Aufzeichnung und Übertragung von Audio over IP Daten und Synchronisation mit den zugehörigen Bilddaten

Masterarbeit

durchgeführt von

Markus Luttenberger

am Institut für Breitbandkommunikation
der Technischen Universität Graz

Leiter: Univ.-Prof. Dipl.-Ing. Dr. Gernot Kubin

in Zusammenarbeit mit AviBit data processing GmbH

Begutachter: Ao. Univ.-Prof. Dipl.-Ing. Dr. Erich Leitgeb
Betreuer: Dipl.-Ing. Dr. Konrad Köck, Dipl.-Ing. Dr. Thomas Leitner

Graz, im September 2010

Kurzfassung

Advanced Surface Movement Guidance & Control Systems (A-SMGCS) werden im Tower von Flughäfen eingesetzt, um den bodennahen Flugverkehr computergestützt kontrollieren und steuern zu können. Die dortigen Abläufe sind sicherheitstechnisch sehr bedeutend. Es entstand die Anforderung sämtliche Aktivitäten (Bildschirminhalte, Kommunikationsprotokolle) während des Betriebs aufzuzeichnen, um bestimmte Situationen im Nachhinein wieder abspielen und analysieren zu können.

Für die Firma **AviBit**, welche ein eigenes A-SMGCS entwickelt hat, wurde das *Legal Recording and Replay* System im Rahmen dieser Diplomarbeit weiterentwickelt. Dieses System vereint Video und Audio Recording in netzwerkorientierten Applikationen. Die produzierten Daten werden archiviert und können zu einem beliebigen späteren Zeitpunkt in einem geeigneten Media Player wiedergegeben werden. Das Video Recording Programm war bereits vorhanden, ebenso der Video Player. Im Zuge der Diplomarbeit wurde die Audio Recording Komponente entwickelt und der Player erweitert, so dass eine synchrone Wiedergabe von Audio- und Videomaterial möglich ist.

Der theoretische Teil der vorliegenden Arbeit veranschaulicht den Weg von der Schallentstehung bis hin zum digitalen und codierten Audiosignal. Es werden ausgewählte Aspekte der Akustik, der Signalverarbeitung bezüglich Analog-Digital Wandlung und Speicherung, Komprimierung und Übertragung von digitalen Audiodaten behandelt. Dies umfasst auch die Voice over IP Technik, welche sich in Ansätzen auch in den implementierten Programmen widerspiegelt.

Der praktische Teil widmet sich der Entwicklung der Programmen *AudioRec* (der Audio Recorder) und der Erweiterung des Video Players *AVPlayer*. Es werden das zu Grunde liegende Kommunikationsprotokoll, das Speicherformat der Archivdateien und Design und Architektur der Programme dargestellt. Die entwickelten Programme zeigen, dass sie imstande sind, übers Netzwerk verschickte Audiodaten der Echtzeit zuzuordnen, zu speichern und schlussendlich synchron zu Videodaten im Replay auszugeben.

Abstract

Advanced Surface Movement Guidance & Control Systems (A-SMGCS) are used in airport towers to monitor and control air traffic at the ground level. In this field the safety procedures are critical. There is a requirement to record all activities (screen contents, communication protocols) during normal operation in order to be able to replay and analyze certain situations afterwards.

For the company **AviBit**, which developed their own A-SMGCS, the *Legal Recording and Replay* system was further developed as part of the thesis. The system combines video and audio recording in network-based applications. The produced data is archived and can be replayed at a later date with a suitable media player. The video recording program already existed as well as the video player. The development as part of this thesis includes the audio recording component and the extension of the media player, so that a synchronized playback of the audio and video data is possible.

The theoretical part of this thesis describes the path from the sound generation to the digital and encoded audio signal. There are specific aspects of acoustics, signal processing concerning analog-digital conversion and storage, compression and transmission of digital audio data discussed. This also includes the Voice over IP technology which is reflected in the implemented programs to some extent.

The practical part focuses on the programs *AudioRec* (the audio recorder) and the extension of the video player *AVPlayer*. The basic communication protocol, the storage format of the archive files and the application design and architecture is explained. The developed programs show that they are able to receive audio data over the network, store and map it to the real-time and finally replay audio and video data synchronously in the media player.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

Mein Dank gilt vor allem der Firma **AviBit** und dort im speziellen Dr. Konrad Köck und Dr. Thomas Leitner. Sie haben mir das Projekt ermöglicht und standen während der gesamten Entwicklung hilfreich zur Seite. Ebenso erwähnen möchte ich Dipl. Ing. Gerhard Hofbauer, welcher die Audio Hardware nach den gegeben Spezifikationen entwickelt hat. Mein Dank gilt auch meinem Betreuer, Dr. Erich Leitgeb, der das Thema sofort angenommen hat und die Diplomarbeit begutachtet hat. Abschließend gebührt mein Dank meiner Familie, meinen Freunden und sonstigen Bekannten, die mich während dieser Zeit unterstützt haben und für mich da waren!

Markus Luttenberger
Graz, Österreich, September 2010

1 Einführung

Während eines Arbeitsprozesses werden viele Daten generiert, deren Speicherung (digital wie auch analog) im Allgemeinen als überflüssig erscheint und auch teilweise tief in die Privatsphäre der Mitarbeiter eingreift. Dazu gehören beispielsweise Bewegungsdaten, Kommunikationsprotokolle (Telefon, E-Mail, etc.) und Informationen über Verhaltensweisen, Vorlieben und Abneigungen. Es gibt aber auch Daten deren Aufzeichnung längst als normal angesehen wird und teilweise auch gesetzlich verankert ist. Beispielsweise sind das Anwesenheitsdaten der Mitarbeiter, um eine faire Entlohnung zu garantieren oder auch Zugriffsrechte, um den Zugang oder Zugriff auf sensible Daten oder spezielle Räume und Orte zu gewähren bzw. zu schützen. Es gibt aber auch Bereiche wo Kompromisse über die Aufzeichnung von wichtigen Informationen gemacht werden müssen, weil sicherheitsrelevante Daten verarbeitet werden. Einem solchen Thema widmet sich die vorliegende Diplomarbeit.

1.1 Motivation

Das Projekt wurde im Rahmen der Firma **AviBit data processing GmbH**[1] abgewickelt. Das Softwareenwicklungsunternehmen bietet Lösungen für Flughäfen, um den dortigen Luftverkehr kontrollieren und steuern zu können, üblicherweise als *Advanced Surface Movement Guidance & Control System* (A-SMGCS) bezeichnet. Dazu gehören Plan- und Verwaltungswerkzeuge für landende und abfliegenden Flugzeuge, sowie eine komplette, radargestützte Visualisierung des Flughafens und Umfeldes für Fluglotsen, damit diese im Tower nicht ausschließlich auf Sichtfeldebene agieren müssen. Außerdem bietet die Firma ein digitales Flugstreifensystem an, um den Luftverkehr effizient verwalten zu können.

In diesem Bereich ist das Thema Sicherheit unumgänglich. Alle verarbeiteten Daten werden kontinuierlich aufgezeichnet, um Gefahrensituationen im Nachhinein nachspielen zu können, zu analysieren und aus Fehlern zu lernen. Diese Erfassung betrifft auch direkt die Arbeit der Fluglotsen.

Derzeit wird an der Entwicklung eines Systems gearbeitet, welches direkt die Bildschirminhalte der Fluglotsen aufzeichnet (Screen Capturing) und später bei Bedarf in einem geeigneten Player wiedergegeben werden kann. Das Projekt lief bislang unter dem Namen *Legal Recording*. Nun sollte es auch noch um die Möglichkeit der Sprachaufzeichnung erweiterte werden. Jeder Fluglotse steht mit Piloten und sonstigem Personal in Funkkontakt. Die Idee war also, alle Funkkanäle auf einem Gerät zusammenzufassen und codiert an einen Server weiterzuschicken. Auf diesem läuft eine Software, welche die Sprachdaten empfängt und archiviert. Für die Wiedergabe wird der bereits vorhandene Videoplayer um die Möglichkeit erweitert auch die gespeicherten Sprachdaten abzuspielen und somit das Videomaterial mit dem Audiomaterial zu synchronisieren.

In der Planungsphase wurde noch eine Lösung mittels eines standardisierten Voice over Internet Protocols (kurz: VoIP) erachtet. VoIP beschreibt eine Technik, um Telefongespräche über Computernetzwerke (Internet, LAN, etc.) zu führen. Da es sich ja um Sprachdaten handelt, erschien das als logischer Schluss. Die Kommunikation im Fall des Legal Recordings ist aber lediglich unidirektional: Von der Audio-Hardware zum Computer, der die Daten speichert. Es findet also kein *Telefongespräch* im herkömmlichen Sinn statt, sondern es werde nur Audiodaten geschickt. Schnell wurde also klar, dass die vorliegenden VoIP Standards zu umfangreich und kompliziert für diesen Anwendungsfall sind. Es wurde also ein schlankes und einfaches Protokoll entwickelt, welches auch leicht für die Audio-Hardware zu verarbeiten ist. Damit wurde letztendlich auch die Entwicklung der Audio-Hardware günstiger und einfacher.

[1] `http://www.avibit.com`

In Abbildung 1.1 ist eine grobe, konzeptionelle Architekturübersicht ersichtlich. Die Audio-Hardware wird mit *XC-1124A* bezeichnet, was für "24-Channel Network Audio System" steht. Dieses Gerät wurde im Auftrag der Firma **AviBit** von **Xerxes Technologies**[2] entwickelt und programmiert. Es unterstützt bis zu 24 Eingänge für Audiokanäle. Die Daten werden gebündelt und codiert übers Netzwerk an einen Server geschickt. Dieser empfängt gleichzeitig die Videodaten (das Screen Capturing) von der Working Position des Controllers. Die Daten werden in Archivdateien gespeichert und können später auf eine Workstation kopiert und von einem Player wiedergegeben werden.

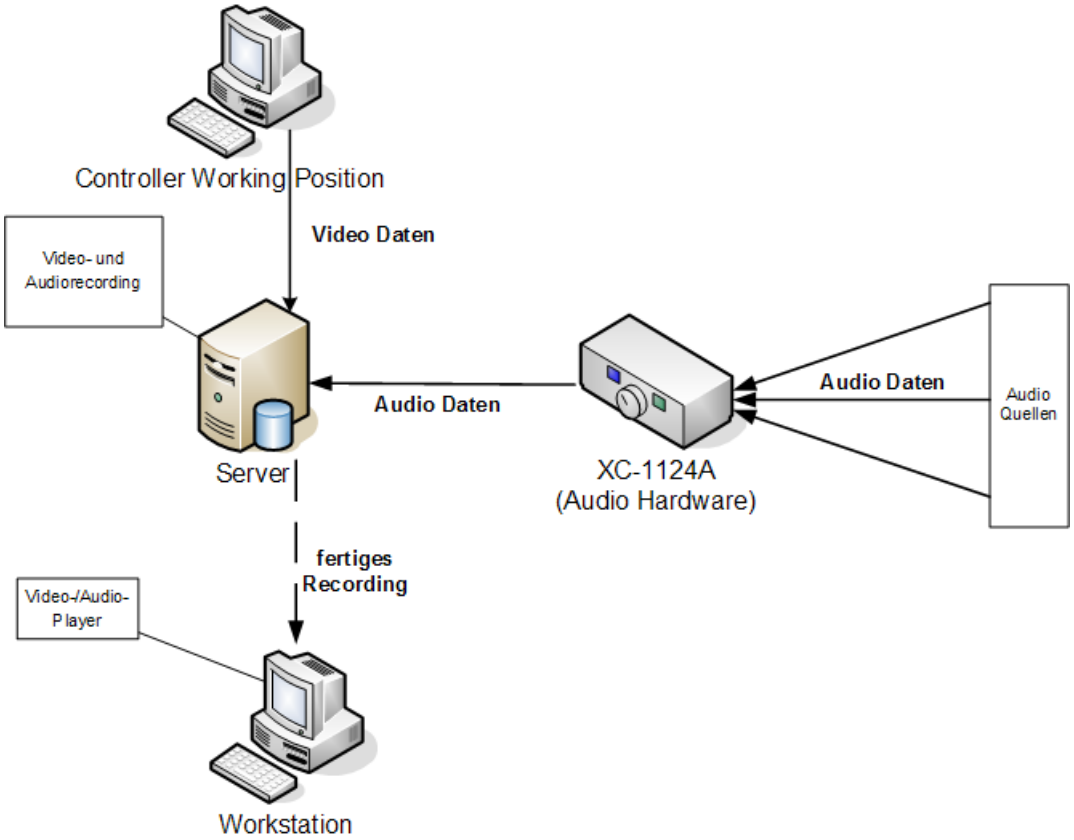

Abbildung 1.1: Konzeptionelle Architekturübersicht des Audio- / Videorecordings

Die notwendigen Arbeitsschritte des Projekts umfassten:

- Entwicklung eines geeigneten Datenprotokolls zur Übertragung der Audiodaten, welches die nachträgliche Synchronisation zum Videomaterial erlaubt.
- Entwicklung einer Software, welche die Audiodaten vom Netzwerk empfängt und in einem geeigneten Format lokal speichert (die Videorecording Software existiert bereits).
- Erweiterung des Videoplayers, so dass dieser zum Video auch Audiodaten von selektierten Kanälen abspielen kann und diese mit dem Video synchronisiert.

Die vorliegende Arbeit widmet sich nicht ausschließlich dem Thema Voice over IP bzw. Internet Telefonie. VoIP stellt in diesem Fall lediglich eine Möglichkeit dar, digitale Sprachsignale in Echtzeit über ein Computernetzwerk zu schicken. Das entwickelte Protokoll entspricht keinem VoIP Standard, sondern ist an die gegebenen Bedürfnisse angepasst. Daher wurde der Arbeitstitel mit *Audio over IP* bezeichnet.

[2] http://www.xerxes-tech.com

Die programmierte Software ist ein Prototyp mit kommerziellen Hintergrund. Das Ziel war es, herauszufinden wie man Software und Hardware entwickeln und einsetzen kann, so dass man dem Kunden in Zukunft eine Legal Recording Lösung anbieten kann. Der Prototyp kann zu Test- und Demonstrationszwecken eingesetzt werden. Kunden bestehen aber in der Regel noch auf eigene Anpassungen, sowie ausführliche technische Dokumentation und umfangreiche Abnahmetests.

Die Software arbeitet direkt mit der Audio-Hardware XC-1124A zusammen. Da das Gerät aber von einer externen Firma entwickelt wurde, wird in dieser Arbeit auf eine genaue Beschreibung der Hardware verzichtet. Gleiches gilt für das bereits vorhandene Videorecording.

1.2 Struktur der Arbeit

Die Arbeit beginnt mit einer kurzen Beschreibung der physikalischen Grundlagen der Akustik (siehe Kapitel 2). Es wird erläutert, was Schall ist, wie er entsteht und wie er sich ausbreitet. Das analoge (Schall-)Signal muss dann für die computergestützte Verarbeitung in eine digitales Signal gewandelt werden. Dazu wird in Kapitel 3 eine Begriffsdefinition von digitalen Signalen gegeben, das Prinzip von Analog-Digital Wandlern erklärt und Verfahren zur Analog-Digital Wandlung beschrieben. Ist das Signal also digitalisiert, kann es auf einem Computer gespeichert und verarbeitet werden.

Kapitel 4 beschreibt Eigenschaften von Audiodaten auf dem Computer, sowie Möglichkeiten der Speicherung, Komprimierung und Übertragung in digitalen Netzen. Das Kapitel widmet sich auch den Grundlagen des Voice over Internet Protocols und zeigt, warum VoIP für den Zweck des Projekts ungeeignet war.

Kapitel 5 widmet sich dann ausschließlich der implementierten Software. Zuerst werden die Anforderungen, die generelle Durchführung des des Projekts und das Datenprotokoll der Audio-Hardware beschrieben. Dann wird die Audio Recording Lösung im Form des Programms *AudioRec* erläutern. Danach folgt die Beschreibung zur Erweiterung des Media Players *AVPlayer*, damit dieser Video- und Audiodaten synchron abspielen kann. Schlussendlich werden die Testmethoden beschrieben, welche verwendet wurden, um Fehler in der Software zu finden.

Kapitel 6 schließt die Arbeit mit einer Zusammenfassung ab. Sie zeigt nochmals die wichtigsten Aspekte des Projekts auf. Ein Ausblick gibt Auskunft über zukünftige Arbeit und Forschung zu diesem Thema.

2 Grundlagen der Akustik

Dieses Kapitel beschreibt einleitend die Grundlagen der Akustik. Es widmet sich dem Thema Schall und erklärt wie Schall erzeugt wird, wie er sich ausbreitet und welche physikalischen Eigenschaften er besitzt.

2.1 Schall und Schallerzeugung

Für die Erzeugung von Schall und dessen Ausbreitung sind zwei Dinge erforderlich: eine Schallquelle und ein Trägermedium. Schallquellen sind im allgemeinen natürlicher oder technischer Art. Beim Menschen werden die Stimmlippen mit Luft aus dem Brustkorb zum Schwingen gebracht und im Rachen, der Mund- und der Nasenhöhle verstärkt [Ria02, S. 94]. Technische Schallquellen wandeln Energie in Schall um, Beispiele dafür sind Motoren, Maschinen, Werkzeuge, Hydraulik- und Pneumatikgeräte, aber auch Musikinstrumente [DEG06, S. A14]. Als Trägermedium kann alles dienen, was eine Ausbreitung von Schallwellen zulässt: feste, flüssige und gasförmige Körper. Das Medium muss elastisch sein - im Vakuum kann sich Schall nicht ausbreiten. Angeregt durch die Schallquelle fangen die nächstgelegenen Teilchen auch zum Schwingen an. Es entsteht dabei eine Verdichtung und Verdünnung im Trägermedium, eine Art Zieharmonikaeffekt, welcher sich weiter fortsetzt und je nach Medium mit fortlaufender Zeit abschwächt [Kre].

[Kre] definiert also:

> *Als Schall bezeichnet man die sich wellenartig ausbreitende räumliche und zeitliche Druckänderung eines elastischen Mediums.*

Schall wird vom Menschen unterschiedlich wahrgenommen. Er kann als angenehm (Naturlaute, Sprache, Musik, etc.) oder als unangenehm (Maschinenlärm, Motorgeräusch, etc.) wahrgenommen werden, wobei natürlich eine subjektive Komponente auch immer mitspielt. Im allgemeinen werden unangenehme Schallquellen aber als Lärmquellen bezeichnet, weil sie den Menschen belasten [DEG06, S. A14].

Der Schall kann auch durch verschiedene Phänomene gelenkt und verändert werden. Ähnlich wie in der Optik existieren Brechung, Reflexion, Beugung und Streuung [DEG06, S. 10]. Dies führt dazu, dass uns der Schall verfolgt, obwohl die erzeugende Quelle schon längst aus dem Blickfeld verschwunden ist. Andererseits werden diese Eigenschaften auch beim Schutz vor Lärmquellen benutzt. Insbesondere die Schalldämmung im städtischen Bereich, bei Maschinen und Motoren, sowie auf Autobahnen und Schnellstraßen ist ein wichtiger Bereich der technischen Akustik, um den Menschen vor zu viel Lärm zu schützen und die Lebensqualität zu erhöhen (siehe auch [Mös07, Kapitel 8 - 10]).

Um Schall besser zu verstehen widmet sich der nächste Abschnitt den physikalischen Grundlagen.

2.2 Physikalische Grundlagen

Möser [Mös07, S. 1] bezeichnet zwei Merkmale als maßgebend für den Schall: Klangfarbe (das Maß dazu ist die Frequenz f) und Lautstärke (welches der Schalldruck p ist). Riat [Ria02, S. 91] nennt dagegen drei Merkmale mit etwas anderer Bedeutung: die Intensität (Schalldruck), die Tonhöhe (Frequenz) und die Klangfarbe, welche dem menschlichen Ohr erlaubt Töne mit gleicher Intensität und Höhe dennoch voneinander zu unterscheiden. Riat [Ria02] ist damit aber schon einen Schritt voraus, denn er behandelt Töne, vor allem aus der Musik, welche zum Grundton zusätzliche Obertöne besitzen. Dadurch kann man ein Instrument vom anderen Unterscheiden, selbst wenn beide den gleichen Ton anspielen. Zum leichteren Verständnis sollen aber vorerst nur einzelne Töne (kein Klangspektrum) untersucht werden.

In Abbildung [2.1] sieht man die Verdichtung und Verdünnung des Schalldrucks in einem elastischen Medium über den Verlauf der Zeit. Die x-Achse markiert den Ruhedruck p_0 des Mediums. Es erfolgt eine abwechselnde Verdichtung und Verdünnung, welche mit zunehmender Zeit schwächer wird. Energie geht nach dem Energieerhaltungssatz natürlich nicht verloren, vielmehr wandelt sie sich in andere Energie (bspw. Wärme) um.

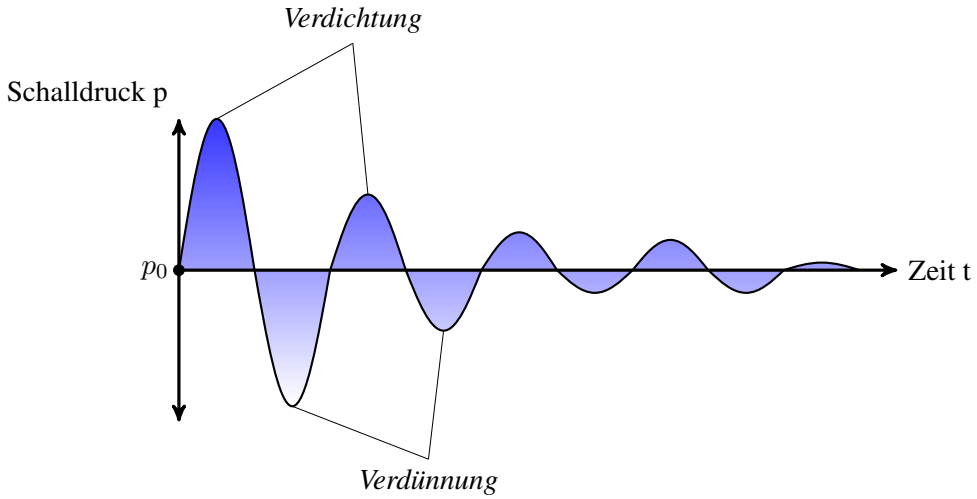

Abbildung 2.1: Schalldruck in einem elastischen Medium (nach [Kre])

Schall ist nichts anderes als eine periodische Schwingung. Lässt man das Ausklingen des Schalls beiseite, so wiederholt sich die Periode T für eine bestimmte Schwingung unendlich oft. Will man nun die Frequenz bestimmen, so bildet man den Kehrwert der Periode (vgl. [Ria02, S. 17]):

$$f = \frac{1}{T} \tag{2.1}$$

Die Einheit der Frequenz ist Hertz[1] (Hz), welche der Periode von einer Sekunde entspricht (siehe Formel [2.2]). Eine periodische Schwingung mit einem Hertz ist in Abbildung [2.2] gezeigt. Desweiteren ist aus Abbildung [2.1] auch ersichtlich, dass trotz Dämpfung sich die Frequenz nicht ändert (vgl. [Ria02, S. 19]).

$$\frac{1}{s} = 1\text{Hz} \tag{2.2}$$

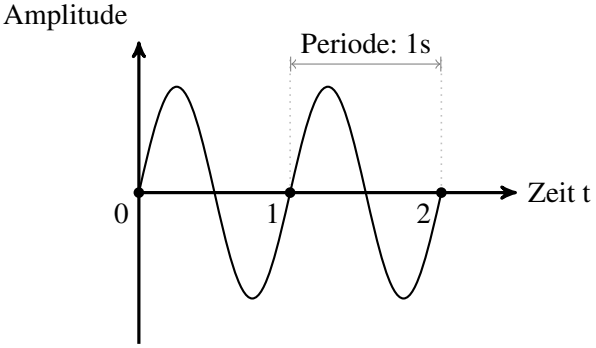

Abbildung 2.2: Eine periodische Schwingung mit einem Hz

Das menschliche Ohr umfasst einen Hörbereich von 16 bis 16000 Hz [Mös07, S. 1]. Die Angaben in der Fachliteratur schwanken jedoch. Setzer [Set] berichtet von einem Hörbereich von bis zu 20000 Hz.

[1] *Heinrich Hertz:* deutscher Physiker, * 22. Februar 1857, †1. Januar 1894

Mit zunehmenden Lebensalter nimmt jedoch der obere Bereich ab, so dass ältere Menschen keine sehr hohen Töne mehr hören können.

Natürlich gibt es auch Töne außerhalb des menschlichen Hörbereichs. Frequenzen unter dem Hörbereich werden als *Infraschall*, Frequenzen darüber als *Ultraschall* bezeichnet. Für die menschliche Sprache ist lediglich ein Bereich von 300 bis 3150 Hz notwendig [Set]. Doch auch diese Angaben sind unscharf, denn der Bereich kann sich durch aktive Beeinflussung (rufen, schreien) oder trainierte Stimmen verändern. Lazarus et al. [LSS+07, S. 51] geben grob einen Bereich von 100 bis 10000 Hz an. Dabei dürften hohe Bereiche aber nur kurzzeitige Spitzen darstellen. So entsprechen 10000 Hz bereits der Note D#9[2], welche das neunte D# in der musikalischen Notation darstellt. Der Kammerton A (A4) liegt bei 440 Hz. Auf einer Gitarre ist im Regelfall ein C7 noch zu erreichen, auf einem Klavier durchaus auch C8.

Bislang wurden nur einzelne Töne behandelt. In der Realität treffen jedoch eine nahezu unendliche Anzahl von Tönen aufeinander. In diesem Zusammenhang spricht man einerseits vom Klang, welcher einen zugrunde liegenden Grundton besitzt, und dem bereits erwähnten Geräusch, welches keine periodische Schwingungen mehr aufweist (vgl. [Set] und [Ria02, S. 16]). Ein Spezialfall des Geräusches ist das *weiße Rauschen*. Dabei umfasst der Frequenzbereich das gesamte menschliche Spektrum, wobei die Intensität linear mit der Frequenz ansteigt. Das weiße Rauschen ist reproduzierbar und dient akustischen Untersuchungen [HSF08, S. 61].

Nun sollte auch klar sein wie die Klangfarbe zustande kommt: Durch Überlagerung mehrere Frequenzen ist es dem Menschen möglich den Grundton (den tiefsten Ton) zu hören und gleichzeitig alle Obertöne. Die Summe, Ausprägung und Intensität derer erzeugt schließlich die Klangfarbe des Schall emittierenden Objekts. Besitzt ein Klang beispielsweise drei Obertöne zu seinem Grundton, so kann man beliebig viele Klänge durch Manipulation dieser drei Obertöne erzeugen - ohne die Frequenzen der Schwingungen selbst zu ändern! In Abbildung 2.3 sind vier Diagramme ersichtlich. Sie zeigen jeweils einen unveränderter Grundton (der linkeste Balken) und unterschiedlich ausgeprägte Obertöne. Dies ergibt jedes Mal eine andere Klangfarbe (vgl. [Kre]).

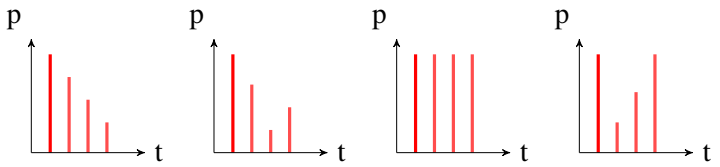

Abbildung 2.3: Verschiedene Klangfarben mit unverändertem Grundton

Es bleibt nun noch offen zu erklären, was der *Schalldruck*, die zweite wichtige Kenngröße von Schall, ist. Der Schalldruck ist eine Schallfeldgröße, denn seine Größe ist proportional zum Quadrat der Energie. Im Gegensatz dazu gibt es Schallenergiegrößen, welche proportional zur Energie sind (beispielsweise Schallleistung, Schallintensität) [Set]. Der Schalldruck p berechnet sich aus der wirkenden Kraft F je Fläche A:

$$p = \frac{F}{A} \tag{2.3}$$

Die Einheit des Schalldrucks ist Pascal[3] (Pa). Diese ist definiert als ein Kilogramm pro Meter mal Sekunde zum Quadrat bzw. ein Newton pro Meter zum Quadrat, was im Grunde der Formel 2.3 entspricht:

$$\frac{1kg}{m \cdot s^2} = \frac{1N}{m^2} = 1\text{Pa} \tag{2.4}$$

[2] Berechnung mit Hilfe von `http://www.phys.unsw.edu.au/music/note/`

[3] *Blaise Pascal:* französischer Mathematiker, Physiker, sowie Philosoph, * 19. Juni 1623, †19. August 1662

Schalldruck wird in der Regel jedoch nicht in Pa angegeben, sondern in Dezibel (dB, nach Alexander Bell[4]). Dazu gibt es den Schallpegel L_p welcher auf der Dezibel Skala aufgetragen ist. Diese ist logarithmisch aufgebaut und beginnt bei der Hörschwelle des Menschen. Diese entspricht 0 dB bzw. $2 \cdot 10^{-5}$ Pa [DEG06, S. 13]. Zur Berechnung des Schallpegels wird aus dem Verhältnis vom Quadrat des Effektivwertes des Schalldrucks $\tilde{p}$ zum Quadrat des Bezugsschalldrucks p_0 der dekadische Logarithmus gebildet [DEG06, S. 13]:

$$L_p = \lg \frac{\tilde{p}^2}{p_0^2}\,\mathrm{B} = 20 \lg \frac{\tilde{p}}{p_0}\,\mathrm{dB} \quad \text{wobei} \quad 1\mathrm{B} = 10\mathrm{dB} \tag{2.5}$$

p_0 ist, wie bereits definiert, $2 \cdot 10^{-5}$ Pa, der Effektivwert $\tilde{p}$ ist abhängig von der Zeit. $\tilde{p}$ ist das quadratische Mittel des gemessenen Schallwechseldrucks (vgl. [DEG06, S. 13] und [Mös07, S. 7]). Der Schallpegel berücksichtigt also bereits die Entfernung von der Schallquelle und die damit abnehmende Lautstärke. Bei der Angabe eines Schallpegelwertes ist aber dennoch auch immer die Entfernung zur Schallquelle anzugeben.

Vollständigkeitshalber sei hier auch noch die bereits angesprochene Schallintensität I erwähnt. Diese ist im Gegensatz zum Schalldruck eine Schallenergiegröße, sie ist proportional zur Energie. Die Schallintensität gibt an, wie viel Energie pro Quadratmeter in eine bestimmte Richtung fließt (vgl. [Set] und [DEG06, S. 15]). Dementsprechend wird die Schallintensität in Watt pro Quadratmeter angegeben. Sie berechnet sich aus dem Produkt von Schalldruck p und der Geschwindigkeit des Schalls $\vec{v}$ (auch als Schallschnelle bezeichnet):

$$\vec{I} = p \cdot \vec{v} \tag{2.6}$$

Auch die Schallintensität lässt sich mit Hilfe des Schallintensitätspegels L_I in Dezibel ausdrücken. Sein Bezugswert I_0 ist genormt und entspricht $10^{-12} \frac{W}{m^2}$. Er kann nach Formel 2.7 berechnet werden [DEG06, S. 15]). Damit lassen sich der Schallintensitätspegel und der Schalldruckpegel miteinander vergleichen, denn die Schallintensität ist proportional zum Quadrat des Schalldrucks.

$$L_p = 10 \lg \frac{\tilde{I}}{I_0}\,\mathrm{dB} \tag{2.7}$$

Wie bereits in der Einleitung des Kapitels definiert, ist Schall zeit- und ortsabhängig. Seine Ausbreitung ist im Idealfall kugelförmig von einer punktförmigen Schallquelle, wobei Faktoren wie das Schallmedium, die Schallschnelle und Hindernisse, die zu Brechung, Streuung und Reflexion führen können, eine wesentliche Rolle spielen. Abschnitt 2.3 behandelt im folgenden als weitere Grundlage die Schallausbreitung in der Atmosphäre.

2.3 Ausbreitung von Schall

Der wichtigste Sachverhalt im Rahmen dieser Arbeit ist die Ausbreitung von Schall in der Atmosphäre, also in Gasen. Daneben gibt es noch den Körperschall in Festkörpern, sowie den Flüßigkeitsschall. Die Ausbreitung in der Luft ist von Alltagswissen geprägt. Ein schönes Beispiel ist ein Feuerwerk bei dem man den Lichteffekt des in die Luft geschossenen Feuerwerkskörpers wesentlich früher sieht, als den Knall, der je nach Entfernung später zu hören ist. Wie man als Kind schon lernt, ist die Ausbreitung des Lichts einfach wesentlich schneller als jene des Schalls. Zusätzlich ist es aber auch möglich selbst mit geschlossenen Augen die ungefähre Richtung der Schallquelle zu bestimmen, sowie das Geräusch einzuordnen. Schall nimmt zwar während seiner Ausbreitung an Intensität ab, verändert dabei seine Frequenz bzw. seinen Klang aber nicht. Dies ist eine Eigenschaft eines nicht dispersiven[5] Mediums. Die Schall-

[4] *Alexander Graham Bell:* britischer Erfinder (Telefon) und Wissenschaftler, * 3. März 1847, †1. August 1922

[5] Dispersion: "Ausbreitung", "Zerstreuung", in diesem Sinne auch "Auseinanderlaufen"

schnelle ist unabhängig von der Frequenz, somit verändert sich die die Struktur des Schalls während der Ausbreitung nicht [Mös07, S. 19].

