acronyms

Acronym	Description	Page List
AC	Alternating Current An electrical current with alternating direction. In this context, a superposition of sine waves with various frequencies.	42, 43, 49, 54
ADC	Analog to Digital Converter Transfers an analog signal into a digital signal.	7, 12, 24, 30, 31, 34, 41, 54
ALSA	Advanced Linux Sound Architecture A subsystem of the Linux kernel for interfacing sound hardware.	15, 16, 22, 24–26, 29, 30, 32–34, 54, 85
AMBA	Advanced Microcontroller Bus Architecture A data bus used for connecting various compontents inside a SoC.	14, 29, 32
API	Application Programming Interface A set of functions for accessing a software component, e.g. a library.	15, 16, 32, 33
ARM	Advanced Reduced Instruction Set Computing Machines A computer architecture invented by a company of the same name. Popular for embedded systems.	14, 32, 61
ASoC	ALSA System On Chip A part of ALSA, specific for embedded platforms.	29, 31–33
CEC	Consumer Electronics Control Allows control of multimedia devices through high definition multimedia interface (HDMI) links.	14
CPU	Central Processing Unit A piece of hardware for executing instructions, i.e. running software.	14, 16, 30, 32, 45, 50–52, 86, 87
DAC	Digital to Analog Converter Transfers a digital signal into an analog signal.	7, 12, 30, 37, 42, 45
DAI	Digital Audio Interface Electrical interface for transmitting audio data. An example is I^2S .	31–33

Acronym	Description	Page List
DC	Direct Current An unidirectional electrical current. In this context, a current with constant or slow changing voltage.	41, 43
DMA	Direct Memory Access A method for transferring data from a peripheral to the RAM or back without occupying the CPU.	14, 30–33, 86
DREQ	Data Request A signal used for signalling the DMA controller that new data should be send or received.	31–33
DSL	Digital Subscriber Line A technique to transmit digital data via regular telephone wires and thus providing Internet access.	6, 10
DSP	Digital Signal Processor A processor dedicated for handling digital signals like audio or video. Compared to a classical microprocessor, it has a special architecture and a lot of extra instructions for fast (real-time) processing.	12
DVB	Digital Video Broadcast A method for broadcasting video and audio data via digital channels. Widely used for television.	6
GPIO	General Purpose Input Output Lines of ICs that have no dedicated functionality and can be used for application specific control.	49
HDMI	High Definition Multimedia Interface A standard for transmitting audio and video signals. Commonly used in consumer electronics.	12, 14, 39, 44, 45, 62, 85
I/O	Input/output Communication of a IC with the outside world (other ICs, sensors...)	29, 33
I^2C	Inter Integrated Circuit Electrical interface for transmitting generic digital data over short distances with low speed (e.g. for control data)	12–14, 31, 33, 34, 39, 45
I^2S	Integrated Interchip Sound Electrical interface for transmitting digital audio data over short distances	13–15, 24, 29–34, 39, 61
IC	Integrated Circuit	11–13, 33, 35, 40, 43, 86, 87
IPDV	IP Packet Delay Variation Difference between the one-way-delay of two packets.	10
JSON	JavaScript Object Notation A string based data format for transmitting data between applications.	27

Acronym	Description	Page List
MDCT	Modified Discrete Cosine Transform A transform between time and frequency domain, similar to the Fourier transform, but with some advantages for audio coding.	10
MIDI	Musical Instrument Digital Interface A protocol for transmitting audio event messages, such as tone pitch and velocity.	5, 6
P2P	Peer-to-peer Direct connection between clients without an intermeditate server.	6–8
PWM	Pulse Wide Modulation A technique for modulating an analog signal by using a digital signal with a fixed frequency, but variable duty cycle.	37, 54
QoS	Quality Of Service Methods for increasing the quality of a network connection. For example by reserving a certain amount of network bandwidth at each node of the network.	5
RAM	Random Access Memory Memory inside a computer that provides random access to the data by the CPU.	14, 30–32, 86
RTT	Round Trip Time The time for a signal to travel back and forth along a network connection.	5
SNR	Signal To Noise Ratio A measure for the amount of noise in a signal.	54, 56, 58
SoC	System On Chip A microprocessor and some peripheral integrated in a single IC	11, 12, 14, 29, 33, 85
SPI	Serial Peripheral Interface An electrical interface for transmitting general purpose data. An alternative to I^2C .	31
SVG	Scalable Vector Graphics A file format that carries vector graphics.	27
TCP	Transmission Control Protocol A transport protocol for transmission of data streams via the Internet. In contrast to UDP, several techniques improve the reliability of the transmission.	5, 19
TFT	Thin Film Transistor A technology for building liquid-crystal displays.	44, 45
THD	Total Harmonic Distortion A measure for the amount of harmonic distortions induced by non-linearities.	54, 56, 58

Acronym	Description	Page List
UDP	Transmission Control Protocol A transport protocol for transmission of data packets via the Internet.	5, 19, 87
URI	Uniform Resource Identifier A string that identifies a resource, especially in the context of web applications.	27
VoIP	Voice over IP Communicating over the Internet like with a classical telephone.	10
WAMP	WebSocket Application Messaging Protocol A protocol that provides remote procedure calls and the publish and subscribe pattern via WebSocket.	27

Content of the DVD

The attached DVD contains the implemented software components and PCB files. Furthermore, an electronic version of this thesis is contained. The directory structure consists of the following parts:

- The simulator used for evaluating the system is stored in the *echo* directory.
- In the directory *jamberry/src*, the Audio over IP software as presented in this thesis is stored. The GUI can be found in *jamberry/gui*. Some other directories in *jamberry* provide additional modules needed for compilation.
- The directory *kernel* contains the Linux kernel sources as used on the embedded system.
- *latex* contains the source code for compiling this thesis.
- The schematics and the PCB layouts can be found in *pcb*.
- A PDF version of this thesis is located in the root directory.