In einem Schallfeld nimmt die Schallintensität proportional zur Entfernung zur Schallquelle ab. Da die Schallintensität Leistung P pro Fläche A entspricht und das Schallfeld in dieser Betrachtung eine Kugel ist, entspricht die Fläche der Oberfläche einer Kugel. Die Schallintensität für einen Schallquelle mit Abstandsradius r entspricht also der Formel (vergleiche dazu auch Abbildung 2.4):

$$I = \frac{P}{4 \cdot \pi \cdot r^2} \tag{2.8}$$

Da diese Ausbreitung zeitgebunden ist, ergeben sich eine Vielzahl von Abstandsradien $r_{1...n}$ und daraus eine Vielzahl an Schallintensitätswerten $I_{1...n}$. Es besteht also das Verhältnis (vgl. [Set] und [HSF08, S. 215ff]):

$$\begin{aligned} \frac{I_1}{I_2} &= \frac{r_1^2}{r_2^2} \\ I_1 &= I_2 \cdot r_2^2 \cdot \frac{1}{r_1^2} \\ I &\sim \frac{1}{r^2} \end{aligned} \tag{2.9}$$

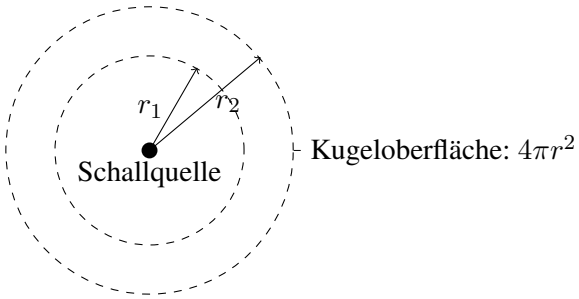

Abbildung 2.4: Kugelförmige Schallausbreitung von einer punktförmigen Schallquelle (vgl. [Mös07, S. 72])

In der Praxis wird der Schallintensitätswert auch von anderen Faktoren abgeschwächt. Dazu zählen die Luftabsorbtion, Bodenabsorption, bauliche und natürliche Elemente die eine Ausbreitung erschweren. Durch Reflexionen ist es andererseits möglich, dass sich die Intensität erhöht [Set]. Henn et al. [HSF08, S. 310ff] geben Berechnungsgrundlagen und allgemeine Hinweise zur Schallabsorption eines Raumes an.

Für diese Arbeit ist die Schallabsorption jedoch zu vernachlässigen, da beim Funk der Schall sofort von einem Mikrofon aufgenommen wird und es zur Entfernung zum Sprecher keine Hindernisse zu erwarten sind. Das nächste Kapitel widmet sich somit den elektrotechnischen Grundlagen zur Signalwandlung, denn das analoge Signal des Sprechers muss in ein digitales Signal, welches elektronisch weiterverarbeitet werden kann, umgewandelt werden.

3 Signalverarbeitung

Nachdem in Kapitel 2 der Schall auf Basis der Physik erklärt wurde, stellt Kapitel 3 die Wandlung von analogen Signalen in digitale dar. Dafür wird zuerst der Begriff *digitales Signal* erklärt, danach folgt die Beschreibung der analog nach digital Wandlung sowie bekannte Realisierungsverfahren.

3.1 Digitale Signale

In der Informationstechnologie wird fast ausschließlich nur noch mit digitalen Signalen gearbeitet. Sie bieten zahlreiche Vorteile gegenüber analogen Signalen und die Nachteile digitaler Signale können durch gestiegene Verarbeitungsgenauigkeit und -effizienz beinahe aufgehoben bzw. vernachlässigt werden. Ein großer Vorteil ist die Reproduzierbarkeit und Weiterverarbeitbarkeit von digitalen Signalen. Zwar lassen sich analoge Signale durchaus auch verarbeiten, transportieren und speichern, doch bei digitalen Signalen ist der Prozess wiederholbar und im Allgemeinen verlustfrei. Ein Beispiel dazu ist die Aufnahme von Sprache auf einem analogen und einem digitalen Speichermedium. Je nach Equipment wird man auf beiden Speichermedien anfänglich immer einen gewissen Informationsverlust haben, dieser ist aber meist tolerierbar. Die Weiterverarbeitung danach ist aber gravierend: Während beim analogen Medium jeder Bearbeitungsschritt immer eine Reinterpretation des Signals zur Folge hat und dieses somit verändert (meist auf Kosten der Qualität), ist es kein Problem das digitale Signal unverändert zu reproduzieren. Dies gilt sowohl für das Übertragen (kopieren, senden, etc.) als auch für die Verarbeitung.

Grüningen [Grü08, S. 12f] nennt weitere Vorteile der digitalen Signalverarbeitung. Allgemein sind Systeme zur digitalen Signalverarbeitung nicht von Schwankungen der Temperatur betroffen, sie sind über längere Zeit stabil, haben eine höhere Zuverlässigkeit und eine geringere Störempfindlichkeit. Zusätzlich weist die digitale Signalverarbeitung spezielle Bereiche auf, die nur so bearbeitet werden können bzw. deren Bearbeitung wesentlich verbessert werden kann. Dazu zählen Filter mit linearem Phasengang, adaptive Filter, die sich selbstständig anpassen, erweiterte Möglichkeiten bei der Musikabmischung, Bild- und Videoverarbeitung, Sprachverarbeitung und Datenkompression. Besonders die zwei letztgenannten Punkte sind für diese Arbeit relevant.

Nachteile nach Grüningen [Grü08, S. 14f] sind beispielsweise ein zusätzlicher Schaltungsaufwand. Man benötigt einen Analog-Digital Wandler (siehe dazu auch Abschnitt 3.2), um das Signal vorerst zu digitalisieren und dann zu verarbeiten. Zur Wiedergabe auf einem Bildschirm oder mittels Lautsprechern wird wiederum einen Digital-Analog Wandler benötigt. Dabei ist es weniger der Qualitätsverlust, der mit moderner Technik kaum mehr ins Gewicht fällt, sondern Komponenten wie Verarbeitungszeit und zusätzlicher Bedarf an Bauteilen, welche Platz und Strom benötigen und Kosten verursachen. Ein für im Rahmen dieser Arbeit zu vernachlässigender Nachteil ist die Verarbeitung von hochfrequenten Signalen (Signale über 20000 kHz). Die Hardware muss aus sehr schnelle Bausteinen bestehen um eine geringe Zykluszeit zu erreichen, damit ein so hochfrequentes Signal noch abgetastet werden kann.

Was unterscheidet nun ein digitales von einem analogen Signal? Ein digitales Signal besteht aus diskreten, also unterschiedlichen und voneinander getrennten Werten, typischerweise repräsentiert als 0er und 1er. Ein analoges Signal dagegen besitzt kontinuierliche, stufenlose Werte und kann von physikalischen Veränderungen beeinflusst werden [Tel, Stichworte: Analog Signal].

Zur Repräsentation von digitalen Signalen dienen Zahlensysteme der Basis B [WU07, S. 1ff]. Das alltägliche *Dezimalsystem* hat die Basis B = 10, das heißt jede Zahl wird nach Potenzen von B zerlegt. Bei dem Dezimalsystem gibt es pro Stelle (pro Ziffer) zehn Ausdrucksmöglichkeiten, das sind die Ziffern 0 bis 9. Das Dezimalsystem ist aber unpraktisch für die Darstellung von digitalen Signalen, da es in der Elektrotechnik wesentlich einfacher ist nur zwei Zustände darzustellen (ja/nein, true/false, offen/ge-

schlossen, etc.). Außerdem lässt sich das boolesche[1] Algebra, welches die logischen Operatoren UND, ODER und NICHT enthält, darauf anwenden. Aus diesem Grund gibt es das *Dualsystem* zur Basis B = 2, auch bekannt als das *Binärsystem*. Es besteht aus den bereits angesprochenen 0er und 1er, pro Stelle lassen sich genau zwei Zustände darstellen. Aus Gründen der Leserlichkeit für den Menschen ist das Dualsystem aber sehr unpraktisch, besonders bei längeren Zahlen. In der Informatik ist auch das *Hexadezimalsystem* gebräuchlich, welches die Zahl 16 zur Basis hat. Die Zahlennotation wurde dabei um die ersten sechs Buchstaben des Alphabets (A, B, C, D, E, F) erweitert, diese repräsentieren daher die Zahlen 10 bis 15.

Beispiel: Die Zahl 180 im Dezimalsystem entspricht 10110100 im Dualsystem und B4 im Hexadezimalsystem.

Es gibt eine enge Verschränkung des Dualsystems mit dem Hexadezimalsystem. So lassen sich Byte Werte, welche bekanntermaßen aus acht Bit bestehen, im Hexadezimalsystem platzsparend darstellen. Ein Byte benötigt acht Stellen in der Binärnotation, aber nur zwei in der Hexadezimalnotation. Hat man einmal die Umrechnung von einer Hexadezimalzahl zu einer Dualzahl im Kopf, lassen sie sich so wesentlich schneller erfassen als eine lange Zahlenreihe von 0ern und 1ern.

Schließlich sei noch die vorher angesprochene boolesche Algebra erwähnt. In der Literatur findet man häufig die Notation $\wedge$ für die Konjunktion UND (AND), $\vee$ für die Disjunktion ODER (OR) und $\neg$ für die Negierung NICHT (NOT). UND und ODER sind zweistellige Verknüpfungen, NICHT ist einstellig und invertiert das Element. Die Wahrheitstabellen dieser Grundfunktionen sind in Tabelle 3.1 aufgelistet.

A	B	A $\wedge$ B	A $\vee$ B
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

A	$\neg$A
0	1
1	0

Tabelle 3.1: Wahrheitstabelle der logischen Verknüpfungen UND, ODER und NICHT

Desweiteren existieren auch logische Gatter, welche eine Verknüpfung der Grundfunktionen darstellen. NAND beispielsweise konjugiert die Elemente und invertiert das Ergebnis, demgegenüber disjungiert ein NOR die Elemente und invertiert das Ergebnis. Das XOR-Gatter ist ein ausschließendes (bzw. exklusives) ODER. Es konjugierte jeweils A und $\neg$B und $\neg$A und B, das Ergebnis davon wird disjungiert. Als Hilfestellung sind die Formeln dieser drei Gatter in Formel 3.1 zu finden und deren Wahrheitstabellen in Tabelle 3.2. Weiterführendes zur booleschen Algebra ist in Schubert [Sch09, Kapitel 11] nachzulesen.

$$\begin{aligned} \text{NAND} &: \neg(A \wedge B) \\ \text{NOR} &: \neg(A \vee B) \\ \text{XOR} &: (\neg A \wedge B) \vee (A \wedge \neg B) \end{aligned} \tag{3.1}$$

[1] *George Boole:* englischer Mathematiker und Logiker, * 2. November 1815, †8. Dezember 1864

A	B	A NAND B	A NOR B	A XOR B
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	0	0	0

Tabelle 3.2: Wahrheitstabelle der logischen Gatter: NAND, NOR, XOR

Der nächste Abschnitt widmet sich den Analog-Digital Wandlern, erklärt die Grundlagen und beschreibt die drei prinzipiellen Schritte der Umwandlung vom analogen ins digitale Signale.

3.2 Analog-Digital Wandler

Der grundsätzliche Wunsch ist es ein analoges Signal möglichst originalgetreu in eine digitales umzusetzen. Dazu werden *Analog-Digital Wandler* (auch bekannt als *Analog-Digital Umsetzer* (ADU), bzw. (engl.) *Analog-to-Digital Converter* (ADC)) verwendet. Woitowitz et al. [WU07, S. 283] nennen als Einsatzgebiete Messwandler (Druck, Temperatur, Beschleunigung, etc.), Mikrophone und Videokameras. Sie alle stellen eine Schnittstelle von der analogen in die digitale Welt dar. Dabei ist aber offensichtlich, dass kein digitales Bild und auch kein digitaler Ton an das heranreicht, was der Mensch in der Realität wahrnimmt. Als technische Probleme nennen Woitowitz et al. [WU07, S. 283] daher die Anforderungen in die Genauigkeit und die Geschwindigkeit. Diese zwei Dinge hängen eng zusammen. Präzession erfordert meist sehr leistungsfähige (und daher auch oftmals sehr teure) Bauteile und nimmt Zeit, sprich Rechenzeit, in Anspruch. Für das Konzept von Analog-Digital Wandler in technischen Geräten beschreitet man also meist einen Zwischenweg aus dem was wirtschaftlich vernünftig, technisch realisierbar und als Endergebnis akzeptierbar ist.

Nicht alle analogen Signale müssen notwendigerweise in bestmöglicher und technisch realisierbarer Genauigkeit umgesetzt werden. Schon allein aus Kostengründen sollte im Vorhinein geklärt werden, welche Anforderungen ein bestimmter Analog-Digital Wandler besitzten und welche Ergebnisse dieser produzieren soll. Lohninger [Loh08] nennt folgende maßgebende Parameter:

- **Scheitelwert der Amplitude:** Betrifft die Verstärkung von kleinen bzw. die Abschwächung von großen Signalen. Der Scheitelwert muss je nach Anwendungsfall richtig gewählt werden.
- **Dynamik des Signals:** Der Unterschied zwischen der größten und der kleinsten Amplitude ist die Dynamik. Je nach Umfang der Dynamik braucht der Wandler eine notwendige Auflösung um das Signal zu erfassen. Die Auflösung wird in Bit angegeben, welche der Analog-Digital Wandler am Ausgang liefert [Lat98, S. 2].
- **Frequenzspektrum des Signals:** Jedes Geräusch weist, wie in Kapitel 2 bereits behandelt, ein gewisses Frequenzspektrum auf. Dieses muss möglichst genau angenähert werden, da es sonst zu Artefakten im digitalen Signal kommt. Meist begnügt man sich jedoch damit das kleinste Detail im Signal noch wiedergeben zu können.
- **Störungen:** Störungen können sowohl vom analogen Signal selbst kommen, als auch von außen eingebracht werden.

Bei der Signalwandlung sind die drei nachfolgend aufgelisteten Schritte notwendig [WU07, S. 287]. Diese werden in den Unterabschnitte 3.2.1 bis 3.2.3 behandelt. Zur besseren Verständnis ist zusätzlich ein Blockschaltbild eines Systems zur Digitalisierung von analogen Signalen in Abbildung 3.1 angegeben.

1. **Abtastung:** Hält das Analogsignal für den A-D Wandler kurzzeitig konstant.
2. **Quantisierung:** Wandelt zeit- und wertkontinuierliche Signale in zeit- und wertdiskrete Signale um.
3. **Codierung:** Liefert die digitalen Informationen in einem bestimmten Code.

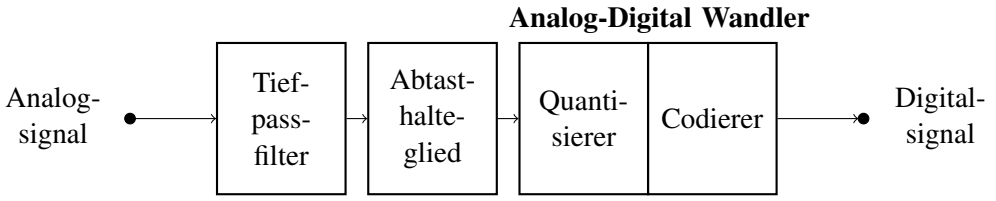

Abbildung 3.1: System zur Digitalisierung (vgl. [WU07, S. 287])

3.2.1 Abtastung

Das Abtasten wird von der *Abtast- und Halteschaltung*, auch *Abtasthalteglied*, (engl.: *Sample and Hold*) erledigt. Als wichtigste Grundlage für diesen Schritt gilt das Abtasttheorem, welches Claude Shannon[2] formalisiert hat:

> *If a function $f(x)$ contains no frequencies higher than ω_{max} (in radians per second), it is completely determined by giving its ordinates at a series of points spaced $T = \frac{\pi}{\omega_{max}}$ seconds apart.* [Uns00]

Im Zitat ist ω die Kreisfrequenz, für die gilt: $\omega = 2\pi f$. Dennoch gilt selbiges für die Frequenz f. Hat man also ein Signal mit maximal f_{max} Hz, so ist eine Abtastrate größer als $2 \cdot f_{max}$ Hz notwendig, um das Signal vollständig zu rekonstruieren. Die Voraussetzung dafür ist also, dass das Signal auf f_{max} Hz bandbegrenzt ist. Sollte dies nicht der Fall sein, so muss vor dieser Abtast- und Halteschaltung noch eine analoge Tiefpassfilterung durchgeführt werden (vgl. [WU07, S. 287f]). Der Tiefpassfilter filtert alle hochfrequenten Anteile im Signal heraus, indem er alle Frequenzen über einer Grenzfrequenz (bspw. 44,1 kHz) abschneidet.

Die Funktion des Abtasthalteglieds ist das Abtasten des analogen Signals zum Zeitpunkt t und das Zwischenspeichern des aktuellen Wertes $s(t)$ bis der dahinterstehende Analog-Digital Wandler diesen umgesetzt hat. Gäbe es kein Abtasthalteglied, müsste der A-D Wandler die Umsetzung wesentlich schneller durchführen und bräuchte somit entsprechend mehr Rechenleistung [WU07, S. 290]. Das Abtasten der Werte ist in Abbildung 3.2 verdeutlicht. Das erste Bild zeigt das analoge Eingangssignal als eine Kurve, das zweite die eruierten Abtastwerte zum Zeitpunkt t.

[2] *Claude Elwood Shannon:* amerikanischer Mathematiker und Elektrotechniker, * 30. April 1916, †24. Februar 2001

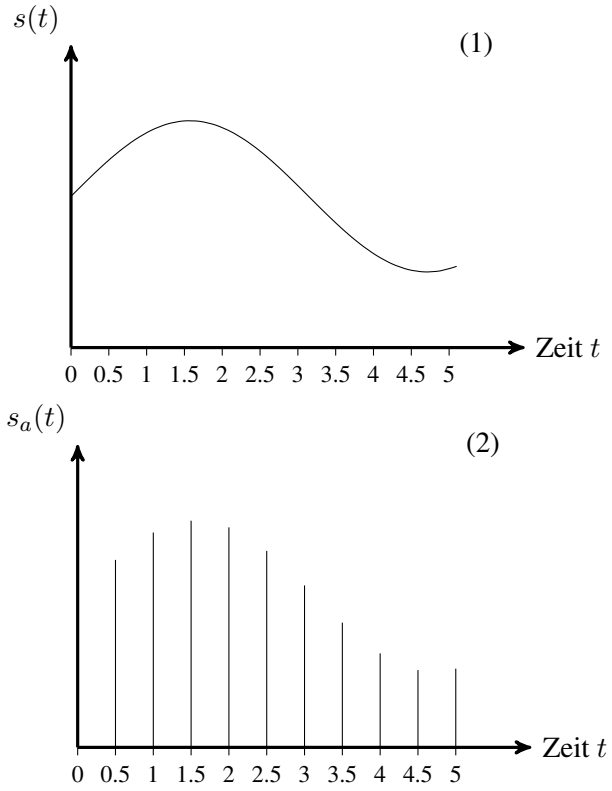

Abbildung 3.2: Abtastung eines analogen Signals (vgl. [Ahl07, S. 148])

3.2.2 Quantisierung

Die Quantisierung erfolgt im A-D Wandler selbst und setzt den bereits begrenzten Wertebereich in einen digitalen, ebenso begrenzten Wertebereich um. Es entstehen also gleich große *Quantisierungseinheiten*, welche einen bestimmten Bereich des Analogsignals abbilden. Überträgt man die dabei entstehenden Werte in ein Diagramm, so erhält man eine Stufenfunktion, welche eine Annäherung an das analoge Eingangssignal darstellt. Ein 4 Bit A-D Wandler besitzt $2^4 = 16$ Quantisierungsstufen, oder allgemein: der maximale Signalbereich eines n-Bit Wandlers beträgt $m = 2^n$ [Hir03, S. 1].

Da das analoge Signal unendlich viele Werte in einem Bereich annehmen kann, bei der Umwandung aber nur auf endlich viele digitale Werte abgebildet werden, entsteht ein *Quantisierungsfehler* [HBG05, S. 382]. Der Fehler kann minimiert werden, indem man eine höherer Auflösung (Bit) wählt.

Im nachfolgendem Bild ist das Prinzip der Quantisierung verdeutlicht, sowie der Quantisierungsfehler erkennbar (die Zahlen sind willkürlich gewählt). Die gepunktete Linie stellt die ideale Quantisierungskennlinie dar. Hirt [Hir03, S. 9] bezeichnet den Bereich als Unsicherheit, in dem ein Analogwert einem Digitalwert zugeordnet werden kann.

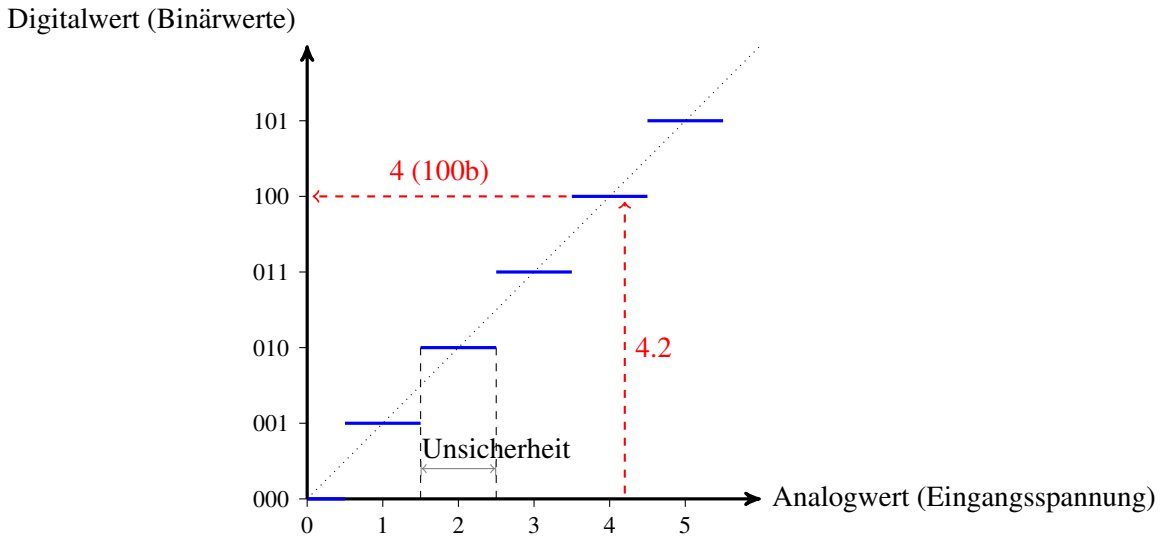

Abbildung 3.3: Quantisierung: Umsetzung von analogen in digitale Werte (vgl. [Loh08] und [Lat98, S. 1])

3.2.3 Codierung

Sind die Werte digitalisiert, so bedarf es der geeigneten Codierung, um diese auch nachher korrekt weiterverarbeiten zu können. Es wird zwischen *unipolarem* und *bipolarem* Code unterschieden [WU07, S. 299]. Erstere decken nur positive Signale ab, zweitere positive und negative. Gebräuchlich ist dabei laut Lattmann [Lat98, S. 3] die Zweierkomplement-Darstellung. Diese Darstellung entspricht dem Dualsystem (siehe 3.1) und besitzt eine vorgestelltes Vorzeichen Bit - 0 bei positiven Zahlen, 1 bei negativen. In einem n-Bit breitem Speicher können somit nur $n - 1$ Stellen für die Zahl selbst belegt werden, da die n-te Stelle für das Vorzeichen verwendet werden muss. Die Zahl -5 wird in der Zweierkomplement-Darstellung in einem 8 Bit breiten Register als 10000101 dargestellt. Die Kenntnis über die richtige Codierung ist also unumgänglich, denn ohne Vorzeichen würde diese Zahl als 133 interpretiert werden.

Der nächste Abschnitt behandelt Verfahren zur Analog-Digital Wandlung. Sie geben Einfluss auf den Aufbau und die Genauigkeit des A-D Wandlers, der je nach Art des Verfahrens weniger oder mehr Strom verbraucht, billige oder treuere Elemente benötigt oder genau bzw. weniger genau arbeitet.

3.3 Verfahren zur Analog-Digital Wandlung

Woitowitz et al. [WU07] teilen die Verfahren zur Analog-Digital Wandlung (auch oft als *Realisierungen* bezeichnet) in **direkte** und **indirekte** Verfahren. Bei direkte Verfahren wird das analoge Signal wird direkt gemessen. Beispiele dafür sind das Parallelverfahren, das Wägeverfahren und das Zählverfahren. Bei indirekte Verfahren wird das analoge Signal wird nicht direkt gemessen, sondern in eine Hilfsgröße überführt, bspw. Zeit oder Frequenz. Ein Beispiele dafür ist das Sägezahnverfahren. Die Verfahren werden in den Unterabschnitten 3.3.1 bis 3.3.4 besprochen.

3.3.1 Parallelverfahren

Ein Wandler, der das *Parallelverfahren* umsetzt, wird auch als *Flash-Wandler* bezeichnet [HBG05, S. 389]. Diese besitzen parallel angeordnete Komparatoren[3], welche eine logische 1 liefern, wenn das anliegende Signal größer ist als ihr Schwellenwert und eine 0, wenn es kleiner ist. Beispielsweise besitzt

[3] Ein *Komparator*, wie schon sein Name verdeutlicht, vergleicht zwei Werte miteinander (bspw. auf größer als oder kleiner als) und liefert ein dementsprechendes boolesches Ergebnis.

ein Flash-Wandler mit einer Auflösung von 4 Bit 16 Quantisierungsstufen und benötigt somit 15 Komparatoren. Dort, wo der eine Komparator noch eine 1, aber der nächste schon eine 0 geliefert hat, ist der Digitalwert zu finden, welcher von einer Dekodierlogik eruiert wird (vgl. [Lat98, S. 5], [Hir03, S. 4] und [WU07, S. 301ff]).

Durch diese parallel arbeitenden Komparatoren kann ein Flash-Umwandler folglich einen Digitalwert pro Taktperiode erzeugen. Der Aufwand dafür ist aber hoch und erfordert viele Bauteile. Laut Woitowitz et al. [WU07, S. 303] ist das Verfahren auf eine Auflösung von 12 Bit technisch beschränkt.

3.3.2 Zählverfahren

Im Gegensatz zum Parallelverfahren arbeitet das *Zählverfahren* seriell. Ein Komparator erhöht so lange seinen Schwellenwert bis der analoge Eingangswert überschritten wird. Durch Abzählen bzw. Aufsummieren der notwendigen Schritte bis zur Überschreitung kann der entsprechende Digitalwert ermittelt werden. Das Zählerverfahren gilt als langsam, dafür ist aber die Schaltung einfacher zu realisieren (vgl. [Lat98, S. 6] und [WU07, S. 308f]). Es ist aber zu erwähnen, dass die Effektivität vom Eingangssignal abhängt. Im besten Fall benötigt das Zählverfahren einen Rechenschritt, im schlechtesten $m-1$ Rechenschritte [Hir03, S. 4].

3.3.3 Wägeverfahren

Das *Wägeverfahren* arbeitet auch seriell, aber effektiver als das Zählverfahren, da es einen besseren Algorithmus nutzt. In jeder Taktperiode wird eine Stelle des n-stelligen digitalen Codeworts bestimmt. Dadurch benötigt das Verfahren unabhängig vom Eingangssignal genau n Rechenschritte [Hir03, S. 5].

Ein n-Bit Wandler besitzt n Normale der Form 2^i wobei $i = 0 \ldots n-1$. Der Algorithmus ist folgender (vgl. [Lat98, S. 7], [WU07, S. 303ff] und [Hir03, S. 5]):

- Für alle $j = n-1 \ldots 0$ vergleiche das Eingangssignal x mit dem aktuellen Normal j.
  - Wenn x größer-gleich j ist, liefere eine 1 für die Stelle b_j des digitalen Codeworts. Das Normal j bleibt erhalten.
  - Ansonsten liefere eine 0 für die Stelle b_j und entferne das Normal j.
- Wenn alle Normale j überprüft wurden ist die Binärzahl b fertig.

Ein Beispiel für einen 4-Bit Wandler ist in Tabelle 3.3 gezeigt. Für das Eingangssignal gilt $x = 5,5$.

Rechenschritt r	**aktuelles Normal** j	**Vergleich**	**binäres Codewort** b	**Normal behalten**
1	$2^{4-1} = 8$	$x < 8 \Rightarrow 0$	0...	nein
2	$2^{4-2} = 4$	$x >= 4 \Rightarrow 1$	01..	ja
3	$4 + 2^{4-3} = 6$	$x < 6 \Rightarrow 0$	010.	nein
4	$2^{4-4} = 1$	$x >= 1 \Rightarrow 1$	0101	ja

Tabelle 3.3: Rechenbeispiel für das Wägeverfahren, für das Eingangssignal gilt $x = 5,5$.

3.3.4 Sägezahnverfahren

Das *Sägezahnverfahren* (in seiner einfachsten Form auch als *Single-Slope Verfahren* bezeichnet) ist eine sehr einfache, indirekte Realisierung zur Umwandlung. Es arbeitet vergleichsweise inakkurat und ist langsam [WU07, S. 320].

Das analoge Eingangssignal U_e wird mit dem Ausgangssignal U_{ref} eines Sägezahngenerators verglichen. U_{ref} nimmt dabei jede Größe bis hin zu U_e im Zeitintervall Δt an. In jedem Takt eines Taktgebers liefert dieser Vergleich eine 1, diese wird kontinuierlich in einem Zähler aufsummiert. Das Ergebnis

steht am Ende des Vorgangs als digitaler Wert zur Verfügung. U_{ref} wird wieder auf das Ausgangssignal zurückgesetzt, der Zähler auf 0. Wie man sieht ist Δt in einem solchen Vorgang proportional zu U_e und das Ergebnis wird letztendlich durch simples Abzählen erreicht (vgl. [WU07, S. 316f] und [Hir03, S. 44ff]).

Natürlich stehen noch eine Menge weitere Realisierungsverfahren zur Verfügung, welche teilweise auch Verbesserungen der hier besprochenen darstellen. Dazu gehören das erweiterte Parallelverfahren, das erweiterte Zählerverfahren sowie Dual-, Quad- und Multi-Slope Verfahren. Zusätzlich existieren patentiere Verfahren wie das PREMA Mehrflankenverfahren [Hir03, S. 55ff] oder durch verbesserte Halbleitertechnologien möglich gewordene Realisierungen wie das Sigma-Delta Verfahren [HBG05, S. 392]. Diese Arbeit beansprucht jedoch keine Vollständigkeit auf Aufzählung von diversen Analog-Digital Umwandlungsverfahren. Stattdessen wird im nächsten Kapitel erklärt, wie ein bereits digitalisiertes Audiosignal auf einem Computer gespeichert, verarbeitet und übertragen werden kann.

4 Speicherung und Übertragung von digitalen Audiodaten auf dem Computer

Mit einem PC lassen sich digitale Audiodaten denkbar einfach erfassen, verwenden und weiterverarbeiten. Festplattenkapazitäten sind mittlerweile so groß, dass sich Aufzeichnung stunden- und tagelang durchführen lassen, ohne dass der Speicherplatz zur Neige geht. Der erste Abschnitt 4.1 widmet sich vorerst den Eigenschaften digitaler Audiodaten. Abschnitt 4.2 behandelt dann das Speichern und Komprimieren. Die Kompression ist nicht nur bedeutend, um den vorhandenen Speicherplatz noch etwas mehr auszunutzen, sondern auch um etwas weniger Daten im Falle einer Übertragung verschicken zu müssen. Die Übertragung von digitalen Audiodaten ist in Abschnitt 4.3 zu finden. Dort wird auch die Voice over IP Technik behandelt. Den Abschluss bildet eine Resümee und eine Diskussion der Übertragungstechniken. Schlussfolgerungen aus der Planungsphase des Projekts werden wiedergegeben, welche schließlich zur eigenen Lösung führten.

4.1 Eigenschaften von digitalen Audiodaten

Wie bereits in Abschnitt 3.2 beschrieben, wird ein analoges Signal durch Abtastung, Quantisierung und Codierung in digitale Werte gewandelt. Dieses Verfahren wird als *Pulse Code Modulation* (PCM) bezeichnet (siehe auch [WW01]). Digitale Audiodaten werden durch diesen Prozess genormt und besitzen folgende fixe Eigenschaften:

- **Abtastrate:** Wird auch als *Sampling Rate* bezeichnet. Darunter versteht man die Rate, wie oft das Signal abgetastet wurde. Der Wert wird in Hertz angegeben. Eine Audio CD besitzt eine Abtastrate von 44.100 Hz, die Tonspur auf DVDs meist schon 48.000 Hz und mehr. Für Sprachaufnahmen sind dagegen lediglich 8.000 Hz notwendig.
- **Samplingtiefe:** Gibt den Umfang von Werten an, welcher während des Abtastens erreicht werden kann. Dies ist somit auch der Dynamikumfang der Audiodaten. Eine Tiefe von 8 Bit bedeutet lediglich einen Umfang von 256 möglichen Werten, während mit 16 Bit (Audio CD Standard) ein Umfang von 65.536 Werten erreicht werden kann.
- **Kanäle:** Die Anzahl an Quellen in den Audiodaten. Bei Musikaufnahmen hat man meist zwei Quellen (Stereo), bei der Sprache reicht eine (Mono: der Sprecher).

Durch die Angabe dieser Werte lässt sich die Datenrate berechnen (siehe Formel 4.1). Dieser Wert wird in Kilobit pro Sekunde (kbps) angegeben und ist eine relevante Größe bei der Verarbeitung. Außerdem spiegelt die Datenrate auch die Qualität der Audiodaten wider, da eine größere Datenrate eine höhere Abtastrate und Samplingtiefe impliziert.

$$\text{Datenrate (kbps)} = \text{Abtastrate in Hz} \cdot \text{Samplingtiefe in Bit} \cdot \text{Kanäle} \tag{4.1}$$

Geht man davon aus, dass die menschliche Sprache auf maximal 4.000 Hz beschränkt ist [SKM+06, S. 71], so ergibt sich nach dem Abtasttheorem eine notwendige Abtastrate von 8.000 Hz. Unkomprimierte Sprach-Audiodaten mit einer festen Bitrate von 8 Bit und einem Kanal besitzen also eine Datenrate von 64 kbps: 8.000 Hz · 8 Bit · 1.

Die grafische Darstellung von Audiodaten erfolgt meist in zweidimensionaler Ansicht. Die x-Achse gibt die Zeit an, die y-Achse den Ausschlag der Amplitude. Die dargestellt Kurve wird als *Waveform* bezeichnet. Grafische Audiobearbeitungsprogramme zeigen diese in der Regel im Hauptfenster der Applikation an und ermöglichen dem Benutzer den Ton zu manipulieren und abzuspielen. In Abbildung 4.1[1] sind zwei Töne als Waveform dargestellt. Die erste Spur zeigt einen niedrigfrequenten Sinuston, der seine Lautstärke ändert. Die zweiter Spur zeigt einen regelmäßigen Stereoton, der leiser wird.

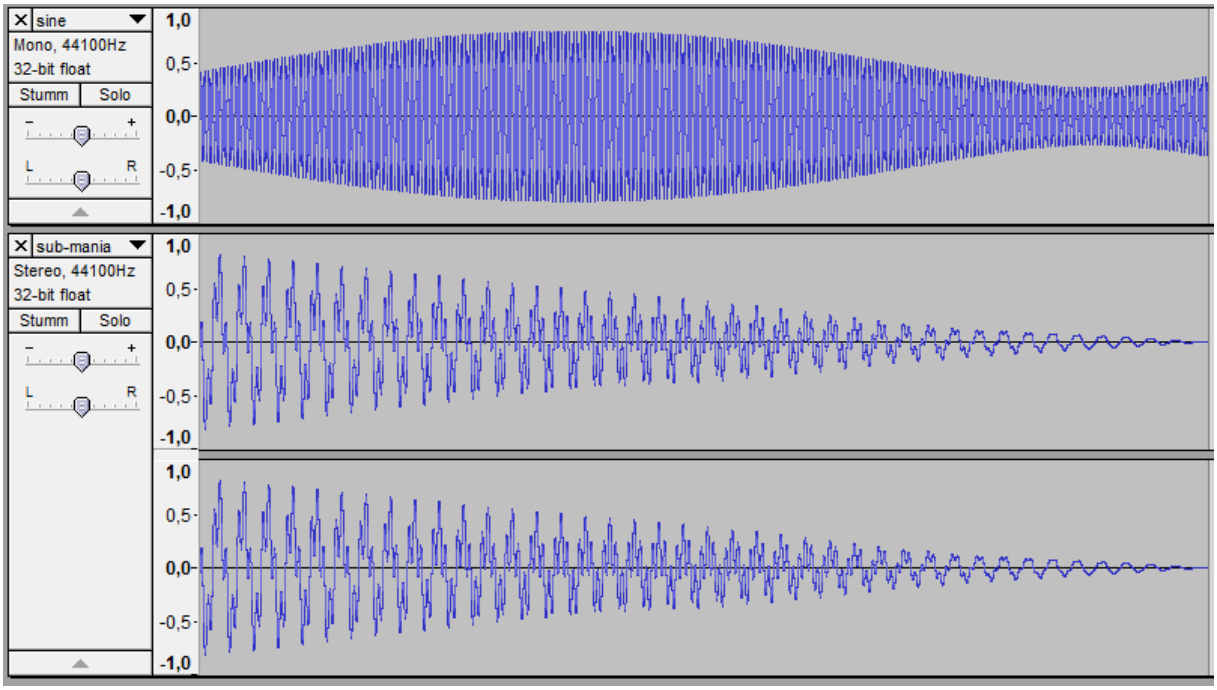

Abbildung 4.1: Screenshot zweier Waveforms

Statt der Amplitude kann man jedoch auch andere Werte auf der y-Achse auftragen, welche für eine Analyse der Daten nützlich sein könnten. Dazu zählen eine Waveform nach dB (eine logarithmische Skala, welche die Lautstärke besser repräsentiert) oder die Ansicht des Frequenzspektrums.

Die Waveform einer Sprachaufnahme in vier Variationen ist in Abbildung 4.2[1] veranschaulicht. Es ist eine männliche Stimme zu hören, welche die Zahlen *eins* bis *drei* in Englisch aufzählt. Die erste und die zweite Spur repräsentieren jeweils eine Waveform nach Amplitude bzw. dB. Die dritte und vierte Spur zeigen das Frequenzspektrum nach Hz bzw. logarithmisch, also $log(f)$. Das Spektrum zeigt die Verteilung der Energie. Blau bedeutet wenig, rot und weiß viel Energie.

Der nächste Abschnitt widmet sich der Speicherung und Komprimierung von Audiodaten auf dem Computer. Es werden Datenformate und Möglichkeiten zur Datenreduktion beschrieben.

[1] Die Screenshots stammen aus dem Programm **Audacity** (`http://audacity.sourceforge.net`).

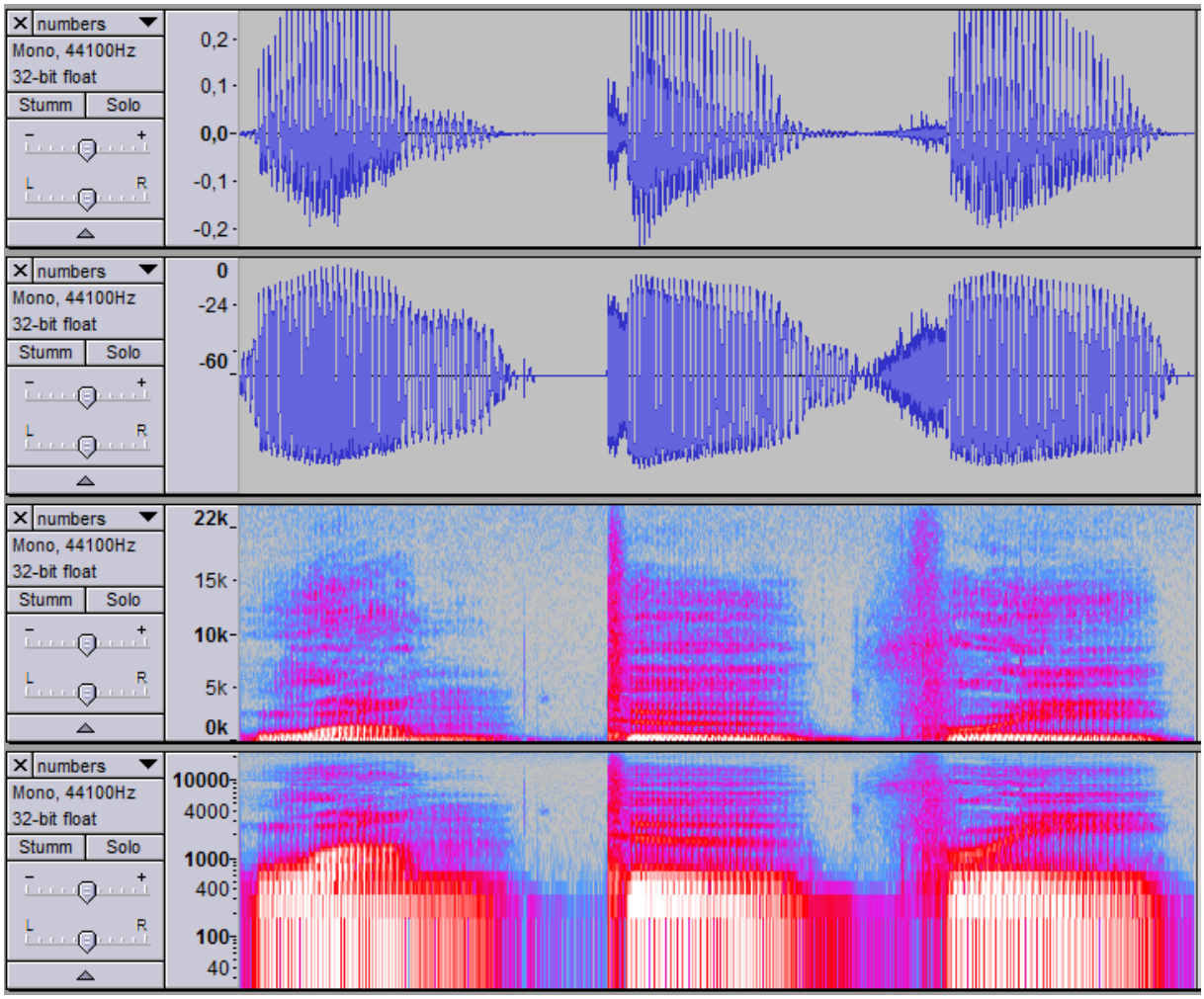

Abbildung 4.2: Eine Sprachaufnahme, dargestellt in vier Waveform Variationen

4.2 Speicherung und Komprimierung von Audiodaten

Wenn digitale Informationen nicht flüchtig sein sollen, dann müssen sie gespeichert werden. Gerade aber im Audiobereich können unter Umständen große Datenmengen entstehen. Dem wurde entgegengewirkt durch gestiegenes Fassungsvermögen von Speichermedien, doch die Übertragung kann einen Flaschenhals darstellen, besonders wenn öffentliche Netze, wie das Internet, benutzt werden. In solchen Bereichen ist es sinnvoll nur kleine Datenmengen verschicken zu müssen. Das spart Bandbreite, entlastet das Netzwerk und erhöht letztlich die Geschwindigkeit.

4.2.1 Datenformate

Der einfachste Ansatz Audiodaten zu speichern, ist die Rohdaten als Datei abzuspeichern. Dies stellt einen geringen Aufwand für den Applikationsentwickler dar, ist aber eine ungeeignete Lösung für eine dauerhafte Archivierung. Will man die Datei abspielen, müsste die Sampling Rate, die Auflösung und die Anzahl der Kanäle vom Audioplayer ”erraten” werden, da diese Eigenschaften nicht explizit bekannt sind. Deswegen existieren Dateiformate, welche zu den Audiodaten zusätzliche beschreibende Elemente beinhalten. Petermichl [Pet08, S. 691] unterscheidet zwischen folgenden fünf Formatfamilien:

1. **Rohdateien:** Keine implizite Beschreibung, enthält lediglich Audiodaten.
2. **IFF Familie:** *Interchange File Formats* wurden für den Austausch von Mediendateien entwickelt und sind nicht auf Audiodaten beschränkt. Sie besitzen einen beschreibenden Header am Anfang

der Datei und danach die Mediendaten. Beispiele: WAVE, AVI (*Audio Video Interleaved*) oder AIFF (*Audio Interchange File Format*).

3. **Streaming Format Familie:** Diese können für den Broadcast verwendet werden. Die Daten sind in Paketen organisiert, wobei ein Client sich mitten in einen Stream einklinken kann und sofort weiß wie ein Paket gelesen / dekodiert werden muss. Dies ist bei IFF nicht möglich. Beispiele: MPEG-1, MPEG-2, MPEG-1 Layer 3 (besser bekannt als MP3) und AC-3 (ein Tonsystem der Firma Dolby[2]).

4. **Container Format Familie:** Container speichern neben Mediendaten auch beschreibende Elemente (Meta-Daten), Timecodes, Markierungen (bspw. Kapitel in einem Film) und ganze Texte. So kann zu Musikdateien der Songtext hinzugefügt werden oder zu Videodaten Untertitel. Beispiele: MPEG-4, OGG-Vorbis und Matroska.

5. **Applikationsspezifische Formate:** Spezielle Applikationen können auch eigene Formate verarbeiten, die keinem Standard folgen. Dies ist nützlich, wenn die Datei Daten enthalten soll, die über die verfügbaren Formate nicht abgedeckt sind. Ein Beispiel dafür ist das für das Speichern von Archivdateien verwendete **AviBit** interne Format, welches auch für das Legal Recording verwendet wurde (siehe auch 5.3.1).

Im folgenden wird das WAVE Format beschrieben, da es für das Projekt Relevanz hat und beispielsweise vom Hilfsprogramm *Audio-Hardware Simulator* (siehe 5.2.3.3) verwendet wurden.

Das WAVE Format

Das WAVE Format basiert auf der *Resource Interchange File Format* (RIFF) Spezifikation von **Microsoft**[3] welche wiederum auf der IFF Spezifikation basiert [HR00]. RIFF kann, ebenso wie IFF, Multimediadaten speichern. Der Unterschied zwischen den beiden Spezifikationen ist die Byte Reihenfolge. RIFF benutzt Little Endian Werte[4].

Eine WAVE Datei ist in Chunks, in Datenblöcken, organisiert, welche wiederum Sub-Chunks enthalten können. Somit ist das RIFF Chunk der Hauptdatenblock und enthält selbst als Sub-Chunks zumindest den Format und den Data Datenblock. Der RIFF Chunk beginnt mit der eindeutigen Kennzeichnung ”RIFF”, gefolgt von der Größe der Datei in Byte und der WAVE Kennzeichnung ”WAVE”. Die detaillierte Struktur des WAVE Dateiformats ist in Abbildung 4.3 ersichtlich.

Der Format Chunk enthält neben seiner Identifizierung und Größenangabe Informationen über das Audiodatenformat, die Anzahl der Kanäle, Anzahl der Samples, die Datenrate in Byte pro Sekunde und die Auflösung in Bit. Interessant ist dabei vor allem das Audiodatenformat Feld, welches einen Code enthält, der das Format angibt. Diese WAVE Codec Registries sind im Standard RFC2361 spezifiziert [Fle98, S. 5ff]. Beschrieben ist dort beispielsweise das Format PCM, welches Audio Rohdaten repräsentiert. Ebenso zu finden ist A-law, welches eine wichtige Rolle im Projekt einnimmt (siehe auch 5.2.1). Format Identifier 0x0055 kennzeichnet MPEG-1 Layer 3 Daten, das bedeutet also, dass WAVE Dateien auch MP3-codierte Daten enthalten können.

Der Data Chunk beherbergt schließlich neben seiner Identifizierung und Größenangabe die eigentlichen Audiodaten. Es existieren noch weitere optionale Sub-Chunks, die aber in der Praxis selten vorkommen. Beispielsweise definiert der Cue Chunk Positionen innerhalb der Audiodaten, ähnlich wie Kapitel auf einer DVD, mit Hilfe des Playlist Chunks kann dann angeben werden, in welcher Reihenfolge diese Positionen abgespielt werden sollen und mittels dem Chunk Label können für die Positionen auch noch Beschriftungen vergeben werden (vgl. [HR00] und [Pet08, S. 696f]). Das Feld *Padding* wird nur hinter den Audiodaten angehängt, wenn die Datenlänge ungerade ist.

[2] `www.dolby.com`
[3] `www.microsoft.com`
[4] Bei Little Endian steht das niederwertigste Byte an der vordersten Stelle.

	Feld	Bezeichnung	Start Byte	Länge (Byte)
	Identifier "RIFF"	ckID	0	4
	Größe (in Byte)	cksize	4	4
	WAVE Identifier "WAVE"	WAVEID	8	4
Format Chunk	Identifier "fmt"	ckID	12	4
	Größe	cksize	16	4
	Audiodatenformat	wFormatTag	20	2
	Anzahl Kanäle	nChannels	22	2
	Anzahl Samples	nSamplesPerSec	24	4
	Datenrate	nAvgBytesPerSec	28	4
	Blockgröße	nBlockAlign	32	2
	Auflösung	wBitsPerSample	34	2
Data Chunk	Identifier "data"	ckID	36	4
	Größe	cksize	40	4
	Audiodaten		44	n
	Padding		45+n	0/1

Abbildung 4.3: Aufbau und Struktur des WAVE Dateiformats (vgl. [Pet08] und [Kab06]).

Da die Dateigröße, besonders bei der Übertragung von Audiodateien, immer ein wesentlicher Faktor ist, existiert die Komprimierung. Der folgende Unterabschnitt behandelt verschiedene Aspekte.

4.2.2 Komprimierung

Komprimierung beschreibt eine Datenreduktion durch Algorithmen, wobei das originale Signal nicht in seinen Klangeigenschaften verändert wird. Je nach Verfahren (im englischen meist als *Codec* bezeichnet) wird dazu ein entsprechender *Coder* verwendet, der das Signal kodiert und somit komprimiert. Zum Dekomprimieren wird dementsprechend ein *Encoder* verwendet [SKM+06, S. 5].

Im Allgemeinen wird zwischen zwei Formen der Komprimierung unterschieden (vgl. [Ler08] und [WW01, Kapitel 5]): der verlustlosen und der verlustbehafteten. Bei der **verlustlose Komprimierung** bliebt das Ursprungssignal bitgenau erhalten, der Prozess ist beliebig oft ohne Qualitätsverlust wiederholbar. Dies ist vergleichbar mit der Datenreduktion von Binärdaten: Man will auf jeden Fall die kompletten Daten nach der Kompression wiederherstellen können, weil sonst die Binärdaten wertlos wären. Verlustlose Verfahren bieten ein maximale Datenreduktion von 2:1. Bekannte Codecs sind Shorten (mittlerweile veraltet), der Free Lossless Audio Codec FLAC und Monkey's Audio (APE). Entgegen werden bei der **verlustbehaftete Komprimierung** unrelevante Informationen im Ursprungssignal entfernt oder nur näherungsweise abgespeichert. Dadurch kann eine Datenreduktion von 10:1 und mehr erreicht werden. Bekannte Codecs sind MPEG-1 Layer 3 (MP3), OGG-Vorbis, AAC und AC-3.

Die erwähnten Codecs versuchen jeweils eine möglichst hochwertige Qualität zu erzielen und sind vor allem für Musikaufnahmen geeignet. Bei Sprachaufnahmen gelten jedoch andere Kriterien. Betrachtet man die VoIP-Telefonie ergeben sich folgende Anforderungen:

- **Geringere Datenrate:** Je geringer die Datenrate, desto schneller lässt sich die Aufnahme übertragen, die Latenzzeit bleibt gering. MP3-codierte Musikdateien benötigen 128 kbps und mehr, um CD Qualität zu erreichen. Für Sprachaufnahmen reichen 64 kbps damit sie noch immer klar verständlich sind.
- **Simple Dekodierung:** Telefone verfügen über eine begrenzte Rechenleistung. Das Dekodieren sollte also so unaufwendig wie möglich sein und möglichst keine Fließkommaoperationen benötigen.
- **Paketorientiert:** Der Datenstrom soll in einzelne Pakete unterteilt werden können, so dass einzelne Bereiche unabhängig von anderen dekodiert werden können. Die Latenzzeit kann minimiert werden je kleiner die Pakete sind, die man verschickt.

4.2.2.1 Sprachkompression

Die bekanntesten Codecs mit offenem Standard stammen von der *Internationalen Fernmeldeunion* bzw. (engl.) *International Telecommunication Union*[5]. Deren Standardisierungsabteilung ITU-T entwickelt Normen, Standards und Empfehlungen, welche in 23 Klassen (die Buchstaben A - Z ohne die Klassen B, C und W) eingeteilt sind) [ITUa]. Die Klasse H beispielsweise widmet sich audiovisuellen und multimedialen Systemen. Sie enthält unter anderem Videocodecs wie die H.26x Serie. Für die Sprachkompression ist die Klasse G relevant. Diese beschreibt vor allem Empfehlungen zur Datenübertragung und digitalen Netzwerken. Darunter fällt eben auch die Telefonie und somit die Sprachkompression.

Codecs zur Sprachkompression sind unter G.7xx zu finden. Die wichtigsten sind nachfolgend kurz zusammengefasst (vgl. [ITUb] und [SKM+06, S. 27f, 71ff]).

Der Codec **G.711** arbeitet mit einer Abtastrate von 8.000 Hz und einer Auflösung von 8 Bit. Die erzielte Datenrate liegt somit bei 64 kbps, der angesprochene Frequenzbereich liegt zwischen 300 und 3.400 Hz. Bei der Quantisierung wird in Europa das A-Law Verfahren angewandt und in Nordamerika und Japan µ-Law (siehe unten). Diese beiden Verfahren sind nicht miteinander kompatibel. G.711 bietet ISDN Sprachqualität und schneidet bei subjektiven Tests sehr gut ab. Der Codec ist nach wie vor im Einsatz und wird wegen seiner Einfachheit von vielen Hardware- und Softwaresystemen unterstützt.

G.726 wird als *Adaptive Differential Pulse Code Modulation* bezeichnet. Im Gegensatz zu G.711 quantisiert diese Technik die Sprachsignale nicht direkt, sondern den Unterschied (*differential*) zwischen einem vorhergesagten und dem nächsten realen Signal. Die Vorhersage wird in jedem Rechenschritt neu angepasst (*adaptive*). Dadurch lässt sich eine niedrigere Datenrate bei vergleichbarer Qualität erzielen. Diese ist bei G.726 wählbar und kann zwischen 16 und 40 kbps liegen.

Der Audiocodec **G.722** arbeitet bereits mit einer Abtastrate von 16.000 Hz und einer Auflösung von 14 Bit. Der abgedeckte Frequenzbereich liegt bei 50 bis 7.000 Hz. G.722 nutzt ebenfalls *Adaptive Differential Pulse Code Modulation* und splittet das Eingangssignal zusätzlich in zwei Bereiche (*subbands*) auf. Dabei wird der Bandbereich mit einer Datenrate von 16 kbps codiert und der untere mit 48 kbps, das ergibt insgesamt eine (fixe) Datenrate von 64 kbps. Es lassen sich drei Operationsmodi auswählen, welche letztendlich die Audioqualität bestimmen. Dadurch, dass die Datenrate aber die selbe bleibt, kann der eventuell verbleibende Platz für zusätzliche Informationen genutzt werden.

Der große Vorteil von **G.728** ist seine niedrige Latenzzeit (lediglich 0.625 ms bei 8.000 Hz). Zur Codierung wird das *Code Excited Linear Prediction* Verfahren verwendet. Dabei wird eine Tabelle mit angenäherten Waveforms verwendet, wobei lediglich der Tabellenindex übertragen werden muss. G.728 hat eine Datenrate von 16 kbps.

CELT ist ein freier Audio Codec der Xiph Foundation[6]. Sein Zielt ist eine möglichst niedrige Latenz (niedriger als G.728/G.729 bei vergleichbarer Abtastrate, laut Website zwischen 3 und 9 Millisekunden). Er ist nicht nur für Sprache, sondern auch für Musik einsetzbar, erlaubt also auch höhere Abtastraten (bis zu 96.000 Hz). Die Datenrate liegt zwischen 32 und 128 kbps, außerdem kann der Codec auf Paketverluste bei der Übertragung in Netzwerken reagieren. Die Kodierung ist verlustbehaftet.

Das A-Law und das µ-Law Verfahren arbeiten logarithmisch. Kleinere, das heißt, leisere Signalwerte, die in der Regel häufiger vorkommen, werden deshalb feiner skaliert und das Rauschen besser unterdrückt [SKM+06, S. 71f]. Die Werte verlaufen entlang der A- bzw. µ-Kennlinie (siehe Abbildung 4.4). Für die Datenreduktion bedient man sich folgendem Trick: Bei der Quantisierung werden die Werte mit einer höheren Auflösung (13 Bit, mit einem Bit als Vorzeichen) abgetastet und danach auf 8 Bit umgerechnet (vgl. [Voi], [ITWa] und [ITWb]).

Die folgenden Formeln zeigen die Berechnung der A- und µ-Kennlinie. Formel 4.2 ist die stetige Funktion für A-Law im Bereich $[0, 1]$, 4.3 die inverse Funktion dazu. Ebenso ist Formel 4.4 die stetige Funktion für µ-Law im Bereich $[-1, 1]$ und 4.5 die Umkehrung dazu. Abbildung 4.4 zeigt die resultie-

[5] http://www.itu.int
[6] http://www.xiph.org

renden Werte der A-Law Funktion in einer Kurve, der A-Kennlinie.

$$C(x) = \begin{cases} \frac{Ax}{1+\ln A}, & \text{wenn } 0 \le x \le \frac{1}{A} \\ \frac{1}{1+\ln A} + \frac{\ln Ax}{1+\ln A}, & \text{wenn } \frac{1}{A} < x \le 1 \end{cases} \tag{4.2}$$

$$C^{-1}(y) = \operatorname{sgn} y \begin{cases} \frac{|y|(1+\ln A)}{A}, & \text{wenn } |y| < \frac{1}{1+\ln A} \\ \frac{\exp(|y|\left(1+\ln A\right)-1))}{A}, & \text{wenn } \frac{1}{1+\ln A} \le |y| \le 1 \end{cases} \tag{4.3}$$

... mit $A = 87,56$

$$C(x) = \operatorname{sgn} x \frac{\ln\left(1+\mu|x|\right)}{\ln\left(1+\mu\right)}, \qquad \text{für } -1 \le x \le +1 \tag{4.4}$$

$$C^{-1}(y) = \operatorname{sgn} y \frac{1}{\mu}\left((1+|\mu|)^{|y|} - 1\right), \qquad \text{für } -1 \le y \le +1 \tag{4.5}$$

... mit $\mu = 255$

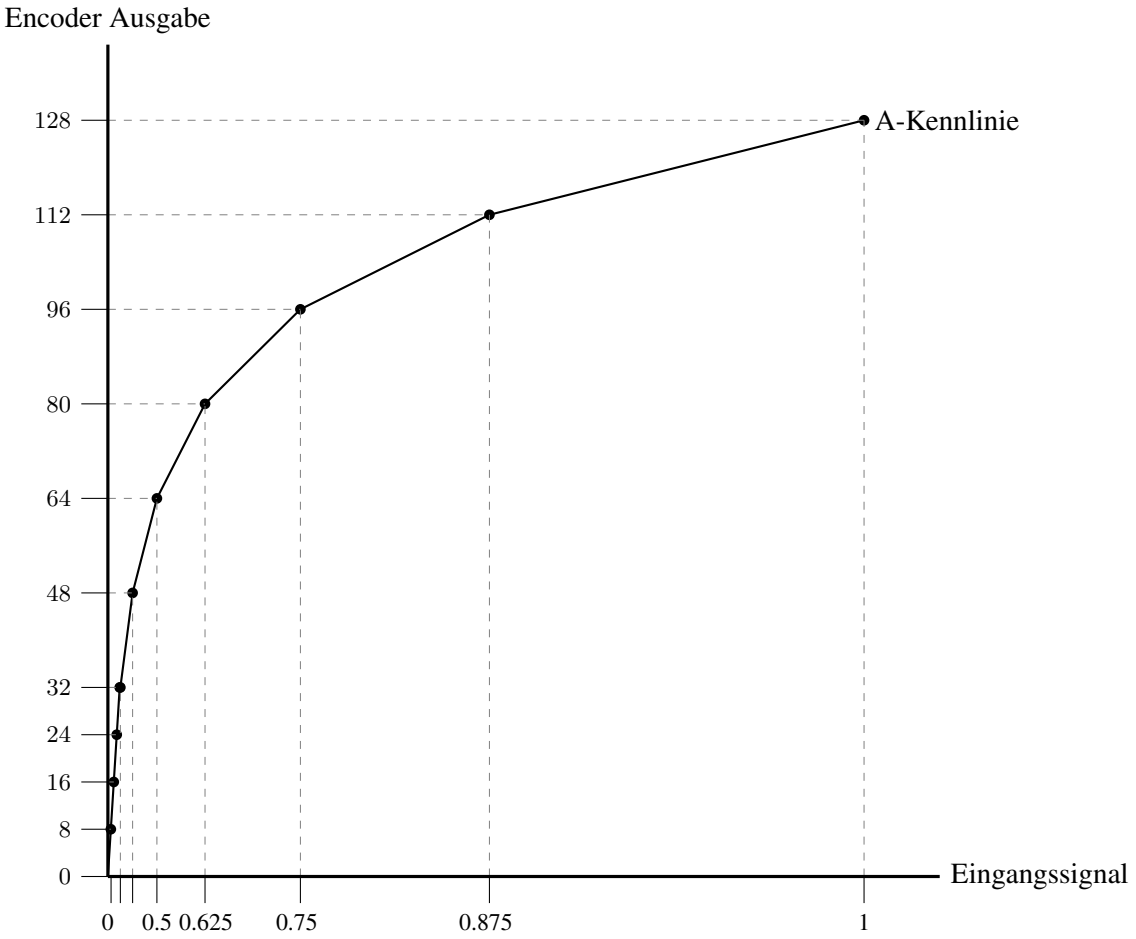

Abbildung 4.4: Die A-Kennlinie der A-Law Kodierung.

Im digitalen Bereich wird ein PCM Audio Sample als 16 Bit Wert repräsentiert. A-Law bzw. µ-Law vollführen also eine Konvertierung von 16 auf 8 Bit wobei der resultierende Wert eine Minimalversion einer Gleitkommazahl darstellt. Sie besitzt ein Bit Vorzeichen, vier Bit Exponent und drei Bit Mantisse[7].

Das Code Listing 4.1 zeigt die Konvertierung eines linearen PCM Samples als `short integer` (16 Bit) in ein komprimiertes `unsigned char` (8 Bit) A-Law Sample [Haz]. Man kann sehen, wie die drei Komponenten - Vorzeichen, Exponent, Mantisse - berechnet werden. Für den Exponent wird ein kleiner Look-up Table zur Hilfe genommen.

```
1  static char alaw_compress_table[128] =
2  {
3       1,1,2,2,3,3,3,3,
4       4,4,4,4,4,4,4,4,
5       5,5,5,5,5,5,5,5,
6       5,5,5,5,5,5,5,5,
7       6,6,6,6,6,6,6,6,
8       6,6,6,6,6,6,6,6,
9       6,6,6,6,6,6,6,6,
10      6,6,6,6,6,6,6,6,
11      7,7,7,7,7,7,7,7,
12      7,7,7,7,7,7,7,7,
13      7,7,7,7,7,7,7,7,
14      7,7,7,7,7,7,7,7,
15      7,7,7,7,7,7,7,7,
16      7,7,7,7,7,7,7,7,
17      7,7,7,7,7,7,7,7,
18      7,7,7,7,7,7,7,7
19 };
20
21 unsigned char linearToALaw(short sample)
22 {
23      int sign;
24      int exponent;
25      int mantissa;
26      unsigned char compressed;
27
28      sign = ((~sample) >> 8) & 0x80;
29      if (!sign)
30           sample = (short)-sample;
31      if (sample > cClip)
32           sample = cClip;
33      if (sample >= 256)
34      {
35           exponent = (int)ALawCompressTable[(sample >> 8) & 0x7F];
36           mantissa = (sample >> (exponent + 3) ) & 0x0F;
37           compressed = ((exponent << 4) | mantissa);
38      }
39      else
40      {
41           compressed = (unsigned char)(sample >> 4);
42      }
43      compressed ^= (sign ^ 0x55);
44
45      return compressed;
46 }
```

Listing 4.1: *linear_to_alaw.cpp:* Konvertierung eines PCM in ein A-Law Sample (nach [Haz]).

[7] Siehe dazu auch Standard IEEE 754 für Gleitkommazahlen http://754r.ucbtest.org/standards/754.pdf.

Der nächste Abschnitt widmet sich der Übertragung von Audiodaten in digitalen Netzen. Die dort geschilderten Möglichkeiten wurden in der Planungsphase des Projekts in Betracht gezogen. Abschließend wird in Unterabschnitt 4.3.4 ein Resümee über diese Möglichkeiten gebildet und die Vor- und Nachteile diskutiert, welche zu einer eigenen Lösung der Übertragungstechnik im Projekt führte.

4.3 Übertragung von Audiodaten über digitale Netze

Über digitale Netze können Daten ohne Verlust übertragen werden. Man bedient sich standardisierter Übertragungsprotokolle, wie das zuverlässige TCP Protokoll[8], Beispielsweise können Audiodaten von einem PC zu einem anderen kopiert werden, mit dem Ziel eine exakte Kopie zu erstellen. Je nach Geschwindigkeit des Netzes sind die Daten nach einer bestimmten Zeit übertragen und können verwendet werden. Dies ist aber unpraktisch, wenn man Daten gleich benutzen will. Dazu greift man auf das schnellere - und weniger zuverlässigere - UDP Protokoll[9] zurück. So ist beispielsweise mit entsprechenden Codecs der Streaming Format Familie (siehe 4.2) möglich, Audiodaten zu hören, noch bevor die Datei als ganzes übertragen ist. Das ist insofern bei Live Streams (z.B. von Sport- oder Konzertübertragungen) nützlich, wo das Ende noch gar nicht bekannt ist. Streaming via UDP Protokoll wird in Unterabschnitt 4.3.1 beschrieben. Eine weiterführende Audio Streaming Technik ist JACK, ein Sound Server Daemon zur Echtzeitübertragung. Dieser wird in Unterabschnitt 4.3.2 vorgestellt. Schließlich soll auch noch auf den wichtigsten Bereich (im Rahmen dieser Arbeit) bezüglich Übertragung von Audiodaten eingegangen werden: die Internettelefonie. Das Thema Voice over IP inklusive Protokolle wird ausführlich in Unterabschnitt 4.3.3 behandelt.

4.3.1 Streaming via UDP Protokoll

UDP ist eine Transportschicht im OSI Schichtenmodell[10]. UDP hat die Eigenschaften, dass es verbindungslos ist und keine Garantie über Reihenfolge und Empfang der geschickten Pakete gibt. Jedes versendete Paket ist unabhängig. Diese eher ungünstigen Eigenschaften machen das Versenden von Daten jedoch schneller als zuverlässige TCP und damit attraktiv für datenintensive Anwendungen. Weiters ist es möglich mittels UDP Multicast Nachrichten zu schicken. Dabei werden Pakete beim Router dupliziert. Somit ist es einem Server möglich viele Clients mit nur einem Datenstrom zu versorgen.

Shuvalov [Shu99] beschreibt einen Video Server, der digitale Inhalte an Clients verteilen kann. Benutzer können die Inhalte selbst auswählen. Die Architektur besteht aus einem Datenbank Server (hält die Registrierung der Medien, verwaltet die Konfiguration), einem oder mehreren Acquisition Server(n) (zeichnet Videoinhalte auf) und einem oder mehreren Push Server(n) (speichert Daten, versorgt die Clients mit Daten). Das Video Streaming der Push Server zum Client erfolgt via UDP.

4.3.2 JACK Audio Connection Kit

JACK ermöglicht den Austausch von Audiodaten zwischen diversen Programmen mit dem Zielen eine möglichst niedrige Latenzzeit zu erreichen, die Daten also möglichst in Echtzeit zu übertragen, und eine einheitliche Schnittstelle zu bieten [JAC]. Es steht für die Betriebssysteme Linux, OS X, Solaris und Windows zur Verfügung. JACK umfasst die folgenden Services:

- **Die JACK API:** Das *Application Programming Interface* muss von Programmen, welche JACK unterstützen sollen, benutzt werden. Es bietet Funktionen (beispielsweise eine Callback Funktion die aufgerufen wird, um hereinkommende Audiodaten zu verarbeiten), Strukturen und Protokolle, welche mit JACK in Verbindung stehen.

[8] TCP: Transmission Control Protocol

[9] UDP: User Datagram Protocol

[10] OSI: Open Systems Interconnection Reference Model, siehe auch ITU Empfehlung X.200: `http://www.itu.int/rec/T-REC-X.200-199407-I/en`

- **JACK Clients:** Die Programme selbst sind Clients des JACK Servers und können durch die Programmierschnittstelle mit diesem kommunizieren.
- **Der JACK Server:** Der JACK Server (ein Daemon Prozess) ist die zentrale Applikation die sich um die Synchronität der Clients und den Transport der Audiodaten kümmert.
- **GUI Control Applications:** Es existieren externe Programme, welche die Verwaltung und Steuerung von JACK auf einer graphischen Oberfläche ermöglichen. Ein Implementation dazu wäre das Programm *QjackCtl*[11].

Wozu wird JACK also benutzt? Tontechniker und Studios greifen schon längst auf computergestützte Audioverarbeitung zurück und benützen ganze Ketten von Aufnahme-, Effekt- und Verarbeitungsprogrammen. Diese werden als *Digital Audio Workstation* (DAW) bezeichnet. JACK kann die Programme einer DAW verbinden und ermöglicht einen synchronen Austausch von Audiodaten.

Mittels NetJack, welches in JACK integriert ist, kann man Daten auch über Netzwerke transportieren lassen. NetJack kümmert sich dabei um die Synchronisierung, die bei verschiedenen Computersystem kritisch ist. Das Problem bei der Synchronisierung von zwei PCs in einem Netzwerk sind laut Akester et al. [AH03] deren Sample Clocks der Audio Hardware. Die Abweichungen dabei sind oft nur gering. Wird eine Musikdatei mit einer Sampling Rate von 44100 Hz abgespielt, so kann es sein, dass der Empfänger tatsächlich durch eine Ungenauigkeit in der Hardware eine Sampling Rate von 44.101 Hz benutzt. Dies führt über eine längere Zeitspanne bei dem Empfänger unweigerlich zu einem Buffer Underrun da der Empfänger die Datei letztendlich zu schnell abspielt. Auch ein Buffer Overrun ist möglich, wenn der Empfänger zu langsam spielt und deswegen über die Zeit zu viele Datenpakete erhält. NetJack behandelt dieses Problem, indem es alle Clients auf eine Soundkarte synchronisiert [Net]. Carôt et al. [CHW09] zeigen mit ihren NetJack Implementierungen, dass es möglich ist, sogar Musiker miteinander zu verbinden, die sich in zwei verschiedenen Städten (Lübeck, Berlin) befinden. Mittels Sequencer wurden auf den PCs Musik komponiert. Beide Parteien arbeitet am gleichen Stück. Es wurde beobachtet, dass das Playback auf den Workstations synchron war und es zu keinen signifikanten Ausfällen kam.

4.3.3 Voice over IP

Voice over IP bedeutet Internet-Telefonie, wobei man dies natürlich auch in einem LAN nutzen kann. Auf jeden Fall ist aber Voice over IP etwas anderes als die klassische Telefonie über das Telefonnetz, auch wenn es letztendlich den gleichen Zweck erfüllt. Für den Anfang seien hier die Vor- und Nachteile der Internet-Telefonie im Gegensatz zu anderen Telefonsystemen aufgelistet.

Die wichtigsten Vorteile der Voice over IP Telefonie sind:

- Der wichtigste Vorteil gegenüber der herkömmlichen Telefonie sind die **Kosten**. An Orten, an denen ohnehin eine Breitbandleitung mit Flatrate Tarif vorhanden ist, ist ein Anruf von einem VoIP Teilnehmer zu einem anderen meist kostenlos. Dazu ist zu bemerken, dass es populäre kostenlose Services gibt (wie z.B. Skype[12]), aber natürlich auch kostenpflichtige. Es können zusätzliche Gebühren anfallen, wenn beispielsweise ein VoIP Teilnehmer eine Festnetznummer anrufen will, da man dafür einen Gateway ins normale Telefonnetz benötigt.
- Durch die höhere zur Verfügung stehende **Bandbreite** bei Breitbandleitungen muss das Signal nicht so stark komprimiert bzw. bandbeschränkt werden. Die Sprachqualität bei VoIP kann besser sein, als bei anderen Telefonnetzen, hängt aber in letzter Linie vom verwendeten Codec ab. Diese können zur Skalierung des Datentransfers vom Benutzer oder Systemadministrator auf ausgewählte Codecs beschränkt werden.

[11] http://qjackctl.sourceforge.net
[12] www.skype.com

- Die **Portabilität** ist auch ein wichtiger Faktor. Ein Anschluss ist unabhängig vom Aufenthaltsort immer unter derselben Nummer zu erreichen. Dies ist auch bei Mobiltelefonen gegeben, sofern sie im jeweiligen Gebiet eine Netzabdeckung haben.
- Zusätzlich zum Telefonieren erlaubt VoIP Zusatzdienste wie **Telekonferenzen** mit mehreren Teilnehmern, Videotelefonie und Fax (Fax over IP). Außerdem lassen sich VoIP Systeme vielfach konfigurieren und anpassen.

Die Nachteile der Voice over IP Telefonie sind:

- Telefonnetze sind in der Regel nicht von **Stromausfällen** betroffen. Das Telefon im Haus funktioniert noch, wenn der Strom ausfällt. Gleiches gilt natürlich auch für das Mobiltelefon. Voice over IP Telefone werden jedoch durch das Stromnetz versorgt und sind deswegen direkt von einem Ausfall betroffen. Das selbe gilt für einen Ausfall des Internets. Um eine Ausfallsicherheit zu gewähren braucht man eine redundante Stromversorgung auf die im Notfall umgeschalten werden kann, sowie eine zweite Internetverbindung.
- Genau so wie VoIP eine bessere Sprachqualität bieten kann, kann sie auch schlechter sein. Wenn die Leitung zu langsam ist entsteht eine zu lange Verzögerung was die **Echtzeitfähigkeit** massiv einschränkt. Dieses Problem hat man bei herkömmlichen Telefonnetzen nicht.

Die Voice over IP Technik besitzt eine Menge an Eigenschaften mit denen sich Entwickler auseinandersetzen müssen. Diese werden auch als Quality of Service (Qos) bezeichnet. Davidson et al. [DPB+06, Kap. 7] beschreiben diese:

- **Latenzzeit:** Die Verzögerung oder Latenzzeit beläuft sich auf die Zeit, welche eine Information (bspw. ein gesprochenes Wort) vom Sprecher zum Zuhörer braucht. Dabei gibt es die Ausbreitungsverzögerung (Schall in der Luft, aber vor allem auch die digitalen Informationen über Kabel), die Abwicklungsverzögerung in der Bearbeitung der digitalen Signale und die Warteschlangenverzögerung bevor IP-Pakete genug Daten besitzen, bis sie versendet werden können.
- **Jitter:** im VoIP Bereich wird der Jitter durch die Reihenfolge verursacht, in der Datenpakete beim Empfänger ankommen. Da nicht alle Pakete die gleiche Route nehmen müssen, ist auch keine Empfangsreihenfolge garantiert. Um dem entgegenzuwirken wird ein Buffer eingesetzt, welche die Pakete zuerst sortiert. Dies geschieht aber auf Kosten der Verzögerung und summiert sich zur gesamten Verzögerung dazu.
- **Analog-Digital Wandlung:** Da die Internettelefonie natürlich ausschließlich digital funktioniert, muss das analoge Signal des Sprechers digitalisiert werden. Dazu ist eine Analog-Digital Wandlung notwendig (siehe auch Abschnitt 3.2). Digitale Signale haben beim Transport jedoch den Vorteil, dass sie nicht verändert werden. Beim Empfänger wird dann wiederum eine Digital-Analog Wandlung durchgeführt um einen Ton produzieren zu können.
- **Kompression:** Die durch die Analog-Digital Wandlung resultierenden PCM Daten werden vor dem Verschicken komprimiert um Bandbreite zu sparen. Am häufigsten werden dabei das A-Law bzw. das μ-Law Verfahren eingesetzt, da sie die größte Kompatibilität besitzen und schnell zu realisieren sind. Es existieren aber noch viele andere Sprachcodecs (siehe dazu 4.2.2.1).
- **Sprachaktivitätserkennung:** Wenn am Telefon Stille herrscht, so werden unnötiger Weise Daten übertragen, die keine für das Gespräche relevante Informationen enthalten. Stille ist sogar zu mindestens 50% bei einem Dialog der Fall, da ja immer einer zuhört. Eine Gesprächsaktivitätserkennung kann mit Hilfe eines dB Grenzwertes unterscheiden, ob jemand spricht oder nicht. Damit kann man Bandbreite sparen.

- **Paketverlust:** Mit Paketverlust ist immer zu rechnen. Dem kann man aber auf Protokollebene entgegenwirken (siehe 4.3.3.2). Wichtig ist, das Paketverlust erkannt und gehandhabt wird. Kleine Aussetzer kann man überbrücken, beispielsweise in dem man vorherige Pakete nochmals abspielt oder sogar den weiteren Verlauf interpoliert. Bei Lücken von 20ms bekommt der Mensch das nicht mit.

- **Transportprotokolle:** Für Voice over IP existieren Standards, damit eine Konnektivität zwischen unterschiedlichen Systemen gewährleistet ist. Diese Standards betreffen u.a. auch die Transportprotokolle von VoIP. Für Audiodaten wird das RTP[13] verwendet, welches via UDP (wegen des Geschwindigkeitvorteils) versendet wird. Das heißt, ein Audiopaket steckt in einem RTP Packet und dieses steckt in einem UDP Paket. Im Gegensatz dazu wird das SIP[14], welches für den Gesprächsaufbau, die Steuerung und die Gesprächsbeendung zuständig ist, via TCP verschickt. Unterabschnitt 4.3.3.2 geht näher auf die Voice over IP Protokolle ein.

4.3.3.1 Gesprächsablauf

Generell unterscheidet man zwischen zwei Typen bei Voice over IP Verbindungen: Peer-to-Peer und Host-basiert. Diese definieren die notwendige Architektur im VoIP System.

Bei der **Peer-to-peer Verbindung** besteht eine direkte Verbindung zwischen den Teilnehmern einer VoIP Session. Dies setzt Software voraus, die in der Lage ist eine Verbindung herzustellen, zu verwalten und wieder abzubauen. Ein Beispiel dafür ist die Voice over IP Software Skype. Diese ist zwar Closed Source, doch Baset et al. [BS06] untersuchten die Software und die zu Grunde liegenden Transportprotokolle. Eine Verbindung wird via TCP über eine Challenge-Response-Authentifizierung hergestellt. Audiodaten werden via UDP ausgetauscht, deren Länge zwischen 40 und 120 Bytes waren. Skype benötigte in den Experimenten eine Bandbreite von 5 kbps.

Bei der **Host-basierte Verbindung** wird eine Verbindung über einen Host (Server) abgewickelt, auch als *Private Branch Exchange* (PBX) bezeichnet. Ein Beispiel dafür ist der Open Source PBX Asterisk[15]. Diese unterstützt zahlreiche Protokolle zum Verbindungsaufbau (darunter natürlich auch die wichtigen Protokolle: SIP und H.323), damit die Interoperabilität zwischen verschiedenen VoIP Systemen gewährleistet ist. Bei Nicht-Unterstützung eines Protokolls oder durch eine fehlerhafte Implementation kann es zu keiner erfolgreichen Verbindungsherstellung kommen. Zum Versand der Audiodaten wird RTP verwendet. Asterisk unterstützt auch eine Vielzahl von Audiocodecs. Hardware kann über Erweiterungskarten (wie PCI) angeschlossen werden (vgl. [Ast10]).

Der Gesprächsablauf folgt unabhängig von den benutzten Protokollen und dem Infrastrukturtyp einigen notwendigen Schritten. Zuerst wird eine Verbindung hergestellt, entweder zu einem Host oder gleich zu einem Client. Dann kann die Gesprächsführung (*Session*) initialisiert und die notwendigen Parameter, wie beispielsweise verwendete Codecs, ausgehandelt werden. Die Sprachübertragung kann danach starten. Beim Sender wird nun Sprache von einem Inputgerät, bspw. von einem Mikrofon, erfasst. Es folgt die Analog-Digital Wandlung des Signals und eine Komprimierung mit dem ausgehandelten Codec. Pakete zur Übertragung der Audiodaten werden erstellt und verschickt. Der Empfänger wartet auf diese Pakete, liest sie ein und dekomprimiert die Audiodaten. Es folgt eine Digital-Analog Wandlung, damit die Signale über ein Outputgerät, bspw. Lautsprecher, ausgegeben werden können. Schlussendlich wird das Gespräch beendet, die Session abgebaut und die Verbindung getrennt.

4.3.3.2 Voice over IP Protokolle

Protokolle regeln und standardisieren den Datenaustausch zwischen Systemen. Im VoIP Bereich werden zwei Arten von Protokollen unterschieden: Signalisierungsprotokolle und Transportprotokolle. Signali-

[13] RTP: Real-Time Transport Protocol
[14] SIP: Session Initiation Protocol
[15] `http://www.asterisk.org`

sierungsprotokolle sorgen für den Gesprächsaufbau, die Steuerung und die Gesprächsbeendung während Transportprotokolle die Audiodaten transportieren und gleichzeitig den QoS kontrollieren können. Zu den Signalisierungsprotokollen gehören H.323 (eine Empfehlung von ITU-T), SIP (ein Standard der IETF[16]) und MGCP[17] (ebenso von der IETF). Als Transportprotokoll hat sich RTP herauskristallisiert. Es wird von H.323, SIP und MGCP verwendet. Zur Steuerung des Datenstroms wird RTCP[18] verwendet. H.323, SIP und RTP/RTCP werden in den nächsten Absätzen genauer beschrieben.

H.323 ist eine ITU-T Empfehlung mit dem Namen *Packet-based multimedia communications systems* [ITUc, H.323]. H.323 ist für sich jedoch nicht alleinstehend. Die Empfehlung referenziert auf andere Empfehlungen der H Serie. Diese werden benutzt um Anrufe zu signalisieren, zu administrieren und zu serialisieren (H.225), die Kanalbelegung und andere Parameter auszuhandeln (H.245), Servicefunktionen in der IP-Telefonie abzubilden (H.450.x) und für Sicherheit und Authentifizierung zu sorgen (H.235) (vgl. [RTR06, 35f], [DPB+06, Kap. 11] und [Pro, H.323 Protocols Suite]). Das Protokoll unterstützt also die Signalisierung und den Medientransport indirekt über andere Protokolle. Zu den Medien zählen Audio, Video und Daten. In der H.323 Systemarchitektur gibt es Gateways die eine Verbindung zu anderen Telefonnetzen herstellen können (z.B. Festnetz oder SIP-basiertes VoIP), Gatekeeper welche die Adressumsetzung und Zugriff auf das H.323 Netzwerk ermöglichen (also die Signalisierung herstellen), Multi-Point Control Units die Konferenzschaltungen abwickeln und natürlich die Terminals selbst mit denen telefoniert wird [Sch03, S. 6ff]. Diese Infrastruktur ist in Abbildung 4.5 verdeutlicht. Zum Transport der Mediendaten nutzt H.323 wie bereits erwähnt RTP.

Ein Gesprächsablauf ist in Abbildung 4.6 dargestellt. Dabei kommunizieren zwei Endpunkte mittels eines Gatekeepers miteinander. Es sind die verschiedenen Protokolle ersichtlich, mit deren Hilfe Sitzungs- und Gesprächsdaten ausgetauscht werden.

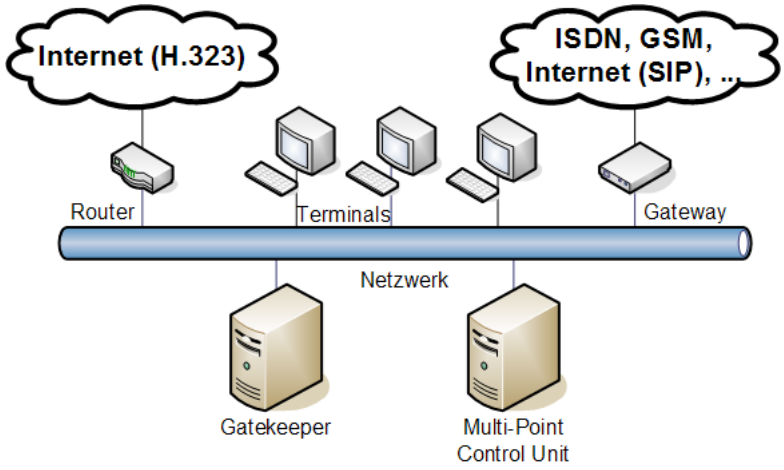

Abbildung 4.5: Die H.323 Infrastruktur mit Gatekeeper, Gateway, Multi-Point Control Unit und Terminals als Endpoints (vgl. [Pro, H.323 Architecture] und [Sch03, S. 6]).

[16] IETF: Internet Engineering Task Force, `http://www.ietf.org`
[17] MGCP: Media Gateway Control Protocol
[18] RTCP: Real-Time Control Protocol

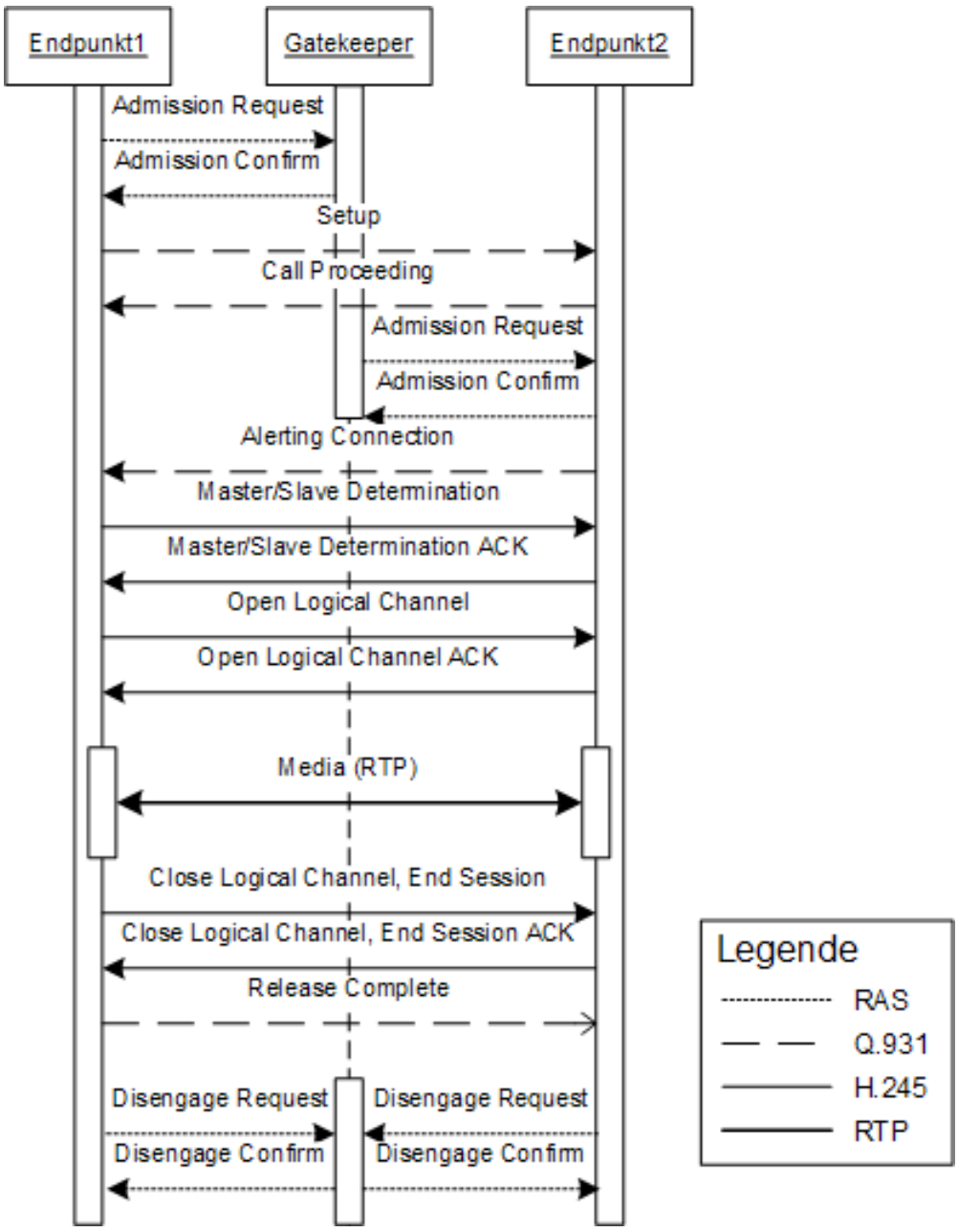

Abbildung 4.6: Ein beispielhafter H.323 Gesprächsablauf (vgl. [Pro, H.323 Protocols Suite] und [Sch03]).

SIP ist im IETF Standard RFC3261 definiert [RSC+02]. SIP ist flexibel und nicht auf VoIP beschränkt. Vielmehr ist es ein Peer-to-Peer Protokoll dessen Funktionalität auf die SIP Infrastruktur verteilt ist [DPB+06, Kap. 12]. Damit lässt es sich auch für Instant Messaging oder Online Spiele verwenden [RTR06, S. 33]. Für die Architektur bedeutet das, dass User Agents sowohl als Clients als auch als Server fungieren. Dennoch beschreibt der Standard Komponenten, welche für den Einsatz in größeren, öffentlichen SIP Netzwerken eingesetzt werden können. Dazu zählen Proxy Server die Anfragen routen können und für Authentifizierung sorgen, Registrare die Registrierungsinformationen über Endgeräte speichern und Redirect Server die Nachrichten umleiten und somit das Netzwerk besser skalieren können. Die SIP Infrastruktur ist in Abbildung 4.7 verdeutlicht. Ähnlich wie H.323 nutzt SIP andere Protokolle, um arbeiten zu können. DNS[19] löst den Hostnamen in eine IP-Adresse auf, SDP[20] beschreibt die (Multimedia-)Parameter einer Session und RPT/RTCP wird für den Medientransport benutzt [DPB+06, Kap. 12]. Im Gegensatz zu H.323 sind alle Server optional, wenn lediglich eine End-to-End Kommunikation mit SIP Clients notwendig ist. Will man das Netzwerk jedoch besser skalieren oder auch eine Schnittstelle zu anderen Protokollen oder Netzen schaffen, müssen diese Server und ein Gateway eingesetzt werden. Der PBX Asterisk ermöglicht all diese Services.

Der Gesprächsablauf ist im Gegensatz zu H.323 (technisch) wesentlich einfacher. Abbildung 4.8 zeigt einen beispielhaften Ablauf bei dem auch ein Redirect Server involviert ist. Dieser teilt dem Endpunkt1 anfänglich die Adresse von Endpunkt2 mit. Jegliche Aushandlung der Medienparameter erfolgt bereits mit der Invite Nachricht.

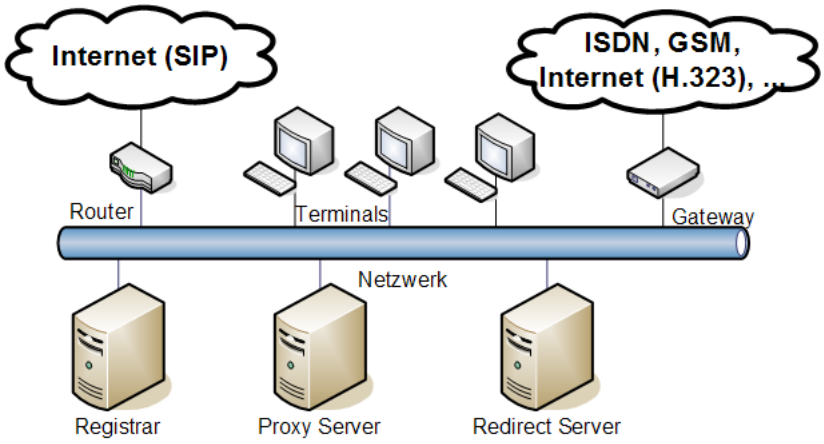

Abbildung 4.7: Die SIP Infrastruktur mit Registrar, Proxy Server, Redirect Server und Terminals als Endpoints.

Das Real-Time Transport Protocol (RTP) sorgt schließlich für die Übertragung der Nutzdaten. Es ist im IETF Standard RFC3550 spezifiziert [SCFJ03]. RTP selbst garantiert noch keine Echtzeitübertragung, für die Überwachung des QoS wird das Real-Time Control Protocol (RTCP) verwendet, welches unabhängig von RTP funktioniert. RTP und RTCP verwenden beide UDP als Transport Layer und benutzen unterschiedliche Ports. Der RTP Packet-Header enthält Informationen, welche der Identifizierung von Daten und Quelle dienen. Dazu gehören eine Sequenznummer des Pakets, ein Zeitstempel, eine Kennzeichnung der derzeitigen RTP Session und die Kennzeichnung der derzeitigen Payload, also der Nutzdaten. Gleichzeitig können obligatorische RTCP Kontrollinformationen geschickt werden. Diese Informationen beziehen sich auf die aktuelle RTP Session und helfen den Datenfluss zu skalieren. Sender und Empfänger können Reports verschicken. So kann beispielsweise die Datenrate bei adaptiven Audiocodecs angepasst oder Video- und Audiodaten synchronisiert werden. Auch die Anzahl der Teilnehmer einer Session wird berücksichtigt, damit die RTCP Datenrate nicht linear mit der Menge der Teilnehmer anwächst.

[19] DNS: Domain Name System
[20] SDP: Session Description Protocol

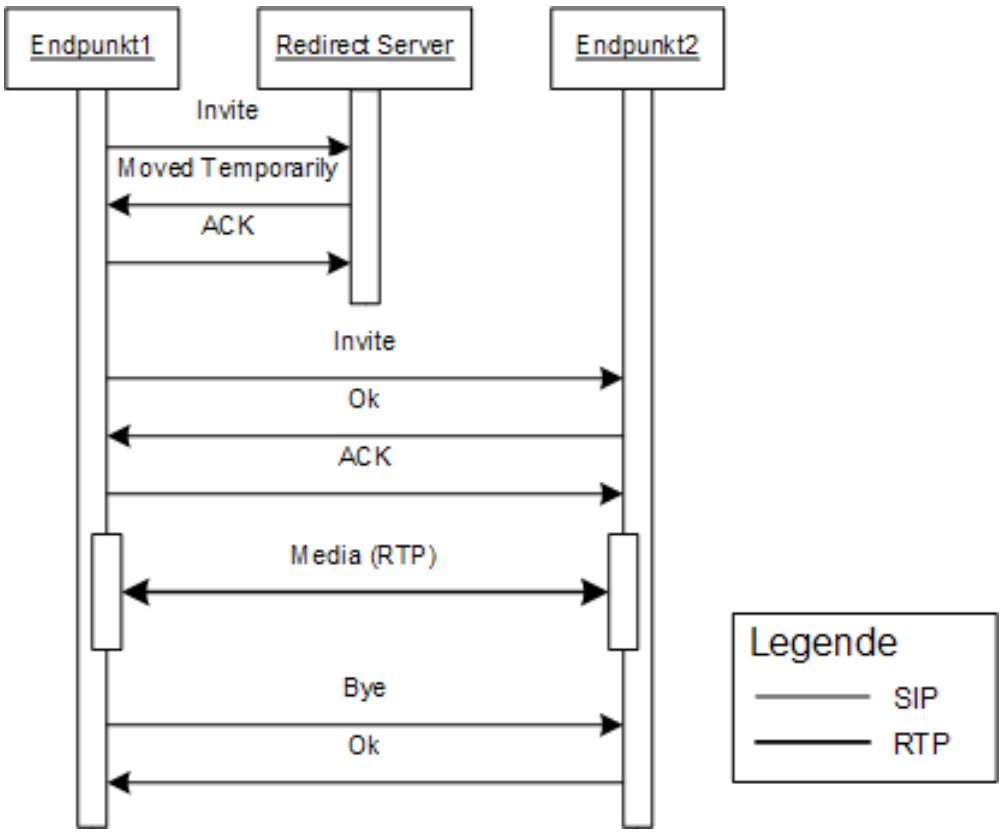

Abbildung 4.8: Ein beispielhafter SIP Gesprächsablauf (vgl. [Wol04, Kap. 5.5.4]).

4.3.4 Resümee und Diskussion der Übertragungstechniken

In der Planungsphase des Projekts hat sich schnell herausgestellt, dass standardisierte Protokolle der Internettelefonie für diesen Zweck nicht geeignet sind. Dies bedeutet jedoch nicht, dass keine echtzeittaugliche VoIP Lösung in Betracht gezogen wurde, sondern, dass die standardisierten Protokolle für eine eigene Implementierung zu aufwändig sind. Wie bereits in der Einleitung erwähnt (siehe Kapitel 1) verläuft die Kommunikation im Gegensatz zu Telefongesprächen nur unidirektional. Dies vermindert die Komplexität erheblich, da gewährleistet ist, dass immer nur eine Seite Daten aussendet, während die andere Seite lauscht. Eine Session Initialisierung verläuft ebenso immer nach dem gleichen Schema, da sich nach der Konfiguration nichts mehr ändert. Die Geräte besitzen fixe IP Adressen, der Verbindungsprozess ist automatisiert. Die Einarbeitung und Entwicklung einer VoIP Lösung mittels H.323 oder SIP würde das Recording von Audiodaten, welches im Grunde auf eine Client-Server Kommunikation hinausläuft, unnötig verkomplizieren.

Eine vorgeschlagene Lösung, um die Entwicklungsarbeit zu reduzieren, war auch der JACK Audio Connection Kit (siehe 4.3.2). Die Lösung sah vor, einen PC mit einer mehrkanaligen Soundkarte auszurüsten. Jeder Eingang sollte einem Funkkanal dienen. Eine darauf laufende Applikation, welche die JACK API implementiert, verschickt die Daten via NetJack zum Server, wo die Daten archiviert werden sollen. Das Verschicken ist notwendig, da eine Archivierung nur auf dem Server erfolgen soll. Diese Lösung hätte einige Vorteile: Es kann Standard Hardware verwendet werden, NetJack übernimmt den Transport von Daten, JACK ist systemunabhängig. Die Nachteile sind dagegen wirtschaftlicher Natur: Die Lösung baut nicht auf der **AviBit** Architektur - bestehend aus Libraries, Datenformaten und Tools - auf, sondern ist etwas externes. Es muss fremder Code verwendet werden der unter einer anderen Lizenz steht (JACK steht unter der GNU GPL sowie LGPL[21]). Wartung und Integrierung von fremden Code

[21] GPL/LPGP: Die GNU General Public License bzw. Lesser General Public License ist eine Lizenz für freie Software, für mehr Informationen siehe: `http://www.gnu.org/licenses/licenses.html`.

ist generell aufwändiger und Kunden äußern Sicherheitsbedenken gegenüber externen Programmen und verlangen umständliche Dokumentationen und Risikoabschätzungen. Demzufolge ist eine eigene Lösung anfänglich zwar aufwändiger, im Nachhinein aber besser wartbar und auf Kundenwünsche besser adaptierbar.

Überhaupt distanzierte man sich in der Planungsphase von einer PC-Workstation, welche die Datenerfassung übernehmen sollte. Es verfestigte sich der Wunsch, dass diese Aufgabe (die Datenerfassung, *nicht* die Speicherung!) eine eigene Hardware übernehmen sollte, die allein diesem Zweck dient. Die groben Anforderungen waren dabei folgende: Das Gerät sollte mehrere Audio Input Schnittstellen (bspw. 3.5 mm TRS Connector) besitzen und als Output die Daten via LAN (bspw. RJ 45) ausschicken. Das Protokoll ist vom Gerät vorgegeben und muss implementiert werden. Auch nach längerer Recherche war es jedoch nicht möglich ein passendes Gerät innerhalb eines vernünftigen Preisniveaus zu finden.

Existierende Lösungen laufen unter dem Namen *Audio over Ethernet*, sind aber vor allem für den Rundfunk oder generelle Beschallungssysteme (Konzerte, Stadien, Flughäfen, etc.) ausgelegt und dementsprechend teuer und aufwändig. Die Firma **Cirrus Logic**[22] bietet mit *CobraNet* eine digitale Audioübertragungslösung an, die auf große Gebiete ausgelegt ist, welche gleichzeitig beschallt werden sollen [Cob]. Die Audiodaten (64 Kanäle, 48 kHz Abtastrate, 20 Bit Samplingtiefe) werden dabei verlustlos über CAT-5- oder Glasfaserkabel übertragen. Eine etwas kleiner skalierte Audio over Ethernet Lösung stellt *EtherSound* von **Digigram**[23] dar. *EtherSound* ist neuer als *CobraNet*, flexibler (nicht nur für Festinstallationen geeignet) und erzielt eine noch geringere Latenz. Die EtherSound Technologie kann lizenziert werden und wird von mehreren Herstellern in ihren Produkten verwendet, beispielsweise von **Barix**[24] und **Yamaha**[25]. Für Entwickler steht ein Software Development Kit (SDK) zur Verfügung. Rothenberg [Rot09] merkt an, dass *EtherSound* für flexiblere Beschallungen (bspw. Konferenzen oder Promotion Veranstaltungen), aber auch für Live Musik geeignet ist. Dies bedeutet aber, dass es für eine simple VoIP Recording Lösung noch immer zu aufwändig ist.

Zu den wirtschaftlichen Aspekten sei noch hinzuzufügen, dass das Audio Recording nur ein Teil eines Produktes, nämlich des *Legal Recordings* (inklusive Video), und dieses wiederum eher ein kleineres Produkt der gesamten Produktpalette ist. Für den Kunden wäre es also nicht nachvollziehbar warum hier eine komplexe Audioinstallation notwendig ist, die zwar eine möglichst niedrige Latenzzeit bei größtmöglicher Kanalbelegung bietet, aber auch dementsprechende Kosten verursacht. Noch dazu im Hinblick, dass simple Lösungen bei sehr geringen Kosten existieren - siehe JACK (4.3.2).

Der eingeschlagene Weg sah also so aus, dass sowohl Hardware als auch Software selbst entwickelt wird, wobei die Hardwareentwicklung an die Firma **Xerxes Technologies** ausgelagert wurde, die **AviBit** aber die Pläne überließ und umfassenden Support zum Produkt liefert. Die Entwicklung der Software fand im Rahmen dieser Diplomarbeit statt.

Das nun folgende Kapitel widmet sich also der praktischen Implementierung der zwei Programme: dem Audiorecorder *AudioRec* zum Archivieren der Audiodaten und dem Media Player *AVPlayer* zur Wiedergabe von Audio- und Videomaterial. Dazu gehört die Auflistung der Anforderungen an die Programme, eine Roadmap zur Durchführung des Projekts, das zu Grunde liegende Kommunikationsprotokoll, sowie eine ausführliche Beschreibung der entwickelten Programme selbst.

[22] http://www.cirrus.com
[23] http://www.digigram.com
[24] www.barix.com
[25] www.yamahaproaudio.com

5 Entwicklung des AviBit Legal Recording and Replay Systems

Unter dem *AviBit Legal Recording and Replay* wird alles verstanden, was Video- und Audiodaten aufzeichnen und später auch wiedergeben kann. Die drei Hauptkomponenten sind dabei *VideoRec* (zur Videoaufzeichnung), *AudioRec* (zur Audioaufzeichnung in Verbindung mit der Hardware XC-1124A, siehe Abschnitt 5.2) und der *AVPlayer*[1], welcher der Wiedergabe dient (siehe Abschnitt 5.3).

Eine Gesamtübersicht dieser Komponenten ist in Abbildung 5.1 ersichtlich. Als Ausgangspunkt dient die Controller Working Position. Hier werden Videodaten am Bildschirm erfasst, sowie Audiodaten vom Funk an die Audio Hardware XC-1124A weitergegeben. Über das interne Netzwerk (gekennzeichnet durch strichlierte Linien) werden diese Daten kodiert an die jeweiligen Prozesse verschickt. Die Prozesse *VideoRec* bzw. *AudioRec* erstellen Archive, welche vom Media Player *AVPlayer* abgespielt werden können. Das Video Capturing mittels *VideoRec* wird kurz in Unterabschnitt 5.1.1 beschrieben.

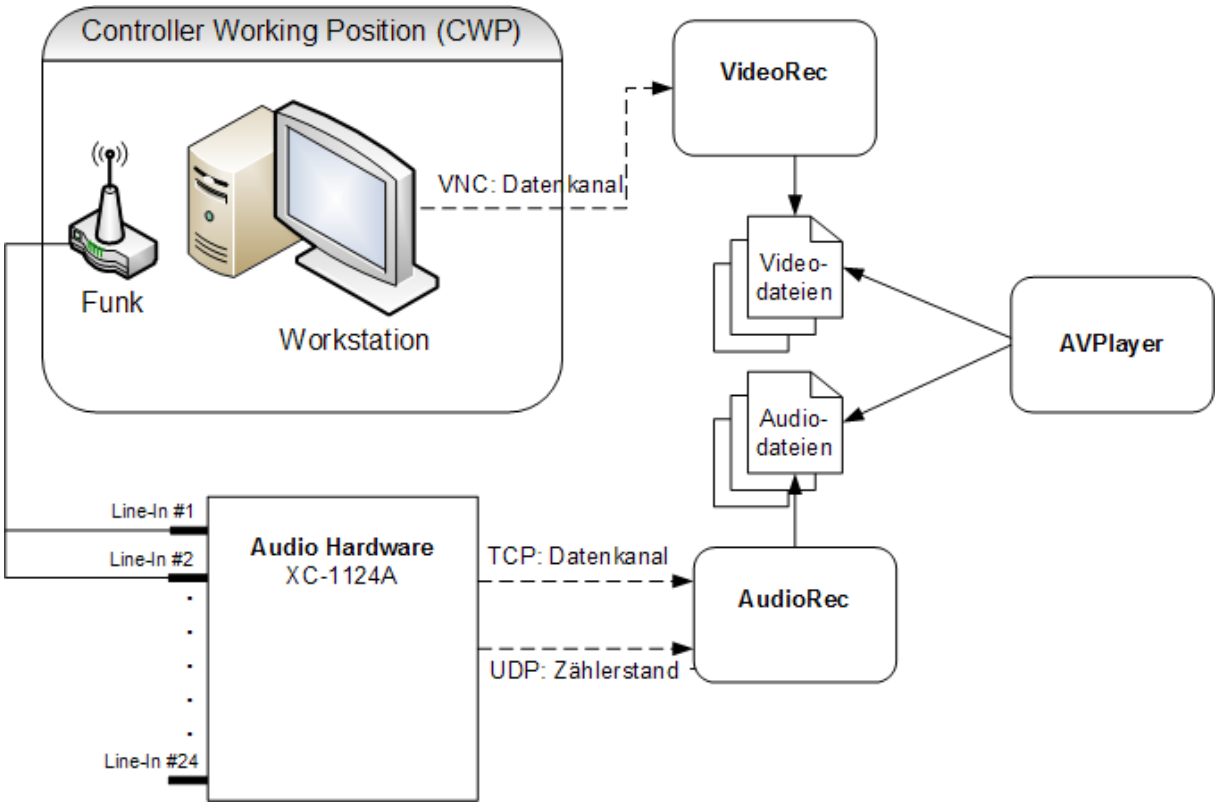

Abbildung 5.1: Legal Recording and Replay Gesamtübersicht

[1] *AVPlayer* steht für Audio/Video Player.

5.1 Allgemeines

Dieser einleitende Abschnitt erläutert zuerst Allgemeines zum Video Capturing mittels *VideoRec* (siehe 5.1.1). Danach werden die Anforderungen an das Programm *AudioRec* und die Erweiterung des Videoplayer *AVPlayer* beschrieben (siehe 5.1.2). Zum Ablauf des Projekts ist in 5.1.3 noch eine Roadmap zu finden, welche einen Plan inklusive Meilensteinen darstellt. Die bei der Implementierung verwendeten Frameworks und Libraries sind in 5.1.4 aufgelistet.

5.1.1 Video Capturing mittels VideoRec

VideoRec wurde 2009 von Matthias Zöhrer für **AviBit** entwickelt. Die Funktionsweise ist folgende: Das Programm verbindet sich via VNC[2] zu der Workstation des Fluglotsen (meist auch als Controller Working Position bezeichnet), holt sich kontinuierlich die Bildschirminhalte via RFB Protokoll[3] und speichert diese in einer **AviBit** Archivdatei. Dieses Dateiformat erlaubt die Speicherung beliebigen binären Inhalts, welcher Quellenidentifikation, Zeitstempel, Kanalnummer und sonstigen zusätzlichen Metadaten versehen werden kann. Somit ist es im Nachhinein eine exakte Wiedergabe der Daten wie zum Aufzeichnungszeitpunkt möglich.

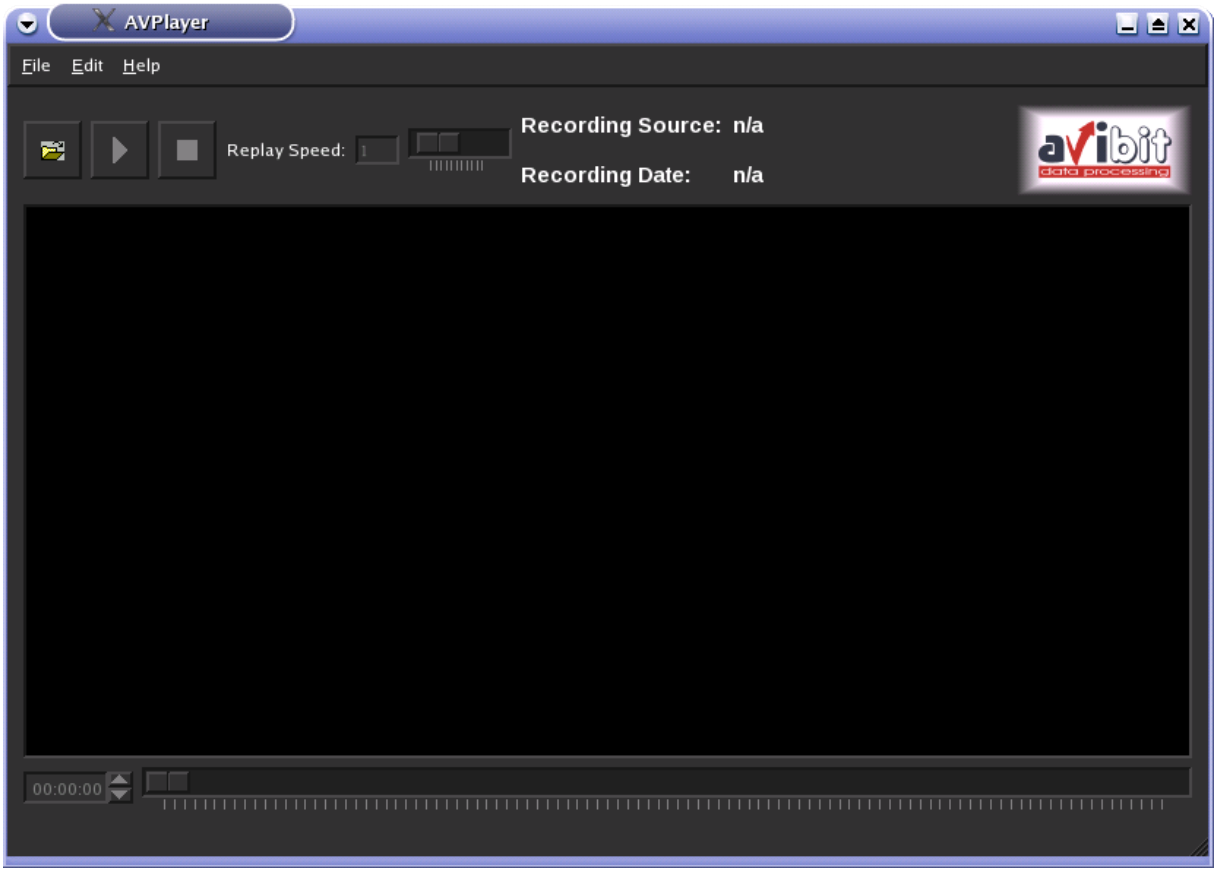

Abbildung 5.2: Die ursprüngliche Benutzeroberfläche des *AVPlayers*

VideoRec läuft als Prozess auf einem Server und besitzt kein Interface zur Steuerung. Die einzige Konfigurationsmöglichkeiten bieten Parameter, welche man beim Programmstart angibt. Zu diesen gehören der VNC Server (also die CWP) und die Portnummer zu der das Programm eine Verbindung

[2]VNC: Virtual Network Computing ist eine Technik die den Fernzugriff auf Computer ermöglicht. Man kann dabei den Bildschirminhalt des anderen Computers sehen, sowie dessen Inputgeräte (Mouse, Tastatur) nutzen. Für weitere Informationen siehe `http://en.wikipedia.org/wiki/Virtual_Network_Computing`.

[3]RFB: Remote Frame Buffer

herstellen soll, ggf. ein Passwort für den Login, der Pfad für die Archivdatei und die Anzahl der Aufnahmedauer in Stunden, bevor ein neues Archiv begonnen wird.

Zur Wiedergabe wurde ebenfalls von Zöhrer die ursprüngliche Version des *AVPlayers* entwickelt. Dieser konnte jedoch nur Video Archive wiedergeben. Die Abbildung 5.2 und 5.3 zeigen zwei Screenshots des Players mit seinem ehemaligen Interface. Die Bedienelemente umfassen Buttons zum Öffnen der Archivdatei, Wiedergabe, Stopp und ein Slider, um die Wiedergabegeschwindigkeit zu ändern. Zusätzlich zum Bild (dafür ist die große, schwarze Fläche reserviert) wird der Name und das Aufnahmedatum des Videos angezeigt, sowie eine Zeitleiste. Im zweiten Screenshot wurde bereits ein Videoarchiv geöffnet. Es wird lediglich eine Bildschirmschoner angezeigt, welcher auf dem entfernten Computer gerade lief. Hinter bzw. unter dem Fenster des *AVPlayers* sieht man die Linux Bash, mit welcher das Programm gestartet wurde. Es wurden ein paar Debug Nachrichten ausgegeben. Das Programm bzw. dessen Erweiterung wird noch ausführlich im Abschnitt 5.3 beschrieben.

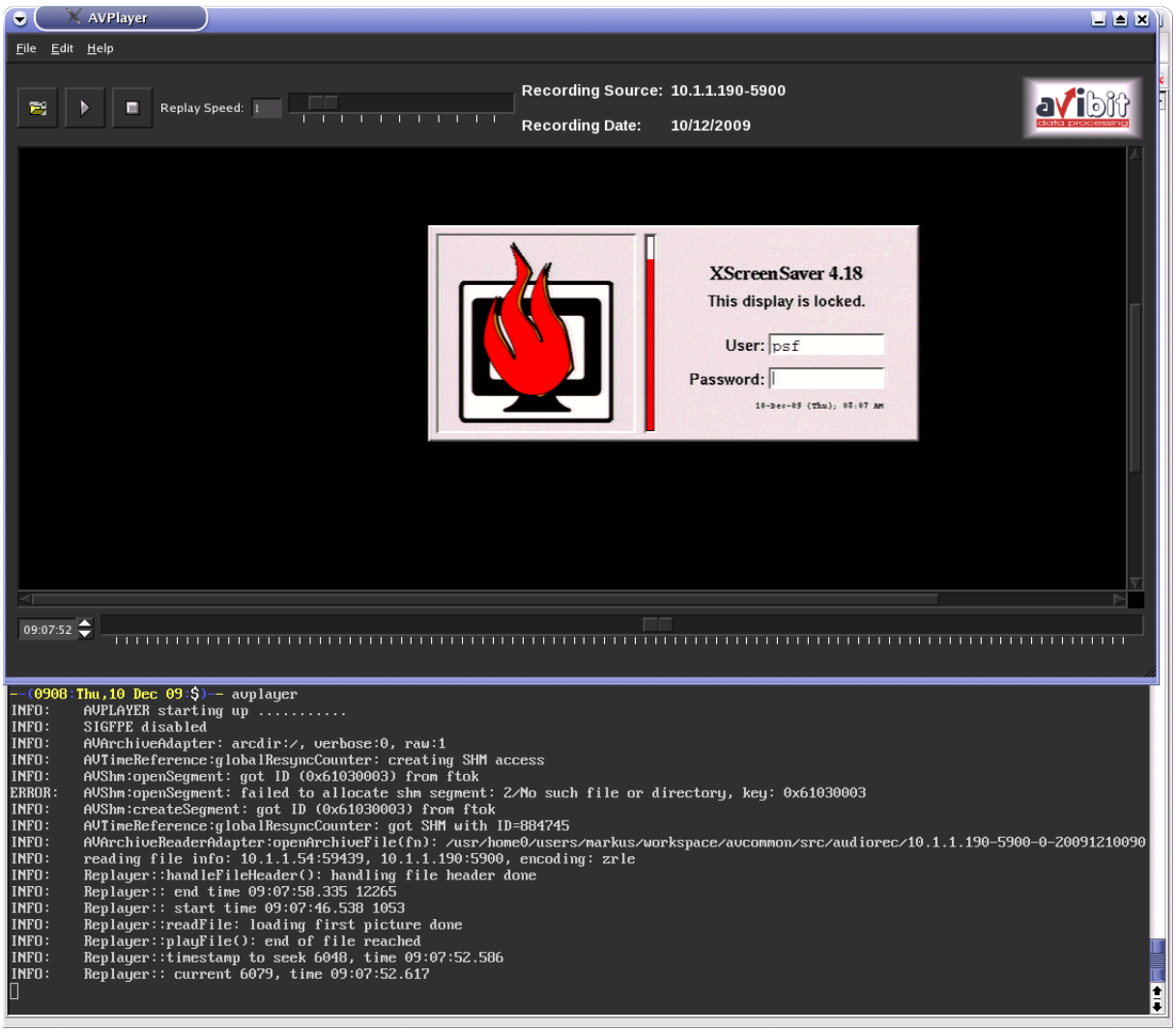

Abbildung 5.3: Hier wurde bereits ein Archiv mit dem *AVPlayer* geladen

5.1.2 Anforderungen

An das Audio Recording wurden anfänglich folgende Anforderungen gestellt, welche bei der Planung und Entwicklung eingehalten werden mussten. Audiodaten sollen über das Netzwerk empfangen werden können. Eine Beschränkung auf eine sinnvolle Anzahl maximaler Quellen ist jedoch notwendig. Weiters werden die Audiodaten zur Zeit der Implementierung von einem Testsystem abgeschickt. Dieses System muss im Nachhinein durch ein anderes (bspw. Hardware-basiertes) System ersetzbar sein. Außerdem sol-

len die Audiodaten lokal (zentralisiert) gespeichert werden können (RAW Recording). Die Speicherung muss so erfolgen, dass die Audiodaten im Nachhinein einer Videoaufzeichnung zuordenbar sind.

Auch an die Audio Wiedergabe wurden Anforderungen gestellt, die im nachfolgenden beschrieben sind. Die Audio Wiedergabe soll in dem bereits vorhandenen Videoplayer integriert werden. Der Benutzer soll auswählen können, welche Audiokanäle zum Video wiedergegeben werden sollen. Audiodaten sollen synchron zum aufgezeichneten Video abgespielt werden. Die Audiodaten sind lokal vorhanden und werden beim Start des Players eingelesen.

5.1.3 Roadmap

Dieses Unterabschnitt beschreibt die Roadmap, welche für die Abwicklung des Implementationsprozesses verwendet wurde. Nach jedem Schritt sind Meilensteine angegeben, die erreicht werden müssen, damit man den Schritt abschließen kann. Jedoch sollten auch immer Schritte zurück möglich sein, falls Änderungen an bereits Implementiertem anfallen. In dem Fall sind die entsprechenden Meilensteine nochmals zu prüfen. Der Prozess ist in Abbildung 5.4 visualisiert.

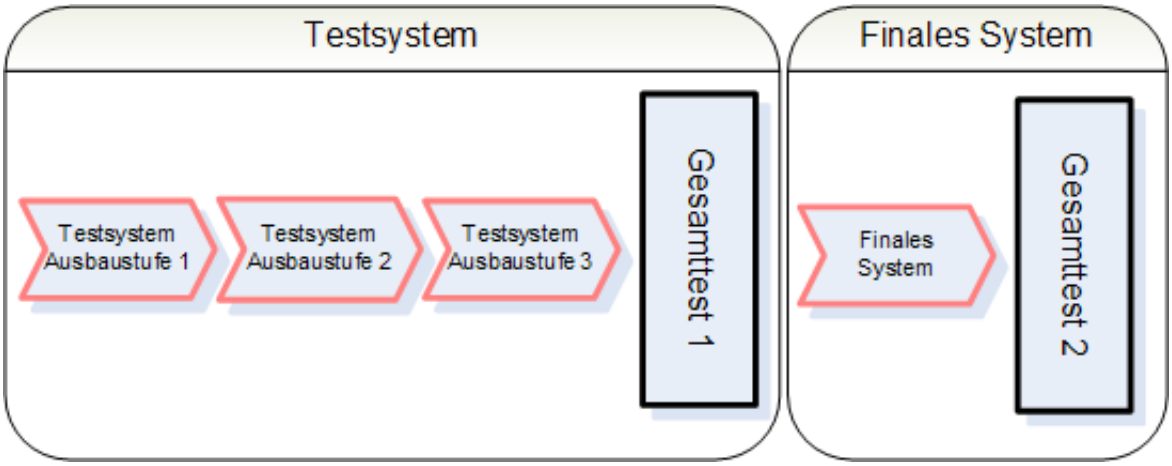

Abbildung 5.4: Visualisierung des Implementationsprozesses

Da die Audio-Hardware XC-1124A erst entwickelt wurde, muss zuerst ein Testsystem aufgebaut werden, welches die Hardware simuliert und Daten für den Input generiert. Der Ausbau des Testsystems erfolgt in drei evolutionären Schritten, welche im folgenden beschrieben sind.

Zuerst wird ein Simulator für den XC-1124A programmiert (siehe dazu auch 5.2.3.3), danach kann der Audio Recorder *AudioRec* entwickelt werden, welcher während der Entwicklung mit Hilfe des Simulators getestet werden kann. Mit Hilfe dieser fortlaufenden Überprüfung kann der Audio Recorder so weit fertig gestellt werden, dass er vollkommen die gestellten Anforderungen abdeckt. Ist dies erreicht, kann an der Arbeit zum Media Player begonnen werden, der ja dann auch Audiodaten wiedergeben soll.

Ziel des Testsystems ist mit Hilfe des Simulators und vorgefertigten Sprachaufnahmen (dazu kann beispielsweise ein Hörbuch dienen) Input für den Audio Recorder zu generieren, welcher die Daten den Anforderungen entsprechend speichert. Danach muss der Player im Stande sein die Audiodaten zusammen mit beliebigen Videodaten abzuspielen.

Nach der Implementierung erfolgt ein Gesamttest, der das System als ganzes über eine längere Zeitdauer testen soll. Die Ergebnisse (vor allem Logmessages in Logfiles) werden überprüft, die aufgenommenen Daten stichprobenartig kontrolliert.

5.1.3.1 Testsystem Ausbaustufe 1

Im ersten Schritt der Ausbaustufe 1 wird der Audio-Hardware Simulator implementiert. Dieser soll anfänglich nur eine Datei einlesen und diese laut Protokollspezifikation per TCP/IP verschicken können.

Dies muss natürlich in Echtzeit funktionieren - als ob die Datei abgespielt wird. Clientseitig lauscht unterdessen nur ein Dummy Prozess und verifiziert, dass Daten ankommen. Man kann die empfangenen Daten mittels eines Hexdumps ausgeben und überprüfen. Der Meilenstein ist die prinzipielle Übertragung einer Datei mittels des Simulators.

Im zweiten Schritt wird der mit der Implementierung des Audio Recorders begonnen. Anfänglich unterstützt dieser nur einen Kanal. Die Herausforderung dabei ist das Extrahieren der Audiodaten aus den TCP Pakete. Die Daten werden lokal in einer WAVE Datei abgespeichert, der Codec wird dabei nicht verändert. Der Schritt ist vollständig, wenn die Datei erfolgreich übertragen werden konnte und dann auch in einem geeignet Audioplayer abgespielt werden kann.

5.1.3.2 Testsystem Ausbaustufe 2

Im ersten Schritt der Ausbaustufe 2 wird der Broadcast zunächst auf mehrere Quellen erweitert. Der Audio Hardware Simulator soll gleichzeitig verschiedene Dateien einlesen und sie laut Protokollspezifikation per TCP/IP versenden. *AudioRec* muss die empfangenen Daten nach Kanälen trennen und dementsprechend abspeichern können. Pro Kanal soll eine Audiodatei (noch immer im WAVE Format) geschrieben werden. Parallel zur Implementierung erfolgt eine Evaluierung des Recorders auf etwaige Fehler und Ausnahmen: Speicherplatzprobleme, Stream reißt ab, Stream ist korrupt, etc.

Danach müssen zwei Meilensteine erreicht werden. Die Dateiübertragung ist auch mit mehreren Dateien erfolgreich und sie lassen sich abspielen, sowie eine angemessene Reaktion auf etwaige Fehler. Diese lassen sich reproduzieren und fließen in die Tests mit ein.

Im nächsten Schritt wird der Recorder so umgebaut, dass die Audiodateien nicht in einen WAVE Container geschrieben werden, sondern in eine **AviBit** Archiv Datei. Der Recorder muss dementsprechend angepasst werden. Alle empfangenen Kanäle sollen inklusive Zeitstempel in ein einzelnes Archiv geschrieben werden. Es sind die dementsprechenden Library Klassen der **AviBit** Library zu verwenden. Der Meilenstein ist das erfolgreiche Schreiben von Audiodaten in eine **AviBit** Archiv Datei.

5.1.3.3 Testsystem Ausbaustufe 3

Die Ausbaustufe 3 des Testsystems behandelt nun erstmals den Media Player *AVPlayer*. Im ersten Schritt muss das ursprüngliche Programm analysiert werden und Ansatzpunkte für die Erweiterung auf Audiowiedergabe gefunden werden. Es müssen etwaige Fehler korrigiert und unnötige Altlasten beseitigt werden. Im ersten Schritt kann auch bereits die GUI[4] angepasst und Features evaluiert werden. Ziel ist es, dass die GUI alle Möglichkeiten bietet, um beliebige Quellen (Audio/Video) auszuwählen und und ab einem beliebigen Zeitpunkt abzuspielen. Außerdem soll sie Bedienungskomfort bieten und intuitiv sein. Die Einschränkung dabei ist, dass noch kein echtes (Audio-)Replay möglich ist.

Schritt zwei verfolgt nun die tatsächliche Implementation der funktionalen Komponenten im *AVPlayer*. Es muss eine geeignete Library zur Wiedergabe von Sound gefunden werden. Mit der Library kann ein Wiedergabegerät (bspw. die Soundkarte) angesprochen und mit Audiodaten versorgt werden. Die G.711 codierten Audiodaten müssen ins PCM Format konvertiert und dann an diese Library übergeben werden. Dies muss kontinuierlich während der Laufzeit des Programms erfolgen. Der Schritt ist erfolgreich abgeschlossen, wenn sich eine oder mehrere Audioquellen parallel zum Video abspielen lassen. Es erfolgt aber noch keine wesentlich Ausnahmebehandlung.

Im dritten Schritt wird die Basisversion des bereits funktionierenden Replays erweitert. Betreffende Punkte sind unter anderem, dass Audio- und Videostream unterschiedliche Anfangs- und Endzeiten haben können, die Stummschaltung einzelner Kanäle während des Replays möglich sein soll, ein nachträgliches Hinzufügen und Entfernen von Archiven und natürlich Sprünge zu bestimmten Zeitpunkten innerhalb eines Archivs. Außerdem erfolgt eine Evaluierung des Players und das Testen auf etwaige Fehler (korrupte Streams, unpassende Streams, fehlerhafte Bedienung, etc.).

[4] GUI: Graphical User Interface

Die zwei Meilensteine dieses Schritts sind die Behandlung von Ausnahmen (z. B. unterschiedliche Anfangs- und Endzeiten der Quellen) und das angemessene Reagieren auf Fehler im *AVPlayer*. Diese lassen sich reproduzieren und fließen in die Tests mit ein.

5.1.3.4 Gesamttest 1

Sind alle Meilensteine erreicht, folgt der Gesamttest des Testsystems. Dazu werden 48 Stunden (zwei Tage lang) Audio Recordings durchgeführt. Diese können dann mit künstlich generierten Videomaterial im Player stichprobenartig abgespielt werden. Es folgt eine Analyse der Logfiles auf etwaige Fehlermeldungen und Warnungen.

5.1.3.5 Finales System

Wenn die Audio-Hardware XC-1124A fertig gestellt, sowie das Testsystem implementiert ist, kann man die Software mit der richtigen Hardware testen. Dazu erfolgt aber erst einmal die Behebung der Fehler nach dem Gesamttest, sowie ein stichprobenartiger Nachtest, um zu gewährleisten, dass die Software auch wirklich funktioniert.

Im ersten Schritt des endgültigen Systems wird zuerst die Konnektivität des XC-1124A mit dem Audio Recorder überprüft. Dann kann mit einem MP3-Player oder ähnlichem die Audio Hardware mit Dateninput versorgt werden, welcher dann im entsprechenden Format verschickt werden soll. Die TCP Daten müssen vom Audio Recorder empfangen und aufgezeichnet werden können. Es ist die Korrektheit der resultierenden **AviBit** Archivdateien zu überprüfen. Gegebenfalls erfolgt eine Fehleranalyse und notwendige Korrekturen des Audio Recorders und/oder der Audio Hardware.

Die Meilensteine sind, dass sich XC-1124A und *AudioRec* verbinden können sowie die Erfassung des Datenstroms. *AudioRec* kann die Daten in der geforderten Form aufzeichnen.

Der zweite Schritt beschäftigt sich dann mit der Überprüfung, ob der Player die Aufnahmen korrekt wiedergeben kann. Es sind sämtliche Konstellationen durchzuspielen. Der Schritt is abgeschlossen, wenn das Replay im *AVPlayer* ohne Probleme möglich ist.

5.1.3.6 Gesamttest 2 und abschließende Analyse

Der zweite Gesamttest verläuft im Grunde wie der erste, nur dass eben hierbei das XC-1124A statt des softwarebasierten Simulators verwendet wird. Als Input können mehrere MP3-Player, PCs oder Stereoanlagen mit abspielenden Medien im Endlosmodus dienen. Danach folgt nochmals eine Evaluierung des Systems. Auffällige Fehler müssen behoben werden. Die offensichtlichen Meilensteine sind, dass jeweils die anfälligen Fehler in den Programmen *AudioRec* und *AVPlayer* behoben sind.

5.1.4 Verwendete Frameworks und Libraries

Zur Programmierung wurden die internen **AviBit** Libraries verwendet, welche auf dem *Qt Framework* aufbauen. Qt (”cute”) ist ein plattformunabhängiges Framework in C++ zum Erstellen von konsolen- und GUI-basierten Applikationen. Es steht in der Version 3.3.8 zur Verfügung, welche die aktuellste Release der dritten Versionsgeneration darstellt (veröffentlicht im Februar 2007). Verwendete **AviBit** Libraries umfassen die generelle `AVLib`, die `AVConfigLib` für die Start-Up Konfiguration, die `AVMsgLib`, welche Zugriff auf das Messaging Framework bietet, die `AVComLib` für die Netzwerkkommunikation und `AVArch`, welche das Archivieren und die Wiedergabe von archivierten Messages erlaubt. Bei **AviBit** ist es üblich den Code mit Doxygen[5] kompatiblen Kommentaren zu versehen. Doxygen stand dafür in der Version 1.3.9.1 zur Verfügung.

[5]Doxygen ist ein Quellcode Dokumentationswerkzeug, siehe `http://www.stack.nl/~dimitri/doxygen`

Desweiteren benutzt der *AVPlayer* zur Soundwiedergabe unter Linux die *ALSA*[6] Library und unter Windows *DirectSound*, welches in **Microsofts** *DirectX* enthalten ist. Das Hilfsprogramm *Audio Hardware Simulator*, welches im Testsystem die Audio-Hardware XC-1124A ersetzt, benutzt außerdem die *libsndfile*[7] in der Version 1.0.21, um WAVE Dateien von der Festplatte einzulesen.

Der nächste Abschnitt beschäftigt sich nun mit der Audio Recording Komponente des Legal Recording Systems. Zusätzlich wird die Audio-Hardware XC-1124A und das zugrunde liegende Kommunikationsprotokoll erläutert.

5.2 Audio Recording Komponente: AudioRec

AudioRec ist dafür verantwortlich die übers Netzwerk geschickten Audiodaten zu empfangen, den Stream in Pakete zusammenzufassen, die Pakete mit Zeitstempeln zu versehen und schlussendlich in eine Archivdatei zu schreiben. Während der Implementation wurde zum Testen der Audio-Hardware Simulator verwendet (siehe 5.2.3.3), welcher später durch die tatsächliche Hardware ersetzt wurde.

5.2.1 Audio-Hardware XC-1124A: Spezifikation und Kommunikationsprotokoll

XC-1124A steht für ”24-Channel Network Audio System”. Die Hardware Komponente bietet 24 Audio Eingänge (RJ 10 Stecker) und einen RJ 45 10/100 MBit/s Ethernet Ausgang. Zusätzlich befindet sich auf der Hinterseite ein Stromstecker und ein USB 2.0 Port, welches als Zugriff auf das Gerät verwendet werden kann, im laufenden Betrieb aber weiter nicht wichtig ist. An der Vorderseite befindet sich ein kleiner LCD Display für das Menü und Statusanzeige, sowie ein Dreh- und Druckknopf zur Bedienung.

Fotos des Geräts sind in den folgenden Abbildungen zu sehen. Abbildung 5.5 zeigt das XC-1124A in einem 19” Rackmount Gehäuse von vorne. In Abbildung 5.6 ist die detaillierte Vorderansicht mit Display und dem Dreh- und Druckknopf Bedienelement ersichtlich. Der Display zeigt an, dass gerade eine Netzwerkverbindung besteht. Er gibt auch Auskunft über aktive (”0”) und inaktive (”_”) Kanäle. In Abbildung 5.7 ist das Gerät schließlich noch von hinten zu sehen. Diverse Elemente sind im Bild nummeriert. Bei den Zahlen befinden sich: 1 - der Stromstecker und Kippschalter zum Ein-/Ausschalten, 2 - 24 RJ 10 Audio Eingänge, 3: der RJ 45 Ethernet Port, 4 - der USB 2.0 Port

Abbildung 5.5: Das XC-1124A in einem 19” Rackmount Gehäuse

[6] ALSA: Advanced Linux Sound Architecture, http://www.alsa-project.org
[7] http://www.mega-nerd.com/libsndfile

Abbildung 5.6: Detaillierte Vorderansicht mit Display und Bedienelement

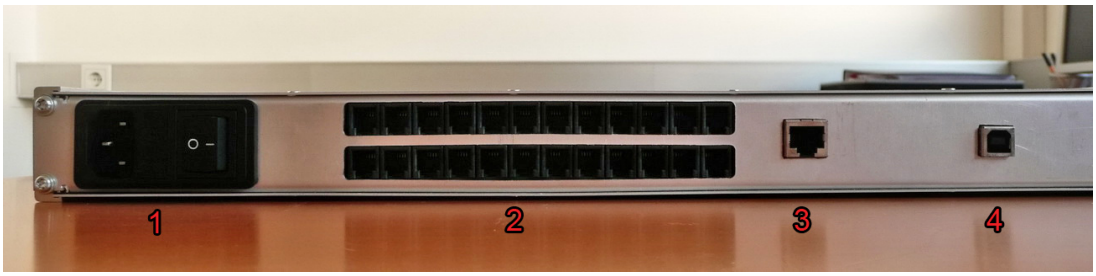

Abbildung 5.7: Hinterseite des XC-1124A mit Ein- und Ausgängen

Das XC-1124A ist mit einem ARM-9 Microcontroller mit 256 kb FLASH und 96 kb SRAM Speicher ausgerüstet [Hof, S. 7]. Mit diesen Spezifikation lässt sich natürlich kein Betriebssystem realisieren. Das bedeutet jedoch auch, dass es keine Implementierung des TCP/IP Stacks gibt, was bei der Netzwerkkommunikation zu Problemen führte. Für eine genauere Beschreibung siehe 5.2.3. Der Microcontroller reicht jedoch aus, um die hereinkommenden Audiodaten zu digitalisieren, encodieren, serialisieren und schließlich via TCP/IP zu verschicken. Um Bandbreite und Speicherplatz zu sparen fiel die Entscheidung für die Audio Kompression auf den A-Law G.711 Codec (siehe auch 4.2.2.1).

Das XC-1124A lässt sich auch auf Stilleerkennung konfigurieren. Für jeden Kanal kann ein Grenzwert in dB angegeben werden, ab dem Stille herrschen soll. Wird dieser Wert unterschritten, so werden für den betreffenden Kanal auch keine Audiodaten mehr gesendet. Es ist anzunehmen, dass beim Funk viele (Sprach-)Pausen auftreten. Dies hilft Bandbreite und Speicherplatz zu sparen. Verwendet man für die G.711 eine Sampling Rate von 8.000 Hz und mono, so entspricht das einer Datenrate von 64 kbps. Das sind 8 Byte pro Millisekunde. Rechnet man das hoch und nimmt die volle Auslastung (alle 24 zur Verfügung stehenden Kanäle an), so kommt man bei durchgängigem Datenverkehr auf beachtliche 15,45 GB pro Tag! Um eben das zu vermeiden sollen während der Funkstille keine Daten übertragen und aufgezeichnet werden.

Wie bereits in Abbildung 5.1 ersichtlich, benutzt das XC-1124A zwei Kommunikationsverbindungen. Audiodaten werden via TCP, der aktuelle Zählerstand, welcher zur Echtzeitsynchronisierung der Daten notwendig ist, wird via UDP verschickt. Zwar hat die Analyse in Abschnitt 4.3 ergeben, dass echtzeitintensive Daten über UDP verschickt werden, da UDP schneller ist, doch in diesem Fall werden die Daten nicht sofort verwendet (niemand muss sie gleich hören können oä.). Stattdessen besitzen die TCP Datenpakete einen Zähler (Counter), welcher auf eine Millisekunde genau ist und jede Millisekunde Audiodaten beziffert. Dieser Counter wird auch regelmäßig (ca. alle zehn Sekunden) via UDP ausgesandt. Der nun über UDP Verbindung empfangene Counter kann zu einem realen Zeitpunkt gemapped werden. Damit ist es dem Programm *AudioRec* möglich den TCP Counter der Audiodaten auf einen Zeitpunkt zu mappen, welche mit Hilfe des UDP Counters ermittelt wurde.

Ein Beispiel für das UDP - TCP Counter Mapping ist in Tabelle 5.1 verdeutlicht. Ein UDP Counter wird, sobald er empfangen wurde, mit einem Zeitpunkt verknüpft (erste Spalte). Jede Millisekunde

Audiodaten ist ebenfalls mit einem Zählerstand versehen (der TCP Counter, zweite Spalte). Damit ist im Nachhinein eine genaue Zuordnung der Daten an einen realen Zeitpunkt möglich (dritte Spalte): Für einen TCP Counter wird der nächstliegende UDP Counter gesucht, welcher mit einem Zeitpunkt verknüpft ist. Dann braucht man nur noch die entsprechende Anzahl Sekunden und Millisekunden hinzufügen bzw. abziehen.

XC-1124A-intern ist der UDP und der TCP Counter natürlich ein und die selbe Zahl. Sie ist vier Byte lang und ist somit vom Datentyp `unsigned integer`. Der Zähler hat nach ca. 49,7 Tagen durchgängigen Betrieb einen Overflow und beginnt dann wieder bei 0. Dieser Aspekt musste natürlich in der Implementation berücksichtigt werden.

Bei der Datenverarbeitung auf Netzwerkbasis ist es üblich die *Network Byte Order* zu verwenden. Diese ist gleichzusetzen mit der *Big-Endian* Notation. Dabei steht das höchstwertigste Byte am Anfang und entspricht somit einem natürlichen Aufbau analog den Dezimalzahlen. Die 16 Bit Zahl 2560 entspricht somit der Binärzahl 00001010 00000000 bzw. in Hexadezimalnotation 0x0A 0x00. Dieser Aufbau ist bei der Extraktion von Zahlen zu berücksichtigen.

Der UDP Counter is nun laut Protokoll ([Hof, S. 9]) ein acht Byte langer Wert. Die ersten vier Byte sind eine immer gleich bleibende *Magic Number*, die zur Identifizierung dient. Die Byte in Hexadezimalnotation sind 0x54 0x49 0x4D 0x45, welche die Buchstaben `TIME` repräsentieren. Werden diese vier Byte erkannt, folgt in den nächsten vier Byte der eigentliche Wert des Counters. Dieser muss extrahiert werden. Der UDP Counter ist in Abbildung 5.8 ersichtlich.

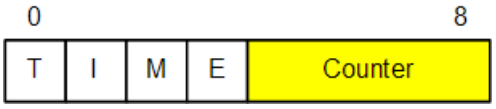

Abbildung 5.8: Der Aufbau des acht Byte lange UDP Counters

UDP Counter ⇔ Zeitpunkt	TCP Counter	Audiodaten Zeitpunkt
1000 ⇔ 12:00:00.000		
	1001	12:00:00.001
	1002	12:00:00.002
	1003	12:00:00.003
	...	...
	1200	12:00:00.200
	1201	12:00:00.201
	...	...
	8002	12:00:08.002
11000 ⇔ 12:00:10.000		
	12000	12:00:11.000
	...	...
	15433	12:00:15.433
...	...	...

Tabelle 5.1: Ein beispielhaftes UDP - TCP Counter Mapping mit verschiedenen Zeitpunkten

TCP Pakete enthalten Zähler-Werte und Audiodaten, die aus dem Paket extrahiert werden müssen [Hof, S. 2f]. Ein Paket ist maximal 1000 Byte lang. Die ersten zwei Byte geben die tatsächliche Länge an. Diese müssen also zuerst in eine 16 Bit Zahl (`unsigned short integer`) umgewandelt werden. Der schematische Aufbau eines solchen Pakets ist in Abbildung 5.9 ersichtlich. Ist nun die Länge bekannt, können die eigentlichen Nutzdaten extrahiert werden. Jede Millisekunde Audiodaten ist durch einen nachfolgenden Counter gekennzeichnet. Wenn kein einziger Kanal aktiv ist, wird auch kein Counter gesendet. Das Schema wiederholt sich für den kompletten Datenfluss und ist in Abbildung 5.10 beispielhaft gezeigt. Dort hat der hinterste Counter nicht mehr als ganzes im ersten Paket Platz und muss

somit teilweise auf das nächste aufgeteilt werden. Es ist also zu beachten, dass ein TCP Paket an *jeder* Stelle unterbrochen werden kann, da ein TCP Paket eine fixe maximale Länge hat. Die Implementierung muss also gewährleisten, dass sich Werte auch über zwei Pakete erstrecken können.

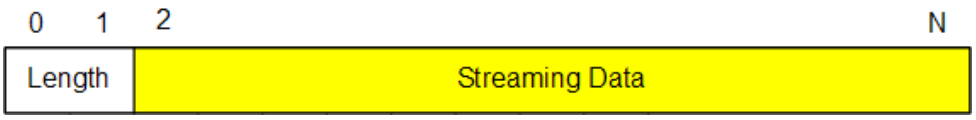

Abbildung 5.9: Ein TCP Paket der Länge N

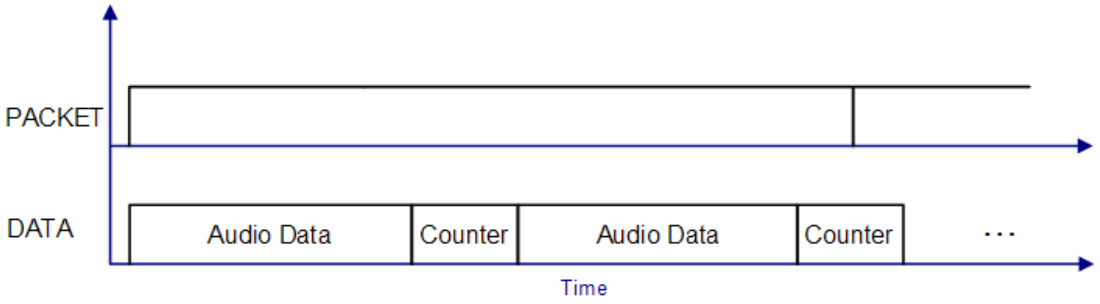

Abbildung 5.10: Ein beispielhafter TCP Paket Datenstrom

Eine G.711-komprimierte Millisekunde mit einer Sampling Rate von 8.000 Hz ist acht Byte lang. Da das XC-1124A aber bis zu 24 Kanäle gleichzeitig verarbeiten muss und nicht über ausreichend Buffer Speicher verfügt, ist jedes Byte einer Millisekunde mit einer Kanalnummer versehen, damit die Daten beim Empfangen eindeutig einem Audiokanal zuordenbar sind. Diese sind nun so angeordnet, dass zuerst die Kanalnummer folgt, dann das erste Byte dieses Kanals. Danach folgt die nächsthöhere aktive Kanalnummer und dessen erstes Byte, usw.

Ein Beispiel mit drei Kanälen ist in Abbildung 5.11 angegeben. AD kennzeichnet ein Byte Audiodaten, CH eine Kanalnummer, die maximal ein Byte groß ist. Die benutzten Kanäle sind 1, 2 und 3. Auf jede dieser drei Kanalnummern folgt je ein Byte Audiodaten. Dies wiederholt sich acht Mal, dann ist die Millisekunden voll. Danach folgt ein Counter, der den Daten einen Zeitwert - auf die Millisekunde genau - zuordnet (nicht im Bild).

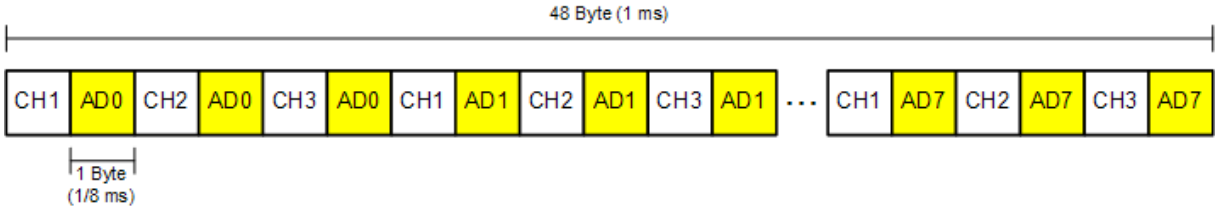

Abbildung 5.11: Anordnung der Audiodaten mit drei aktiven Kanälen

Ein TCP Counter innerhalb der Nutzdaten des TCP Pakets ist acht Byte lang. Sein Bit-genauer Aufbau ist in Abbildung 5.12 verdeutlicht und optisch in vier zwei Byte lange Stücke aufgeteilt. Bit 15 bis 10 sind immer gesetzt. Daran kann beim Parsen des Datenstroms der Counter von den Audiodaten unterschieden werden. Bit 7 bis 0 sind die eigentlich Counter Informationen und müssen zu einer vier Byte langen Zahl (`unsigned integer`) zusammengesetzt werden. Bit 9 und 8 zeigen dabei die Position an (0 bis 3) und dienen somit zur zusätzlichen Verifikation.

Die Audiodaten inklusive der Kanalnummer sind zwei Byte lang. Sie besitzen drei ausgenullte Bit am Anfang, gefolgt von der fünf Bit langen Kanalnummer und danach ein Byte Audiodaten codiert als A-Law G.711. Jedes Byte Audiodaten ist also eindeutig einem Kanal zugeordnet. Für eine Abbildung siehe 5.13.

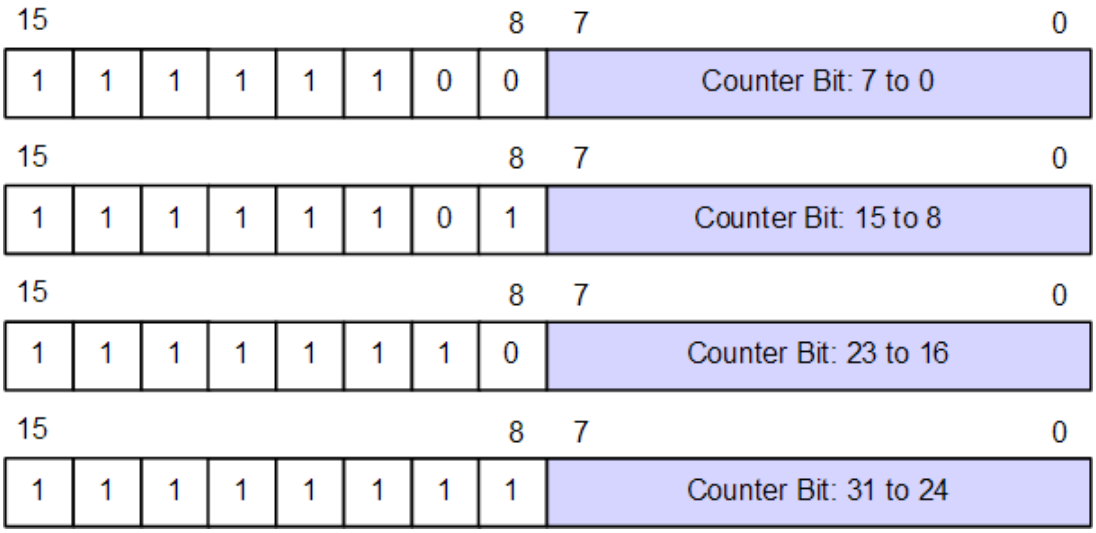

Abbildung 5.12: Der Aufbau des acht Byte langen TCP Counters

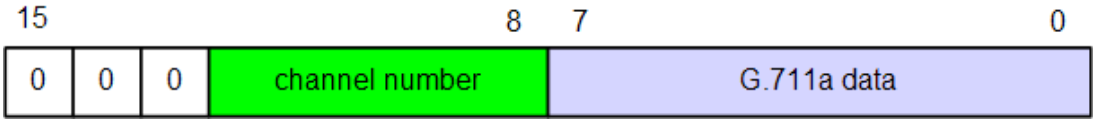

Abbildung 5.13: Der Aufbau des zwei Byte langen Audiodaten Stücks

5.2.2 Architektur und Funktionsweise

AudioRec stellt zum XC-1124A zwei Netzwerkverbindungen (TCP und UDP) her und ist somit der Client, während das XC-1124A der Server ist. Per default benutzt die TCP Verbindung Port 8070 und die UDP Verbindung Port 8071. Dies ist jedoch frei konfigurierbar (siehe 5.2.3).

Zur Übersicht sind die netzwerk- und konfigurationsrelevanten Klassen im UML[8] Diagramm 5.14 dargestellt. Aus Platzgründen sind hier jedoch nicht alle Methoden und Member der jeweiligen Klassen enthalten.

Die Klasse `Recorder` ist ein `QObject`. Das bedeutet, sie unterstützt das Qt-eigene Signal/Slot Handling. Dabei können Signale an Slots gesendet werden und die Klasse kann auf Ereignisse reagieren. In diesem Fall erhält der `Recorder` Signale von den Netzwerkklassen `AVDataTransportUDP` und `RecorderTCPDataTransport`, wenn eine Verbindung hergestellt wird, wenn eine Verbindung beendet wurde, wenn es einen Verbindungsfehler gab und wenn Daten am Socket empfangen wurden. Außerdem benutzt der `Recorder` intern `AVTimer` (siehe auch QTimer [Tro05, QTimer]). Diese arbeiten wie Wecker und senden ein Timeout-Signal aus, wenn die Zeit abgelaufen ist. Beispielsweise ist der Member `m_write_data_timer` dafür verantwortlich regelmäßig ein Timeout zu generieren, worauf der Slot `slotWriteAudioData()` reagiert und die gepufferten Audiodaten in ein Archiv zu schreiben. Diese Zeit kann der Benutzer in der Start-Up Konfiguration vorgeben (siehe 5.2.3) und beträgt per default 5.000 ms.

[8] UML: Unified Modeling Language

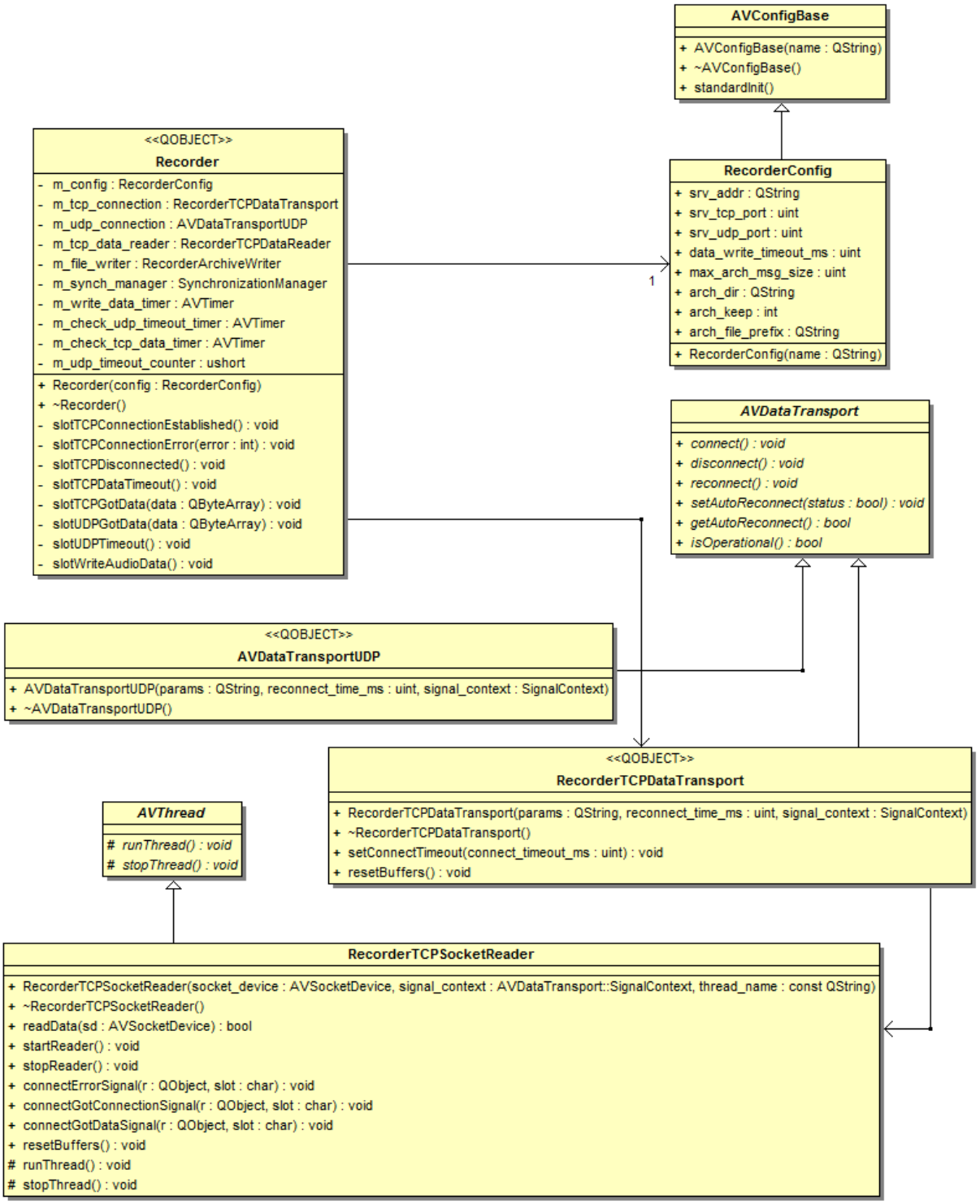

Abbildung 5.14: *AudioRec* Klassendiagramm: Netzwerk und Konfiguration

Wie im Diagramm 5.14 ersichtlich, existiert eine Interface-Klasse `AVDataTransport`. Diese stammt aus der **AviBit** Bibliothek. Konkrete Implementation sorgen für die Abwicklung des jeweiligen Netzwerkprotokolls, beispielsweise `AVDataTransportUDP` für das UDP Protokoll und `AVDataTransportTCP` für das TCP Protokoll. Leider konnte `AVDataTransportTCP` nicht verwendet werden, wegen der nicht standard-konformen Implementierung des TCP Stacks im XC-1124A. Das XC-1124A braucht für jedes TCP Paket eine sofortige Bestätigung (TCP Acknowledge), welches aber üblicherweise erst einige Millisekunden später gesendet wird. Für den Audio Recorder musste also die Klasse `AVDataTransportTCP` angepasst werden. Da diese Änderung aber lediglich projektspezifisch ist, wurde eine eigene Klasse erstellt, welche nicht Teil der **AviBit** Bibliothek ist, sondern nur Teil des `AudioRec` Programms. Das TCP Acknowledge wird nun mit Hilfe der Klasse `RecorderTCPSocketReader` geschickt. Diese ist ein Thread (implementiert `AVThread`) und liest Daten aus dem TCP Socket aus. Die relevante Codezeile ist in Listing 5.1 zu sehen. Mit Hilfe der Funktion `setsockopt(..)`[9] kann das `TCP_QUICKACK` Flag gesetzt werden. Dieses Flag ist jedoch flüchtig und muss jedes Mal erneut gesetzt werden, wenn neue Daten vom Socket eingelesen werden. Wie zu sehen ist, wird diese Zeile nur in Linux Betriebssystemen ausgeführt. In Windows ist das nicht notwendig, da man Windows auf ein schnelles TCP Acknowledge konfigurieren kann.

Die Konfigurationsklasse `RecorderConfig` (eine Ableitung der AviBit Bibliotheksklasse `AVConfig`) hält die Parameter der Start-Up Konfiguration (siehe 5.2.3) als `public` Members. Hiervon liest der `Recorder` die Werte aus, welche für die Verbindung benötigt werden (IP Adresse, Port Nummern) sowie zum Schreiben von Archiven (Schreibzyklus, Speicherort, Dateinamenpräfix, etc.).

```
61
62 bool RecorderTCPSocketReader::readData(AVSocketDevice& sd)
63 {
64     if (m_reader_stop_flag) return false;
65
66 #ifdef Q_OS_LINUX
67     // sets the TCP_QUICKACK flag
68     setsockopt(sd.socket(), IPPROTO_TCP, TCP_QUICKACK, (int[]){1}, sizeof(int));
69 #endif
```

Listing 5.1: *recorder_tcp_socket_reader.cpp:* Setzen des `TCP_QUICKACK` Flags

Die `main(..)` Funktion zum Starten von *AudioRec* ist in Listing 5.2 zu sehen. Die Funktion `AVDaemonInit(..)` (Zeile 110) initialisiert das Programm und den Logger, welcher Debug Meldungen, Informationen, Warnungen und Fehler auf die Konsole oder in eine Log-Datei schreiben kann. Die Konfigurationsparameter werden in einer Hilfsfunktion auf ihre Gültigkeit geprüft (Zeile 116). Danach muss lediglich noch der `Recorder` konstruiert (Zeilen 120 - 121), sowie der Process State initialisiert werden (Zeile 124). In Zeile 127 wird danach die Event Loop gestartet, womit die `main(..)` Funktion in Zeile 128 verharrt. Die Event Loop sorgt nun für den weiteren Ablauf des Programms. Die Funktionsweise ist dem Observer Pattern ähnlich. Viele Objekte sind lediglich durch Signals und Slots verbunden. Die Event Loop sorgt nun dafür, dass die Signale an den richtigen Slot weitergeleitet werden. Durch das Beenden der Event Loop wird die `main(..)` Funktion weiter ausgeführt (Zeile 129 bis zum Ende), womit das Programm gänzlich terminiert.

```
105 int main(int argc, char* argv[])
106 {
107     // init application and logger
108     QApplication app(argc, argv, false);
109     AVConfig::setApplicationName(AVConfig::APP_VVRMAX);
110     AVDaemonInit(UseEventLoop, PROCNAME, "", true);
111
112     // setup configuration
113     const RecorderConfig *cfg = new RecorderConfig();
```

[9] Für mehr Informationen über die Funktion siehe `http://linux.die.net/man/2/setsockopt`.

```
114     AVASSERT(cfg);
115
116     if (!checkConsoleParameters(*cfg))
117         AVLogger->Write(LOG_FATAL, "Could not start %s", PROCNAME);
118
119     // recorder
120     Recorder *recorder = new Recorder(cfg);
121     AVASSERT(recorder);
122
123     // init the process state
124     AVDaemonProcessStateInit();
125
126     // run the event loop
127     int rc = qApp->exec();
128
129     AVLogger->Write(LOG_INFO, PROCNAME" stopping");
130
131     // we're terminated here -> clean up
132     AVDEL(recorder);
133     AVDEL(cfg);
134     AVDaemonDeinit();
135
136     return rc;
137 }
```

Listing 5.2: *recorder_main.cpp:* `main(..)` Funktion von *AudioRec*

Die operationelle Sichtweise von *AudioRec* ist in Abbildung 5.15 dargestellt. Auch hier sind aus Platzgründen nicht alle Methoden und Member der jeweiligen Klassen abgebildet.

Der `Recorder` benutzt die drei Klassen `RecorderTCPDataReader`, `SynchronizationManager` und `RecorderArchiveWriter`, um die empfangenen Daten zu managen und eine Klasse (`AudioData`), um Daten zu repräsentieren.

Die Klasse `AudioData` hält alle Daten, welche für die Audioverarbeitung relevant sind. Die Daten können bequem im Constructor gesetzt werden und nachher via Getter und Setter abgefragt und verändert werden. Zum Debuggen kann das ganze Paket mittels `dumpPacket()` auf der Konsole ausgegeben werden. Die nachfolgenden Member repräsentieren die Daten von `AudioData`. `m_channel_number` ist die Kanalnummer im Bereich 1 - 24. `m_counter` ist der TCP Counter als `unsigned integer` mit 32 Bit. `m_time_stamp` ist ein Zeitstempel vom Typ `QDateTime` [Tro05, QDateTime]. Dieser kann hinzugefügt werden, wenn der TCP Counter mit Hilfe des UDP Counters auf einen realen Zeitpunkt gemapped werden kann. Das `QByteArray` [Tro05, QByteArray] `m_payload_data` enthält die zusammenhängenden Audiodaten für dieses `AudioData` Objekt.

`RecorderTCPDataReader` muss ein TCP Paket von Daten parsen und zerlegen können. Das Grundprinzip ist Raw-Daten vom Socket (ein `QByteArray`) zu übergeben und dafür eine Liste (eine `QPtrList` [Tro05, QPtrList]) von `AudioData` Objekten zu erhalten. Dies erfolgt mit der Methode `getPackets(..)`. Der erste Parameter is ein *in* Parameter, die Klasse erhält die Daten, parsed sie und versucht die Daten zu extrahieren. Der zweite Parameter, ein *out* Parameter, enthält eine Liste von `AudioData` Objekten welche aktuell aus den Raw-Daten erstellt werden konnten. Intern muss die Klasse Bytes puffern, die noch nicht zu einem `AudioData` Objekt zusammengebaut werden konnten.

Der `SynchronizationManager` hat die wichtige Aufgabe Audiodaten zu sortieren und mit der Echtzeit in Verbindung zu bringen. Als Input "füttert" man ihn mit TCP Daten (`feedAudioDataPackets(..)`) und UDP Daten (`feedUDPCounter(..)`). Die TCP Daten sollten nun jene Liste sein, welche der `RecorderTCPDataReader` aus den Raw-Daten extrahiert hat. Der UDP Counter wird direkt vom `Recorder` übergeben, mitsamt einem Zeitstempel, wann dieser empfangen wurde. Intern verwaltet der `SynchronizationManager` Datenstrukturen, um die Liste der Audiodaten nach Kanälen zu ordnen, die letzten aktuellen UDP Counter zu halten, sowie Audiodaten mit Zeitstempel in Verbindung zu bringen. Außerdem kontrolliert der `SynchronizationManager` die Counter Werte auf einen Overflow, welcher nach ca. 49,7 Tagen durchgängigen Betriebs stattfindet.

Diese Situation wird automatisch behandelt, damit ein reibungsloser Ablauf garantiert wird.

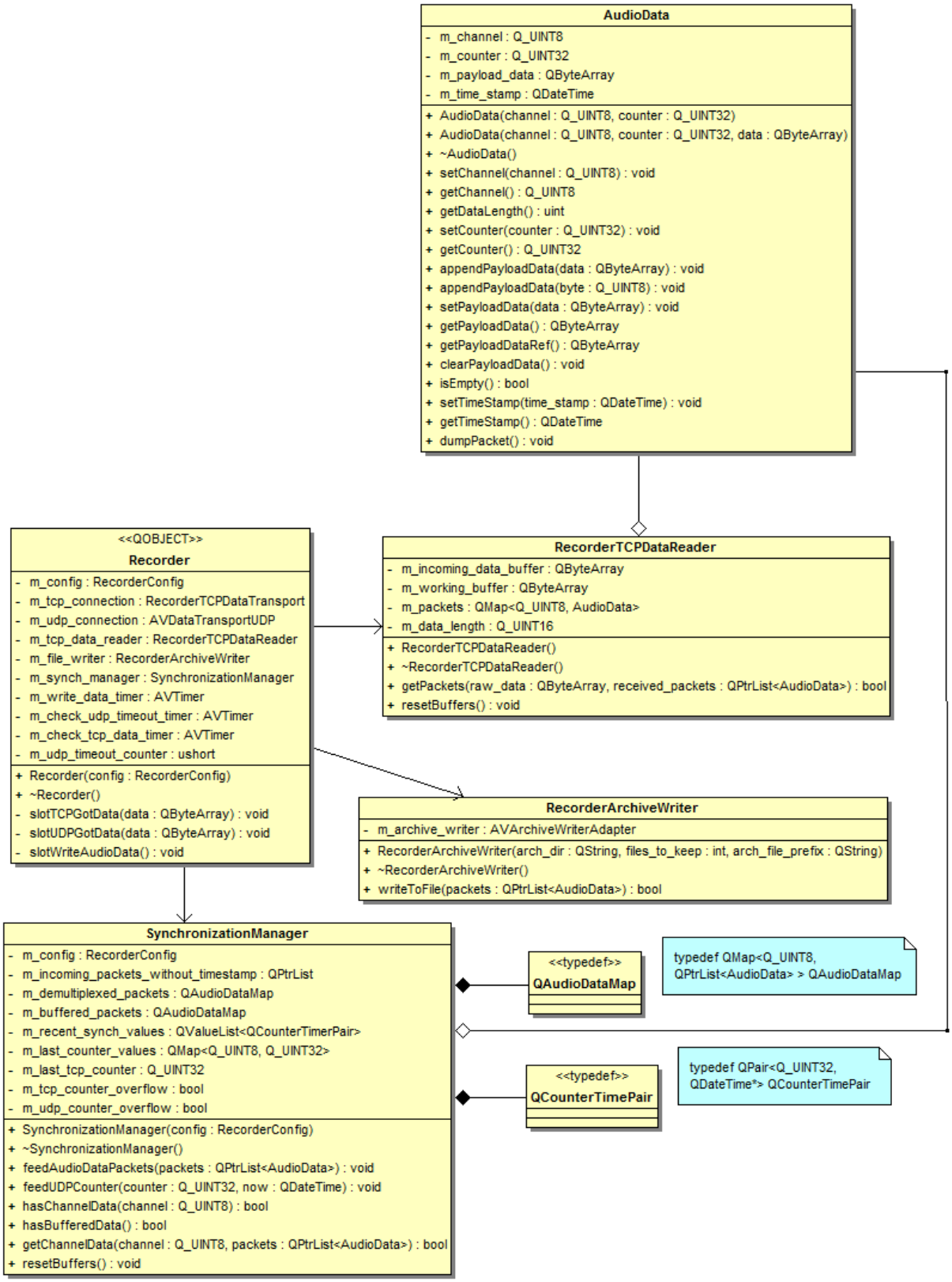

Abbildung 5.15: *AudioRec* Klassendiagramm: Operationelle Ansicht

Die Klasse `RecorderArchiveWriter` ist für das Schreiben von Audiodaten in das Archiv zuständig. Je nach konfiguriertem Schreibzyklus (siehe 5.2.3) werden die vorbereiteten Audiodaten vom `SynchronizationManager` abgefragt und dem `RecorderArchiveWriter` übergeben werden, der diese in eine Archivdatei schreibt. Mit der Methode `getChannelData(..)` erhält man für einen gewissen Kanal alle verfügbaren `AudioData` Objekte in einer `QPtrList` vom `SynchronizationManager`. Diese Liste braucht nur noch dem Archive Writer übergeben werden (Methode `writeToFile(..)`). Der Archive Writer bedient sich intern der **AviBit** Implementation `AVArchiveWriterAdapter`, der wiederum seinerseits Klassen aus der **AviBit** Archiving and Replaying Bibliothek `AVArch` benutzt.

Der `Recorder` selbst übernimmt die leitende Rolle innerhalb des *AudioRec* Programms. Er beschäftigt alle anderen Klassen und arbeitet nur auf Signal/Slot Ebene. Meistens werden Daten weitergereicht, eine Interpretierung findet selten statt. Als Beispiel dafür ist die Methode `slotGotTCPData(..)` in Listing 5.3 angeführt. Dieser Slot wird immer dann aufgerufen, wenn am TCP Socket neue Daten empfangen wurden.

```
130 void Recorder::slotTCPGotData(const QByteArray &data)
131 {
132     // check for end of day
133     if (QDateTime::currentDateTime().time().secsTo(QTime(23, 59, 59)) < 3)
134     {
135         handleEndOfDay();
136         return;
137     }
138
139     QPtrList<AudioData> tcp_packets;
140
141     // parse data and extract packets
142     if (!m_tcp_data_reader->getPackets(data, tcp_packets))
143         return;
144
145     // feed Synchronization Manager
146     if (tcp_packets.isEmpty())
147         AVLogger->Write(LOG_WARNING, "Recorder::slotTCPGotData: no packets "
148                         "for feeding the Synchronization Manager received!");
149     else
150     {
151         m_check_tcp_data_timer->stop();
152         m_check_tcp_data_timer->start(TCP_DATA_TIMEOUT_MS);
153         m_synch_manager->feedAudioDataPackets(tcp_packets);
154     }
155
156     // check data write timer
157     if (!m_write_data_timer->isActive())
158         m_write_data_timer->start(m_config->data_write_timeout_ms);
159 }
```

Listing 5.3: *recorder.cpp:* Aufruf des Slots `slotTCPGotData(..)`

Zeile 133 bis 137 prüft zuerst einmal, ob das Ende eines Tages bevorsteht. Dies ist notwendig, da mit jedem neuen Tag ein neues Archiv begonnen werden muss. In Zeile 139 wird eine `QPtrList` angelegt, die dem Data Reader (Member `m_tcp_data_reader`) in Zeile 142 übergeben wird. Der Data Reader versucht nun aus den Raw-Daten die Audio Pakete zu extrahieren. Gelingt dies und ist die retournierte Liste also nicht leer, kann der `SynchronizationManager` mit diesen Paketen "gefüttert" werden (Zeile 153). In Zeile 151 bis 152 wird ein Timer neu gestartet, der auf mögliche TCP Verbindungsabbrüche überprüft.

Das Sequenzdiagramm 5.16 gibt abschließend nochmals eine Übersicht über die wichtigsten operationalen Funktionen des *AudioRec* Programms. Stellvertretend für die diversen Signale die ausgelöst werden, steht die Event Loop, welche die Slots vom `Recorder` bedient. Das Diagramm zeigt die Aufrufe der `public` Methoden beim Empfangen von TCP und UDP Daten: `slotTCPGotData()` respektive

`slotUDPGotData()`. Die Methode `slotWriteAudioData()` wird aufgerufen, wenn der Timer `m_write_data_timer` ein Timeout generiert. Dieses signalisiert, dass gepufferte Daten ins Archiv geschrieben werden sollen.

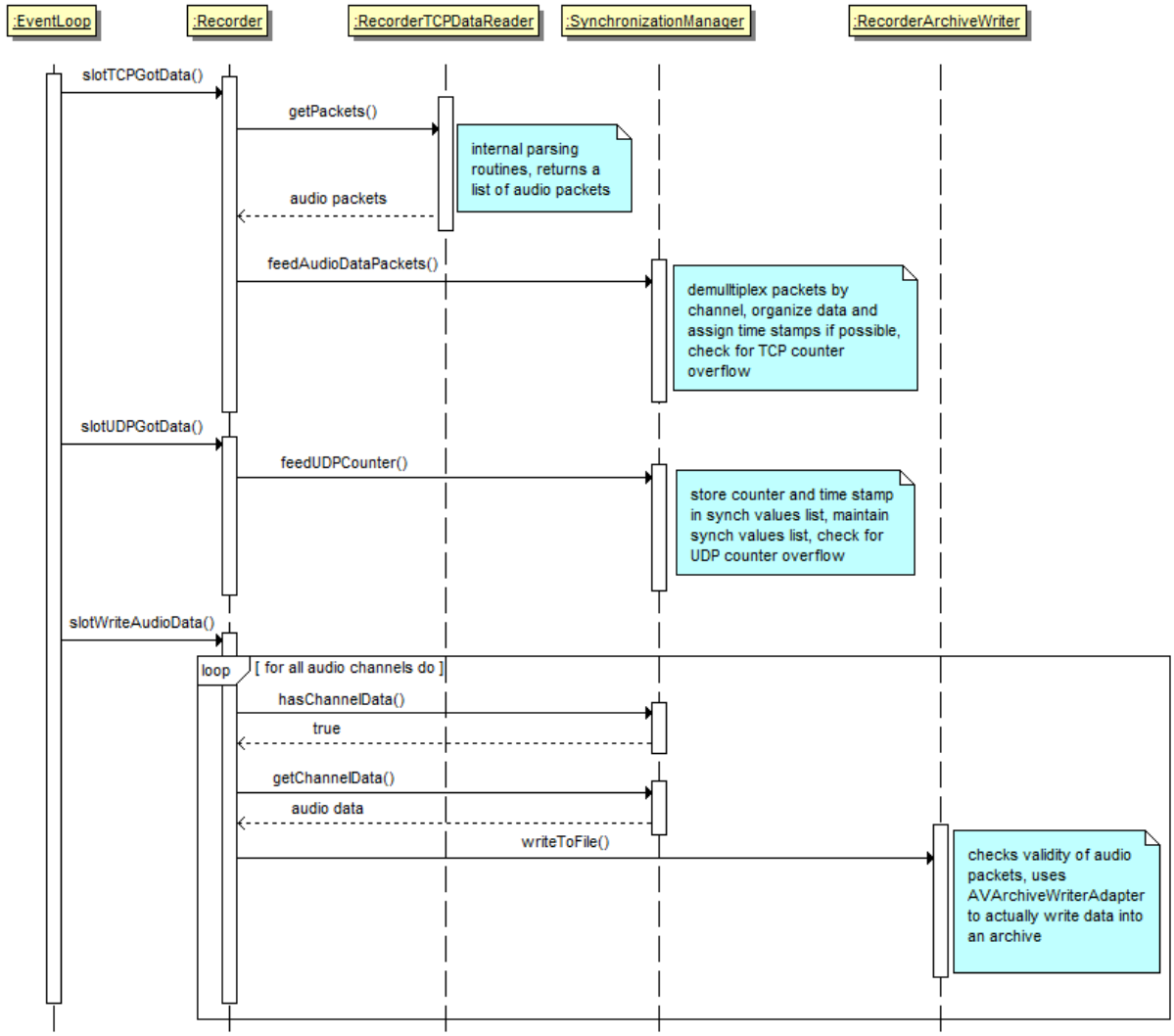

Abbildung 5.16: *AudioRec* Sequenzdiagramm für die wichtigsten operationalen Funktionen

5.2.3 Konfiguration und Debugging

Die einzige Möglichkeit *AudioRec* zu steuern ist über die Start-Up Konfiguration, welche beim Programmstart eingelesen wird. Hier werden Verbindungsparameter bestimmt und Angaben zur Archivierung gemacht. Außerdem loggt das Programm Aktionen mit und gibt diese auf der Konsole aus bzw. schreibt diese in ein separates Log File. Die Meldungen besitzen diverse Status Levels. `INFO`, `WARNING` und `ERROR` werden per default immer ausgegeben. Gleiches gilt für `FATAL`, wobei das Programm aber auch automatisch beendet wird. `DEBUG`, `DEBUG1` und `DEBUG2` sind Meldungen mit Debugging Informationen und werden im üblichen Programmablauf nicht gelogged.

5.2.3.1 Start-Up Konfiguration

Das Binary von *AudioRec* nennt sich `audiorec`. Programmparameter werden wie in Linux üblich durch einen Bindestrich gefolgt von dem Parameterkürzel eingeleitet. Dann folgt der Wert, sofern der Parameter einen benötigt. Gibt man auf der Konsole `audiorec -help` ein, so erhält man folgenden Output:

```
INFO:    AUDIOREC starting up ...........
INFO:    SIGFPE disabled

Usage: audiorec [options]

General Options:

    -cfgdir d .... use this config directory
    -cfg c ....... use this config file or server
    -defps ps .... use another default parameter set
    -save ........ save the used parameters
    -debug n ..... set the debug level to n (default=0)
    -an app ...... set application to app (not supported by all progs.)
    -help ........ this help text
    -version ..... show version information

Program specific options:

    -addr x ...... Audio server TCP/IP address to connect to
                   current value: 127.0.0.1
    -tcpp x ...... Audio server TCP/IP port, all audio data is transferred
                   over a single connection
                   current value: 8070
    -udpp x ...... Audio server UDP port, the current counter is send
                   over the UDP connection
                   current value: 8071
    -write x ..... Writes audio data every x milliseconds to disk
                   current value: 5000
    -dir x ....... The desired archive directory to place the files in or
                   an empty string for the default
    -keep x ...... The number of files to keep in the archive directory
                   current value: 5
    -pre x ....... A prefix that can be prepended to the current archive
                   file name
```

Das Programm beendet danach. Die Parameter des Abschnitts `General Options` finden sich in jedem **AviBit** Programm. Sie bieten Möglichkeiten gezielt Parameter Sets aus einer Datei im Dateisystem auszulesen, die mehrere solcher Parameter Sets beinhaltet oder eben das derzeitige Parameter Set in eine Datei zu schreiben. Außerdem kann das Debug Level angepasst oder die Programmversion abgefragt werden.

Für *AudioRec* selbst sind die programmspezifischen Parameter (Abschnitt `Program specific options`) interessant. Zusätzlich zum Hilfstext ist eine genauere Erklärung in Tabelle 5.2 nachzulesen.

Parametername	Erklärung	Default Wert
`-addr`	IP Adresse zu der das Programm verbinden soll	127.0.0.1
`-tcpp`	TCP Port Nummer (für Audiodaten)	8070
`-udpp`	UDP Port Nummer (für den UDP Counter)	8071
`-write`	Schreibzyklus in Millisekunden, gibt an wie oft Audiodaten ins Archiv geschrieben werden	5000
`-dir`	Verzeichnis für die Archivdatei	
`-keep`	Die Anzahl der aktuellen Archivdateien, die im Archivverzeichnis erhalten bleiben sollen	5
`-pre`	Ein beliebiger Präfix der vor den Archivnamen gestellt werden kann	

Tabelle 5.2: Programmspezifischen Start-Up Parameter von *AudioRec*

Der Archivname beinhaltet immer einen Datums- und Zeitstempel und besitzt die Dateiendung `.ara`. Der Name hat also die Form `YYYYMMDD-HHMMSS.ara`. Wird ein Archiv also am 1. Mai 2010 um Punkt 15:00 angelegt, lautet der Dateiname `20100501-150000.ara`.

5.2.3.2 Logging und Debugging

Debugmeldungen sind wichtig, um Rückmeldungen über Aktionen während des Programmablaufs zu geben. Das Loggen von Debugmeldungen kann mit `-debug` aktiviert werden. Feinere Log Levels sind mit `-debug1` bzw. `-debug2` möglich.

Auch mit deaktiviertem Debugging werden wichtige Informationen über Status und Zustand des Programms geliefert. Eine Analyse der Log Messages, besonders jener mit Level `WARNUNG` und `ERROR`, ist im Fehlerfall unumgänglich.

Zur Illustration ist der Konsolen-Output von *AudioRec* nachfolgend angegeben (Debugging ist deaktiviert). Das Programm verbindet sich lokal zum *Audio Hardware Simulator* (siehe 5.2.3.3, legt ein neues Archiv mit dem Namen `20100427-131659.ara` an und archiviert sieben Kanäle. Danach wird es manuell beendet.

```
INFO:    AUDIOREC starting up ..........
INFO:    SIGFPE disabled
INFO:    RecorderTCPDataTransport:setConnectTimeout: 1000ms
INFO:    RecorderTCPDataTransport:setAutoReconnect: 1
INFO:    RecorderTCPDataTransport::connect: connecting to 127.0.0.1:8070
INFO:    AUDIOREC: TCP connected
INFO:    AVArchiveAdapter: arcdir:., verbose:0, raw:1
INFO:    AVArchiveWriterAdapter: files2keep:5, correct_dt:0
INFO:    AVTimeReference:globalResyncCounter: creating SHM access
INFO:    AVShm:openSegment: got ID (0x6103c0a8) from ftok
INFO:    AVTimeReference:globalResyncCounter: got SHM with ID=5832708
INFO:    AVArchiveWriterAdapter:openArchiveFile(fn): 20100427-131659.ara
INFO:    AVArchiveReaderWriter:closeFile: removing empty archive file
(/usr/home/users/markus/20100427-131659.ara)
INFO:    AVUdpTransport::AVUdpTransport: Changed operating system receive
buffer size from 110592 bytes to 262142 bytes (wanted 512000 bytes)
INFO:    AUDIOREC start-up complete
INFO:    AVProcessStateFactory:getNewProcessState: creating
AVProcessStateOld object
INFO:    AVShm:openSegment: got ID (0x6103c0aa) from ftok
INFO:    Wrote 116000 bytes to archive for channel 1, time stamp:
2010-04-27 13:16:59.366
INFO:    Wrote 116000 bytes to archive for channel 2, time stamp:
2010-04-27 13:16:59.366
INFO:    Wrote 116000 bytes to archive for channel 3, time stamp:
2010-04-27 13:16:59.366
INFO:    Wrote 116000 bytes to archive for channel 4, time stamp:
2010-04-27 13:16:59.366
INFO:    Wrote 116000 bytes to archive for channel 5, time stamp:
2010-04-27 13:16:59.366
INFO:    Wrote 116000 bytes to archive for channel 6, time stamp:
2010-04-27 13:16:59.366
INFO:    Wrote 116000 bytes to archive for channel 7, time stamp:
2010-04-27 13:16:59.366
INFO:    AUDIOREC stopping
INFO:    shutting down AUDIOREC
INFO:    Writing remaining data to the archive
INFO:    Wrote 24000 bytes to archive for channel 1, time stamp:
2010-04-27 13:17:13.867
INFO:    Wrote 24000 bytes to archive for channel 2, time stamp:
```

```
2010-04-27 13:17:13.867
INFO:    Wrote 24000 bytes to archive for channel 3, time stamp:
2010-04-27 13:17:13.867
INFO:    Wrote 24000 bytes to archive for channel 4, time stamp:
2010-04-27 13:17:13.867
INFO:    Wrote 24000 bytes to archive for channel 5, time stamp:
2010-04-27 13:17:13.867
INFO:    Wrote 24000 bytes to archive for channel 6, time stamp:
2010-04-27 13:17:13.867
INFO:    Wrote 24000 bytes to archive for channel 7, time stamp:
2010-04-27 13:17:13.867
INFO:    AVTimeReference:cleanup
INFO:    process ended
```

5.2.3.3 Hilfsprogramme

Ein wichtiges Hilfsprogramm ist der *Audio-Hardware Simulator*, kurz *AHWSim*. Es diente als Stellvertreter, so lange die Audio Hardware XC-1124A selbst noch in Entwicklung war. Der *AHWSim* arbeitet ebenso autonom wie *AudioRec*. Bei Programmstart liest es WAVE Dateien (bis zu 24 Stück) aus einem angegeben Verzeichnis ein und wartet auf eine Verbindung. Verbindet sich nun *AudioRec* zu dem Programm beginnt der *AHWSim* die Dateien in Echtzeit zu senden. Für die Entwicklung wurde ein Hörbuch verwendet. Jede einzelne Datei stellte ein Kapitel dar. Der *AHWSim* konnte also bis zu 24 Kapitel gleichzeitig an *AudioRec* schicken (genau in der Geschwindigkeit, in der das Hörbuch abgespielt werden soll) und somit einen kontinuierlichen Strom von Audiodaten erzeugen. Das Format entsprach dabei natürlich jenem des XC-1124A Systems (siehe 5.2.1).

Ein weiteres Hilfsprogramm ist das Tool *Arch2Wave*, welches aus **AviBit** Archivdateien einzelne WAVE Dateien extrahieren kann. Die Voraussetzung ist dafür natürlich, dass G.711 komprimierte Audiodaten im Archiv enthalten sind. Das Programm kann so konfigurierte werden, dass alle oder nur ausgewählte Kanäle aus dem Archiv extrahiert werden. Die resultierenden WAVE Dateien sind im PCM Format und können mit einem beliebigen Player abgespielt werden. Dieses Programm hat den Vorteil, dass der *AVPlayer* nicht benötigt wird, um Archive zu verifizieren. Man kann einfach einen Kanal extrahieren und anhören. Da Stille nicht aufgezeichnet wird, befinden sich Lücken zwischen den archivierten Audiodaten. *Arch2Wave* kann so konfiguriert werden, dass die Lücken mit einer fixen Zeit beim Extrahieren aufgefüllt werden, mit der tatsächlichen Zeit der Pause oder auch gar nicht aufgefüllt werden.

Der nächste Abschnitt behandelt weiters den Media Player, der bislang zum Abspielen von Videodaten verwendet wurde und nun auf Audiodaten erweitert werden musste. Die große Problemstellung dabei war, diese beiden Medien synchron wiederzugeben.

5.3 Erweiterung des Videoplayers: AVPlayer

Der *AVPlayer* besitzt eine grafische Oberfläche und gibt ausgewählte Video- und Audioarchive synchron wider. Der Benutzer kann zu einer beliebigen Zeit springen und die Wiedergabe jederzeit starten und stoppen. Bevor der *AVPlayer* ausführlich behandelt wird, soll aber zuerst geklärt werden, wie eine Audio Archivdatei aufgebaut ist und welche Eigenschaften sie besitzt.

5.3.1 Aufbau einer Archivdatei

Eine **AviBit** Archivdatei ist ein Containerformat zur Speicherung beliebige Daten. In vielen Fällen sind das binäre Streams die sequentiell gespeichert werden können. Das Archiv kann mit verschiedensten Klassen der `AVArch` Bibliothek geschrieben und gelesen werden. *AudioRec* verwendet die Klasse `AVArchiveWriterAdapter`, der *AVPlayer* die Klasse `AVArchiveReaderAdapter`. Geschrieben und gelesen werden einzelne `AVMsg` Objekte. Diese Objekte kann man mit einer Message Identifikation versehen, eine Kanalnummer zuweisen (0 - 255 Kanäle sind möglich), ein Zeitstempel kann hinzugefügt oder automatisch generiert werden und natürlich hält das Objekt auch die binären Daten an

sich. Falls notwendig lässt sich eine `AVMsg` auch komprimieren. Die Objekte selbst sind auch serialisierbar und können somit übers Netzwerk versendet werden.

Im Gegensatz zu Programmen die kontinuierliche Daten archivieren, besitzen die von *AudioRec* geschriebenen Archivdateien aber einige Eigenheiten. *AudioRec* schreibt bis zu 24 Kanäle gleichzeitig. Das bedeutet, für einen Zeit- und Datumsstempel, der auf die Millisekunde genau ist, können bis zu 24 gleiche Nachrichten in demselben Archiv enthalten sein. Diese sind beginnend mit der niedrigsten Kanalnummer nacheinander angeordnet. Innerhalb der Audiodaten können jedoch auch Pausen (Stille) vorkommen, die eben nicht geschrieben werden. Das heißt, von Zeitstempel t_1 bis zu Zeitstempel t_2 des selben Kanals ist nicht garantiert, dass es vollständige Audiodaten gibt. Die Nachricht kann ohne weiteres früher aufhören. Sie ist jedoch niemals länger. Dies führt letztendlich zu einer Fragmentierung des Archivs, welche in Abbildung 5.17 illustriert ist. Das Beispiel zeigt eine stark fragmentierte Archivdatei, welche zur besseren Illustration auf einer Zeitachse aufgetragen ist. Ein *Msg* Block repräsentiert einen Block an - zusammenhängenden - Audiodaten. Die Zahl dahinter zeigt die Reihenfolge an, in der die Blöcke geschrieben wurden.

Das Problem bei der Wiedergabe ist, dass diese an einer beliebigen Stelle im Audioarchiv starten können muss. Der `AVArchiveReaderAdapter` unterstützt aber lediglich den Sprung zum Anfang einer archivierten `AVMsg` und selbst dann muss noch geprüft werden, von welcher Kanalnummer diese stammt. Wie dieses Problem gelöst wurde ist unter anderem in Unterabschnitt 5.3.3 erklärt.

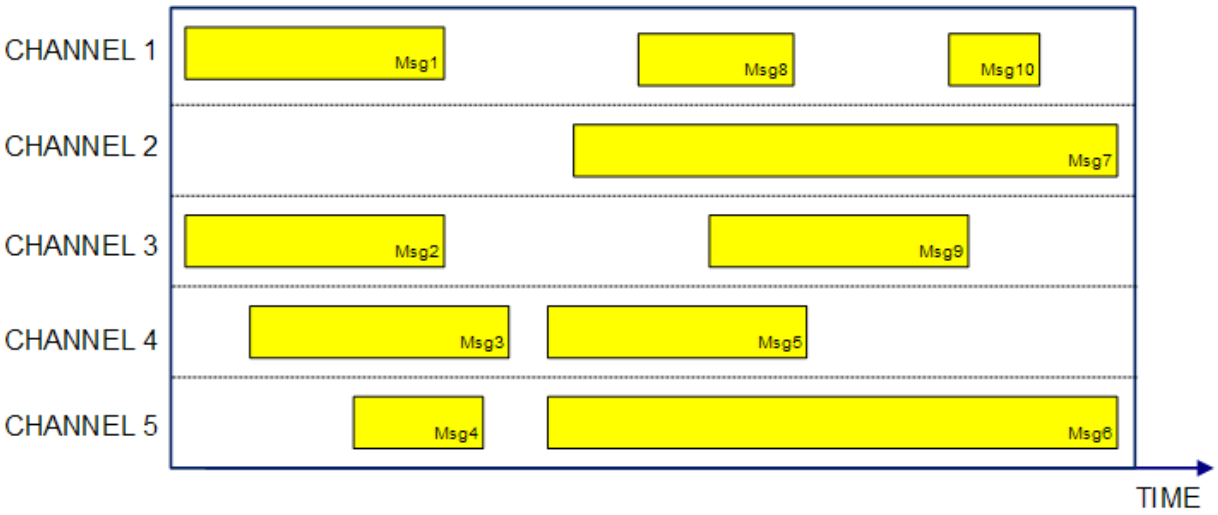

Abbildung 5.17: Beispiel der Fragmentierung einer Archivdatei

5.3.2 Grafische Benutzeroberfläche

Die ursprüngliche Oberfläche des Programms wurde bereits in den Abbildungen 5.2 und 5.3 gezeigt. Diese musste aber auch erweitert werden, wobei das alte Aussehen größtenteils erhalten blieb. Abbildung 5.18 zeigt die neue Oberfläche. Vor allem die Buttonleiste links oben wurde erweitert. Die zwei linkesten Knöpfe - Beschriftung 1 und 2 - dienen dem Öffnen von Archivdateien. Video- und Audioarchive werden separat geöffnet. Die anderen Buttons der Leiste, markiert mit 3, sind die Player Elemente und zeigen deren gewohnte Piktogramme. Von links nach rechts sind deren Funktionen: Zurückspulen bis zum Anfang, Sprung zurück, Stopp (Abspielposition bleibt erhalten), Play (Wiedergabe startet von der aktuellen Position), Sprung vorwärts. Die Sprünge vorwärts und rückwärts sind konfigurierbar (siehe 5.3.4). Bei Beschriftung 4 befindet sich der Slider zur Änderung der Wiedergabegeschwindigkeit. Dies ist aber nur beim Video anwendbar. Eine gegebenfalls aktivierte Soundausgabe würde verstummen, so lange sich dieser Slider ungleich der Position 1 befindet, was der normalen Geschwindigkeit entspricht. Die mit *Video Information* bzw. *Audio Information* beschrifteten Fläche zeigen Informationen über die Medien an. Dazu gehören der Dateiname, das Aufnahmedatum und der zeitliche Umfang (von-bis). Rechts davon findet sich die einzelne Knöpfe zum Aktivieren der Audiokanäle (*Audio Channels*). Je-

der Kanal kann separat aktiviert und deaktiviert werden. Der *AVPlayer* spielt bis zu fünf Audiokanäle gleichzeitig ab. Diese werden der Abspielsituation entsprechend vom *AVPlayer* zusammen gemixt. Da von vielen Sprechpausen in den einzelnen Kanälen auszugehen ist, fällt die Überlagerung von mehreren gleichzeitig gesprochenen Stimmen nicht ins Gewicht. Bei Beschriftung 5 ist der Slider für die aktuelle Abspielposition zu sehen. Dieser kann mit der Maus auf der Achse nach links und rechts verschoben werden. Die Zeit (diese bezieht sich aufs Archiv) wird links daneben angezeigt.

Abbildung 5.19 zeigt den *AVPlayer* mit einem geöffneten Video- und Audioarchiv. Es sind vier Audiokanäle aktiviert. Die Informationen über die Archive werden in den jeweiligen Boxen angezeigt (*Video Information* bzw. *Audio Information* unterhalb der Buttonleiste). Das Video besteht lediglich aus einem aufgenommenen Konsolenfenster, welches eine Uhr (rechts oben) zeigt. Somit kann visuell überprüft werden, ob die angezeigte Zeit des *AVPlayers* mit der archivierten Zeit des Videos übereinstimmt.

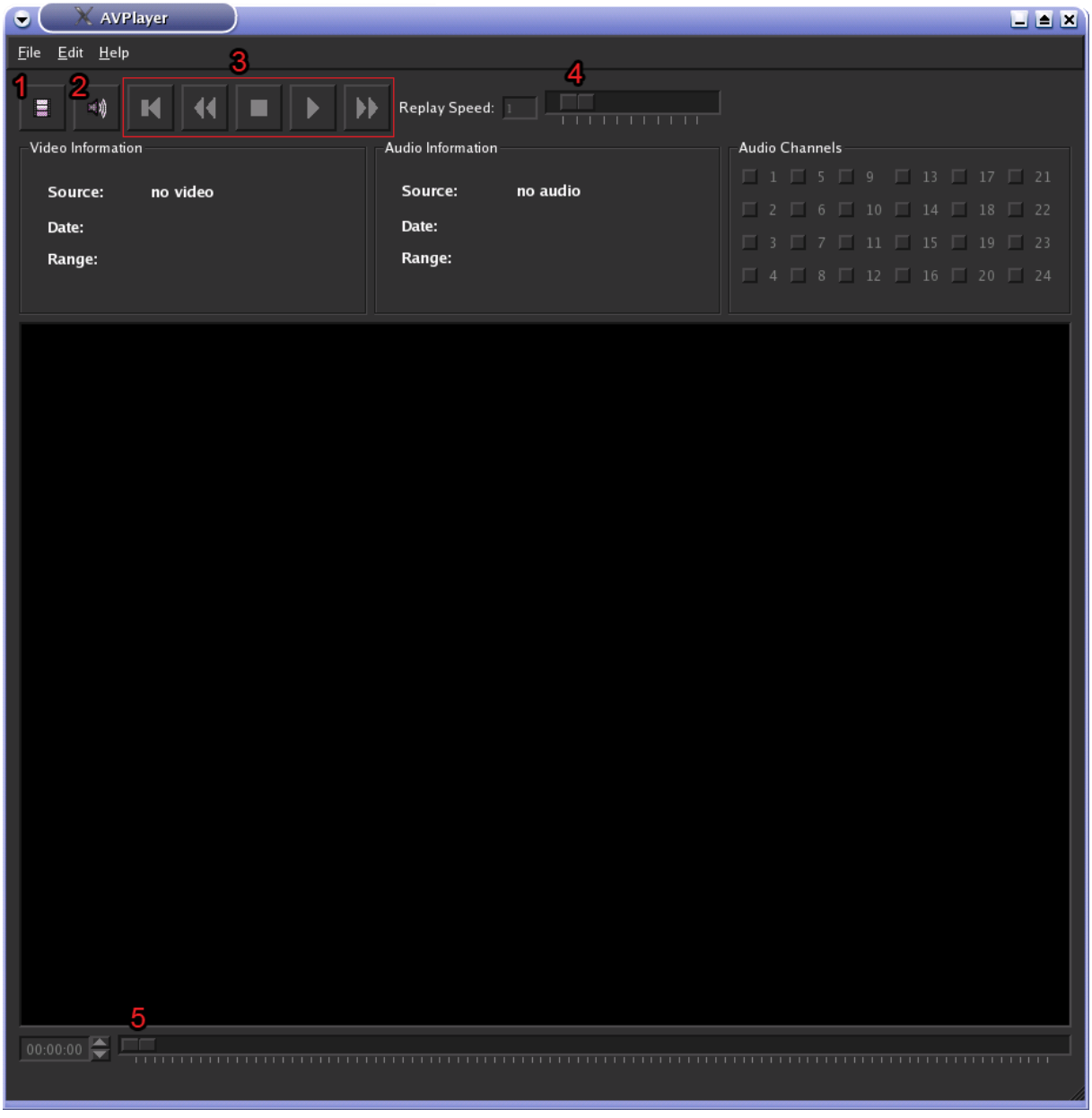

Abbildung 5.18: Screenshot der erweiterten Oberfläche des *AVPlayers*

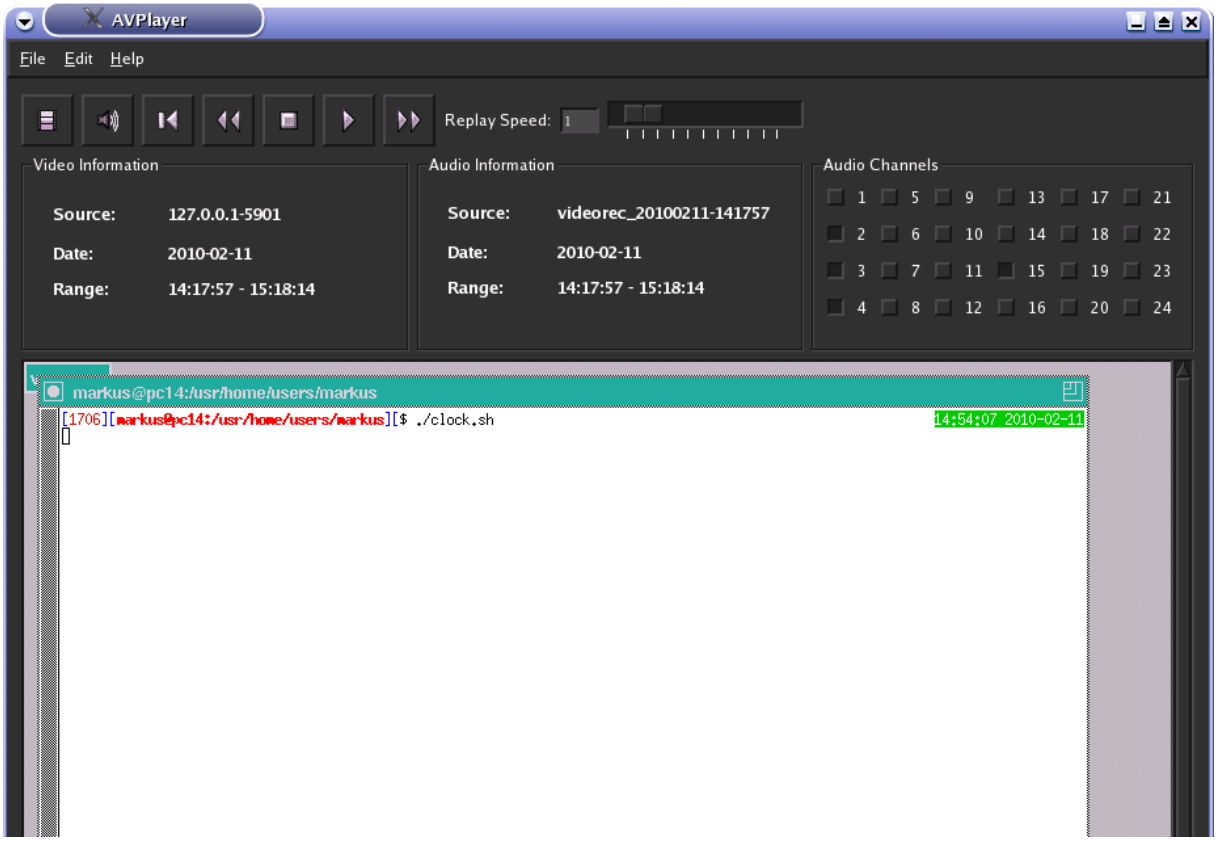

Abbildung 5.19: Ein Screenshot des *AVPlayers* mit einem geöffneten Video- und Audioarchiv

Der *AVPlayer* lässt sich auch über Hotkeys steuern. Eine Übersicht über die verwendeten Tastaturkürzel ist in Tabelle 5.3 zu sehen.

Hotkey	Funktion
Strg + V	Videoarchiv öffnen
Strg + Shift + V	Videoarchiv schließen
Strg + A	Audioarchiv öffnen
Strg + Shift + A	Audioarchiv schließen
Strg + S	Erstellt einen Screenshot des angezeigten Videos
Strg + Q	Beenden des Programms
Leertaste	Play (Wiedergabe starten)
Esc	Stopp (Wiedergabeposition bleibt erhalten)
Pfeil-nach-links	Sprung zurück
Pfeil-nach-rechts	Sprung vorwärts
Pos1	Setzt die Wiedergabeposition auf den Anfang

Tabelle 5.3: Eine Übersicht über die vom *AVPlayer* verwendeten Hotkeys (Tastaturkürzel)

Bei der Entwicklung wurde davon ausgegangen, dass das Videomaterial ausschlaggebend für die Zeitgebung sein soll. Werden nun also Medien geladen, die sich zeitlich nicht exakt abdecken, so zählt immer der Zeitbereich des Videoarchivs als maßgebend. Ist das Audio Recording länger als das Video Recording, wird maximal der Zeitbereich des Videos wiedergeben. Ist das Audio Recording kürzer als das Video Recording, wird die Soundausgabe an nicht vorhanden Stellen mit Stille gefüllt. Überlappen sich die beiden Recordings überhaupt nicht, so erhält der Benutzer eine Warnung, dass keine Soundausgabe möglich ist. Es kann nur Videomaterial angezeigt werden.

Diese drei Situation sind in Abbildung 5.20 illustrierte. Im ersten Bild ist das Audio Recording länger als das Video Recording, in zweiten ist das Audio Recording kürzer und im dritten Bild überlappen sich Audio und Video Recording nicht.

Video- und Audioarchive lassen sich unabhängig voneinander wiedergeben. Es ist also möglich, nur ein Audioarchiv zu laden, ohne dass dieses von einem Videoarchiv zeitlich beschränkt wird. In diesem Fall ist dann nur das Audio Replay aktiv.

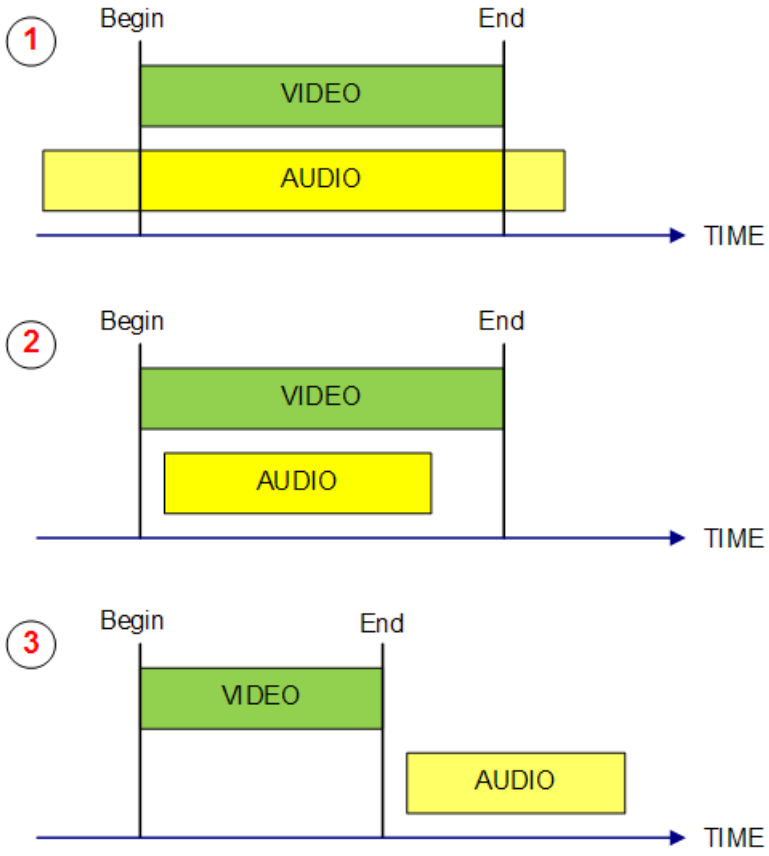

Abbildung 5.20: Drei möglichen Situation bei der zeitlichen Überlappung von Video und Audiomaterial

5.3.3 Architektur und Funktionsweise

Gerade bei grafischen Benutzeroberflächen spielt der Signal/Slot Mechanismus eine wichtige Rolle. Jedes Mal, wenn eine interaktive Schaltfläche betätigt wird, löst diese ein Signal aus. Sofern ein Slot mit dem Signal verbunden ist, kann dieser reagieren und die notwendigen Aktionen ausführen. Ein Beispiel dafür ist der Klick auf den Play Button. Der Button ist vom Typ `QPushButton` [Tro05, QPushButton]. Beim Drücken löst dieser das Signal `clicked()` aus. Dieses ist mit dem Slot `slotStartPlaying()` verbunden. Dieser Slot ist eine `protected slot` Methode der Klasse `ReplayerWidget` und führt nun alle notwendigen Aktionen durch, um die geladenen Medien ab dem gewünschten Zeitpunkt abzuspielen.

Als der Code für die Erweiterung übernommen wurde, konnte natürlich nur Videomaterial abgespielt werden. Das Klassendiagramm 5.21 zeigt jene Klassen, die dafür zuständig sind. Aus Platzgründen zeigen die Klassen nur jene Methoden und Member an, dir für die Video Wiedergabe relevant sind.

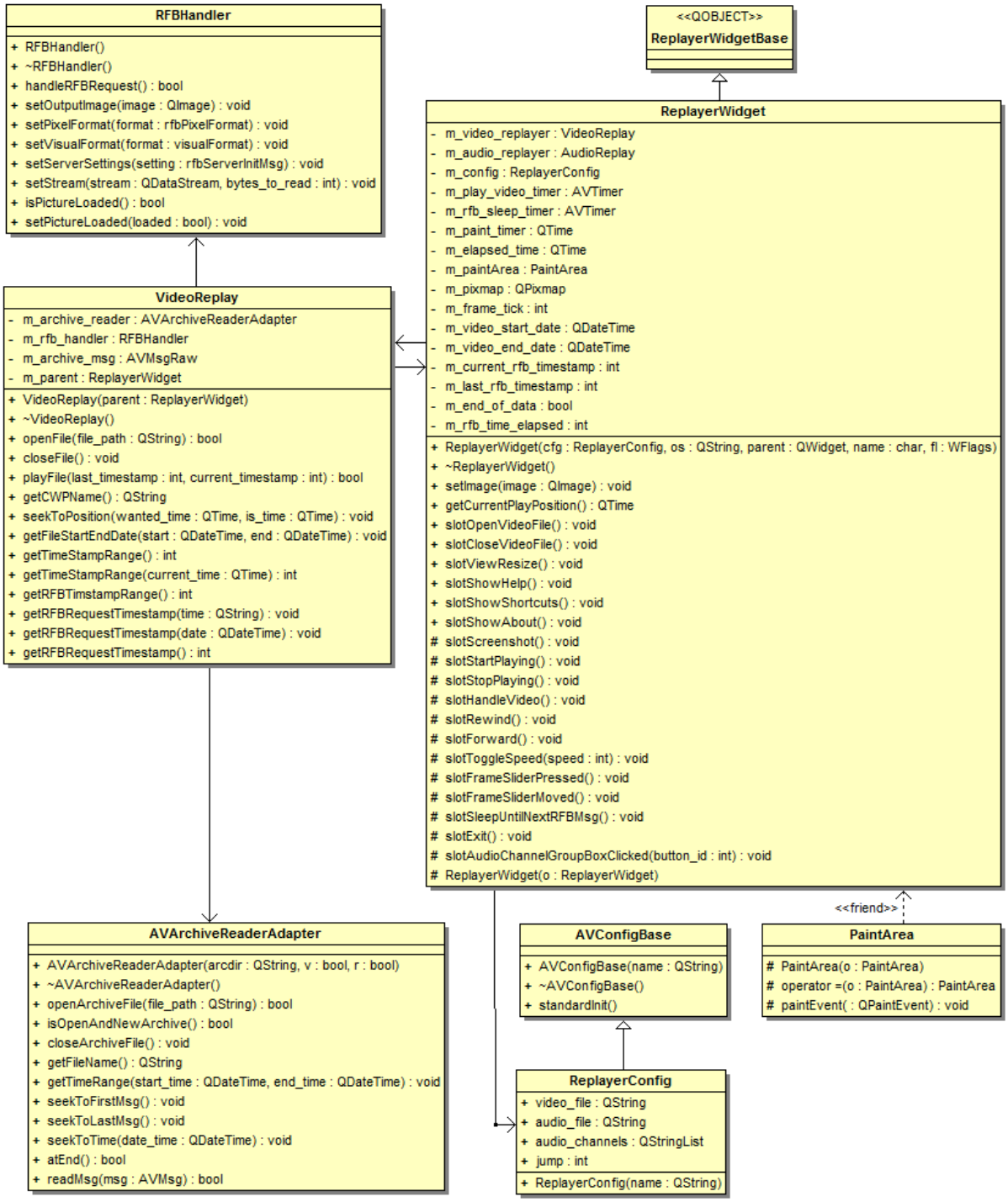

Abbildung 5.21: *AVPlayer* Klassendiagramm: Video Replay

`ReplayerWidgetBase` ist eine vom *Qt Designer* automatisch generierte Klasse. Der *Qt Designer* ist ein Tool von Qt, welches die Erstellung von grafischen Oberflächen in einem WYSIWYG[10] Editor ermöglicht. Es lassen sich Forms und Widgets erstellen und mit notwendigen Elementen, wie Buttons, Checkboxes, Radioboxes, Eingabefelder, Menüs und Beschriftungselementen versehen. Das Layout kann erstellt und Hotkeys definiert werden. Außerdem kann man schon Signale mit ihren zugehörigen Slots verbinden. Dementsprechend beinhaltet die Klasse `ReplayerWidgetBase` als Member alle GUI Elemente, die zwecks komfortablen Zugriff `public` sind. Auch die Slots sind in dieser Klasse enthalten, wurden aber erst in der Ableitung `ReplayerWidget` implementiert.

`ReplayerWidget` ist also die Verwaltungsklasse für die GUI. Sie besitzt alle Methoden, um auf die jeweiligen Aktionen vom Benutzer reagieren zu können. Das meiste davon wird jedoch gleich an die Klasse `VideoReplay` delegiert (Zugriff durch den Member `m_video_replayer`). Beispielsweise wird ein Aufruf zum Öffnen einer Videodatei zwar in `ReplayerWidget::openVideoFile(..)` registriert, das eigentliche Öffnen findet aber in `VideoReplay::openFile(..)` statt. Der `ReplayerWidget` ist nun lediglich verantwortlich die richtigen Daten zu setzen. Er schreibt die Archivinformationen im Feld *Video Information* und initialisiert den zeitlichen Anfang (siehe Abbildung 5.19), aktiviert die Bedienelemente und das Anzeigefeld und setzt einige interne Variablen - beispielsweise, dass eine Videodatei erfolgreich geladen wurde.

Beim Abspielen von Videomaterial reagiert `ReplayerWidget` zeitgesteuert auf die Timer `m_play_video_timer` und `m_rfb_sleep_timer`. Erster ruft nach einem Timeout von einer Millisekunde den Slot `slotHandleVideo()` auf. Dieser Slot benutzt das `VideoReplay`, um ein Einzelbild des Videos darzustellen - `VideoReplay` muss die notwendigen Daten aus dem Archiv auslesen. Danach wird der `m_play_video_timer` deaktiviert und der `m_rfb_sleep_timer` aktiviert. Es wird nämlich nun gewartet, bis der Zeitstempel für das nächste Bild erreicht wird. Dann kann wiederum der `m_play_video_timer` aktiviert und der `m_rfb_sleep_timer` deaktiviert werden, um das nächste Bild darzustellen. Mit dieser Methode erhält man je nach gespeicherter Datenmenge lediglich fünf bis zehn Frames per Second (FPS). Das ist zu wenig, um Filme anzusehen, aber es reicht, um Bildschirminhalte von Anwendungen darzustellen.

Wesentlich wichtiger für die vorliegende Arbeit ist jedoch die Funktionalität zur Audio Wiedergabe. Die dafür notwendigen Klassen sind in Abbildung 5.22 zu sehen. Auch hier werden nur Audio relevante Methoden und Member dargestellt.

[10] WYSIWYG: What You See Is What You Get, eine Metapher zum Bearbeiten und Anzeigen von Elementen, Strukturen und Design, wie sie auch in der Ausgabe dargestellt werden.

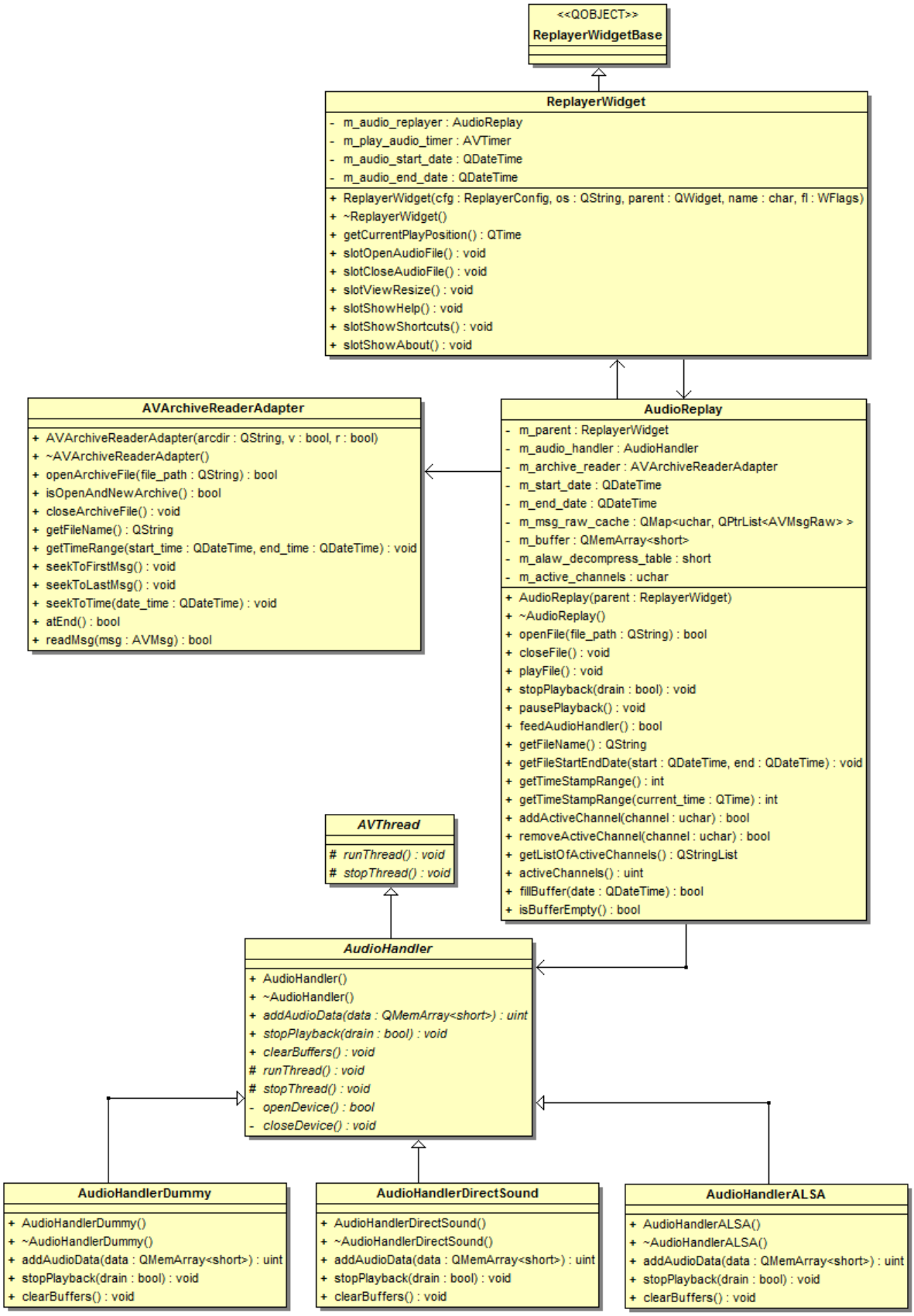

Abbildung 5.22: *AVPlayer* Klassendiagramm: Audio Replay

Ein Wunsch von **AviBit** war es, die Wiedergabe unter Linux und Windows zu ermöglichen. Zwar gibt es hierfür betriebssystemunabhängige Sound Libraries, doch der Implementationsaufwand, um lediglich einen Audiostream abzuspielen, wäre zu hoch gewesen. Die dritte Major Release von Qt bietet hier leider auch keine Möglichkeiten an. Die Lösung bestand nun darin eine abstrakte Basisklasse `AudioHandler` zu erstellen während die konkreten Ableitungen die betriebssystemspezifischen Funktionen zur Audio Wiedergabe implementieren. Im Falle von Linux wird ALSA verwendet (Klasse `AudioHandlerALSA`), unter Windows DirectSound (Klasse `AudioHandlerDirectSound`) (siehe 5.1.4 für Versionsinformationen). Für ein unbekanntes Betriebssystem wurde eine Dummy Implementation, `AudioHandlerDummy`, erstellt, die ohne Funktionalität ist. Der Benutzer erhält in diesem Fall vom *AVPlayer* eine Warnung, dass eine Wiedergabe von Sound nicht möglich ist.

Wie in Abbildung 5.22 ersichtlich ist, ist `AudioHandler` von `AVThread` abgeleitet. Das heißt, auch die konkreten Implementationen sind Threads. Ein Audio Handler sollte nämlich unabhängig vom *AVPlayer* arbeiten können. Die Audio Handler sind dafür zuständig die Soundkarte kontinuierlich mit Daten zu versorgen. Dabei gibt es bei ALSA und DirectSound verschiedene Ansätze.

ALSA wartet auf eine Rückmeldung der Soundkarte, wenn diese neue Daten für den Buffer benötigt. Die Soundkarte löst also einen Interrupt aus, welcher von ALSA registriert wird. Diese Rückmeldung wird an den Audio Handler weitergegeben, der den Buffer mit neuen Daten befüllen kann.

Im Gegensatz dazu muss bei DirectSound selber überprüft werden, wann der Buffer neu mit Daten zu befüllen ist. DirectSound benutzt einen Primary und einen Secondary Buffer. Erster ist für das Mixen und diverse Soundeffekte verantwortlich, was aber im *AVPlayer* nicht benötigt wird (der *AVPlayer* mixt die Audiodaten selbst). Der Secondary Buffer hält die Audiodaten in einem Ring-Buffer. Während der Wiedergabe eruiert der `AudioHandlerDirectSound` die Play Position im Buffer und befüllt je nach Position die erste oder die zweite Hälfte mit neuen Daten.

Der Audio Handler wird von `AudioReplay` bedient. Ein Audio Handler arbeitet zwar autonom, braucht jedoch kontinuierlich Nachschub mit neuen Audiodaten. `AudioReplay` besitzt einerseits einen Cache mit `AVMsg` Objekten (`m_msg_raw_cache`), welche bereits aus der Archivdatei ausgelesen wurden und andererseits einen Software Buffer (`m_buffer`), welcher die fertig gemixten PCM Audiodaten speichert. Dieser Buffer wird regelmäßig dem Audio Handler zur Verfügung gestellt, damit dieser neue Audiodaten erhält. `AVMsg` Objekte werden während des Betriebs kontinuierlich aus der Archivdatei ausgelesen. Der Message Cache ist der Abspielposition ca. 30 Sekunden voraus, so dass auf jeden Fall genügend Daten zum Buffern bereitstehen. Die Festplattenzugriffe auf die Archivdatei müssen zeitlich beschränkt werden, da sonst das Programm blockieren würde. Für einen Zugriff stehen maximal 500 Millisekunden zur Verfügung. Während dieser Zeit wird maximal eine `AVMsg` eines aktiven Kanals ausgelesen und gecached.

Der operationelle Ablauf vom Programmstart bis zur Wiedergabe lässt sich bezogen auf die Klasse `AudioReplay` in folgende vier Schritte einteilen:

1. **Archivdatei öffnen:** Mit Hilfe der Klasse `AVArchiveReaderAdapter` wird eine Archivdatei geöffnet und deren Start- und Endzeitpunkt eruiert.

2. **Audiokanäle aktivieren:** Zumindest ein Kanal muss aktiviert werden, um eine Soundausgabe zu erzielen. `AudioReplay` kümmert sich dann während des Programmablaufs darum, dass für diesen Kanal Daten aus dem Archiv in den Programm Cache nachgeladen werden und, dass der Kanal im Mix enthalten ist.

3. **Audiodaten buffern:** Mit Hilfe des `AVMsg` Caches werden die Audiodaten ins PCM Format konvertiert, gemixt und im Buffer abgelegt. PCM Daten lassen sich leicht mixen: Man braucht ihre Werte nur zu addieren. Um nicht zu übersteuern müssen diese jedoch abgeschwächt werden. Werden also drei Kanäle gleichzeitig gespielt, so hat jeder Kanal eine maximale Lautstärke von einem Drittel der Gesamtlautstärke. `AudioReplay` überprüft kontinuierlich, ob genug Daten im Cache und im Buffer sind und leitet dementsprechende Aktionen ein.

4. **Audio Handler Input:** Der `AudioHandler` erhält regelmäßig Zugriff auf den PCM Buffer, um sich neue Daten für die Soundkarte zu holen.

In Abbildung 5.23 ist ein Sequenzdiagramm zu sehen, welches die oben beschriebenen Operationen nochmal verdeutlicht. Die involvierten Klassen sind `ReplayerWidget`, `AudioHandler`, `AudioHandler` und `AVArchiveReaderAdapter`. Die Aktionen werden vom Benutzer ausgelöst und von der Qt Event Loop gesteuert, die hier symbolisch als die erste Instanz steht. Im ersten Schritt wird das Audioarchiv geöffnet. Dann wählt der Benutzer einen Kanal aus, was in der Methode `ReplayerWidget::slotChannelGroupBoxClicked()` registriert wird. Es wird dazu auf jeden Fall das Replay gestoppt, da der Mix beim Aktivieren und Deaktivieren eines Kanals ein anderer ist. Die Methode `ReplayerWidget::slotStartPlaying()` lässt von `AudioReplay` den Mix erstellen und den Buffer befüllen. Zuerst müssen `AVMsg` Objekte gecached werden, die aus dem Archivfile ausgelesen werden. Dann können die Audiodaten gemixt werden. Ist dies gelungen, werden dem `AudioHandler` Audiodaten überreicht, so dass dieser so schnell als möglich seine Arbeit aufnehmen kann. Danach werden im `ReplayerWidget` alle notwendigen Timer gestartet, die das Replay am Laufen halten. Für die Audio Wiedergabe ist hier lediglich der `AVTimer` `m_play_audio_timer` notwendig. Dieser ruft kontinuierlich `ReplayerWidget::slotHandleAudio()` auf. Dieser Slot ist dafür verantwortlich den Cache zu prüfen und Nachrichten aus dem Archiv nachzuladen, das PCM Buffer-Level zu kontrollieren, sowie dem `AudioHandler` frische, fertig gemixte PCM Audiodaten zur Verfügung zu stellen. Damit diese Routine nicht blockiert, ist das Nachladen von Nachrichten aus dem Archiv zeitlich beschränkt.

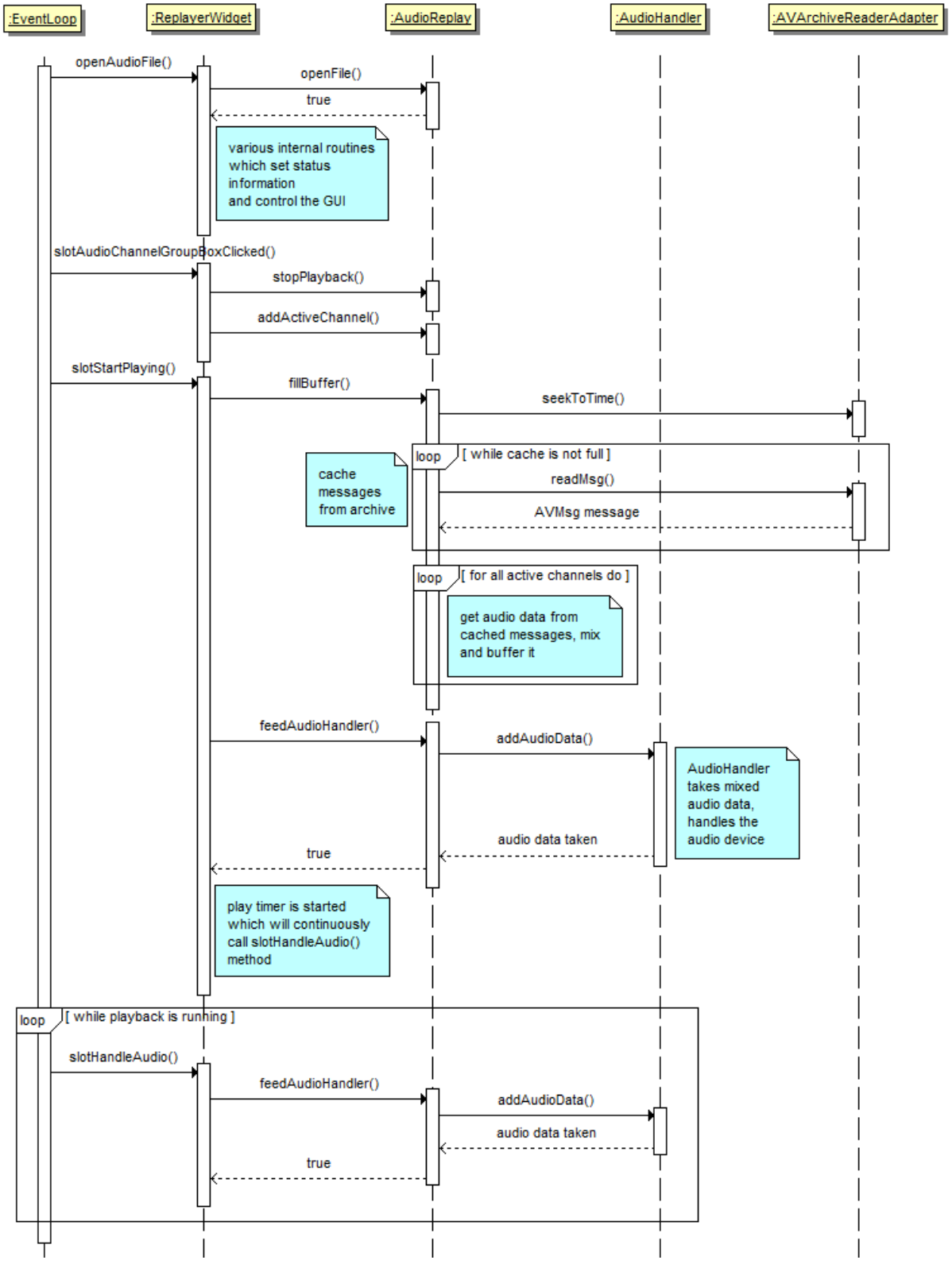

Abbildung 5.23: *AVPlayer* Sequenzdiagramm

Da es dem Benutzer frei gestellt ist, die Wiedergabe ab einer beliebigen Stelle, welche zumindest vom Videomaterial abgedeckt ist, zu starten, stellt sich das Problem, dass der Startzeitpunkt auch Mitten im Datenbereich einer `AVMsg` anfangen kann (vergleiche dazu auch Abbildung 5.17). Dementsprechend wird der `AVMsg` Cache der Audio Nachrichten einfach ca. zehn Sekunden vor der gewünschten Startzeit aufgebaut. Die maximale Länge einer archivierten Audio Message ist von *AudioRec* auf fünf Sekunden beschränkt. Der Cache beginnt also in etwa zehn Sekunden in der Vergangenheit (es hängt letztendlich von der Klasse `AVArchiveReaderAdapter` ab, welche Nachricht ausgelesen wird) und endet 30 Sekunden in der Zukunft. Beim Buffern werden die Audiodaten ihren Zeitstempeln entsprechend zusammen gemixt.

Die Synchronisation des Datenmaterials wird auch über die Zeitstempel in den `AVMsg` Objekten erreicht. Die Daten können so platziert und ausgegeben werden, dass sie auf die Millisekunde genau angezeigt bzw. abgespielt werden. Durch die laufenden Threads des Video und Audio Replays wird eine möglichst hohe Unabhängigkeit von der restlichen Applikation erreicht. So lange diese Threads also mit genug Daten versorgt werden, ist die Synchronisation gewährleistet. Geht das Datenmaterial aus, gibt es zwei Gründe: Ein interner Fehler wurde registriert und das Replay muss stoppen. Oder es wurde das Ende eines Archivs erreicht. Am Ende eines Audioarchivs wird für das restliche Video nur noch Stille ausgegeben. Am Ende des Videoarchivs hält die Wiedergabe an.

Der Code der `main(..)` Funktion zum Starten von *AVPlayer* ist in Listing 5.4 zu sehen. Für das Setup der GUI ist etwas mehr Code notwendig, als für ein reines Konsolenprogramm. In Zeile 136 wird das Betriebssystem überprüft, auf dem die Applikation gestartet wurde. Je nach Betriebssystem muss nämlich die zu Grunde liegende Sound Library verwendet werden. Zeile 140 prüft die Konfigurationsparameter auf ihre Gültigkeit. In den Zeilen 148 bis 151 wird das `ReplayWidget` Objekt konstruiert, der `AVApplication` zugewiesen und schließlich angezeigt. Zeile 154 startet schlussendlich die Event Loop, die für den weiteren Ablauf sorgt. Wird die Event Loop beendet - bspw. wenn im Programm Menü *Quit* aufgerufen wird - so wird die `main(..)` Funktion ab Zeile 157 weiter ausgeführt und terminiert das Programm gänzlich.

```
123 int main(int argc, char **argv)
124 {
125     // create the app
126     AVApplication app(argc, argv, -1);
127     app.setAllowRMBWindowDragging(false);
128     BUILDVERSION();
129     AVConfig::setApplicationName(AVConfig::APP_VVRMAX);
130     AVDaemonInit(UseEventLoop, PROCNAME, "", true);
131
132     // create the logger
133     AVLogger = new (LOG_HERE) AVLog(PROCNAME, true);
134
135     // setup config
136     QString os = checkOS();
137     const ReplayerConfig *cfg = new ReplayerConfig();
138     AVASSERT(cfg);
139
140     if (!checkConsoleParameters(*cfg, os))
141         AVLogger->Write(LOG_FATAL, "Could not start %s", PROCNAME);
142
143     // use our own style here
144     QStyle *style = new (LOG_HERE) AVCWPStyle();
145     QApplication::setStyle(style);
146
147     // create the main widget
148     ReplayerWidget *widget = new ReplayerWidget(cfg, os);
149     AVASSERT(widget);
150     app.setMainWidget(widget);
151     widget->show();
152
153     // run the app loop
```

```
154     int ret = qApp->exec();
155
156     // we're terminated here
157     AVLogger->Write(LOG_INFO, PROCNAME" terminating ...");
158     AVDEL(widget);
159     AVDEL(AVLogger);
160     AVDEL(cfg);
161
162     return ret;
163 }
```

Listing 5.4: *replayer_main.cpp:* `main(..)` Funktion von *AVPlayer*

5.3.4 Konfiguration und Debugging

Auch der *AVPlayer* bietet eine Start-Up Konfiguration an. Diese fällt aber etwas geringer aus, als jene von *AudioRec*, da sich der *AVPlayer* ohnehin über die grafische Oberfläche steuern lässt. Für wiederholende Tätigkeiten kann man aber beispielsweise Archivdateien angeben, die automatisch geladen werden sollen, oder Kanalnummer, die automatisch aktiviert werden sollen. Beim Logging gelten dieselben Levels wie bei *AudioRec*, siehe diesbezüglich 5.2.3.

5.3.4.1 Start-Up Konfiguration

Das Binary vom *AVPlayer* nennt sich `avplayer`. Programmparameter werden auch hier mit einem Bindestrich eingeleitet. Auf diesen folgt der Parametername und dessen Wert, sofern erforderlich. Gibt man auf der Konsole `avplayer -help` ein, so erhält man folgenden Output:

```
INFO:    AVPLAYER starting up ...........
INFO:    SIGFPE disabled
INFO:    Detected operating system LINUX

Usage: avplayer [options]

General Options:

    -cfgdir d .... use this config directory
    -cfg c ....... use this config file or server
    -defps ps .... use another default parameter set
    -save ........ save the used parameters
    -debug n ..... set the debug level to n (default=0)
    -an app ...... set application to app (not supported by all progs.)
    -help ........ this help text
    -version ..... show version information
    -widgetcount . print debug messages about leftover widgets
    -ssenabled n . enable build-in screenshot function
    -ssfullscreen n ... screenshots fullscreen or mainwindow

X11 Options (not all work for all applications):

    -display d ... use the specified display
    -geometry g .. use the specified geometry
    -fn f ........ use the specified font
    -bg c ........ use the specified background color
    -fg c ........ use the specified foreground color
    -btn c ....... use the specified button color
    -name n ...... set the application name
    -title t ..... set the application title (caption)
    -visual v .... use the specified visual
```

```
    -ncols n ..... limit # of allocated colors for 8-bit visuals
    -cmap ........ use a private colormap on 8-bit visuals

Program specific options:

    -v x ......... Video file which shall be automatically opened at
                   start-up; leave empty to disable auto start
    -a x ......... Audio file which shall be automatically opened at
                   start-up; leave empty to disable auto start
    -c x ......... A list of channel numbers which shall be automatically
                   set to active at start-up
                   current value: 0

    -j x ......... Seconds for a forward/backward jump in time when
                   clicking the forward/backward buttons
                   current value: 10
```

Da der *AVPlayer* eine grafische Oberfläche besitzt, bietet er drei zusätzliche Parameter bei den `General Options` an und einen eigenen Abschnitt von Parametern für eine X Window Session (`X11 Options`). Eine genaue Beschreibung dieser Parameter entfällt jedoch in dieser Arbeit. Die programmspezifischen Parameter sind in der nachfolgenden Tabelle aufgelistet.

Parametername	Erklärung	Default Wert
`-v`	Pfad + Dateiname eines Videoarchivs, welches automatisch bei Programmstart geöffnet werden soll	
`-a`	Pfad + Dateiname eines Audioarchivs, welches automatisch bei Programmstart geöffnet werden soll	
`-c`	Eine Liste von Kanalnummer, welche automatisch aktiviert werden sollen, muss in Anführungszeichen gesetzt werden. Bsp.: ”1,2,5”	
`-j`	Eine Zeitangabe in Sekunden für den Vorwärts/Rückwärts Sprung	10

Tabelle 5.4: Programmspezifischen Start-Up Parameter des *AVPlayers*

Die Parameter des automatischen Öffnens von Archiven und des automatischen Aktivierens von Kanälen sind per default leer. Das bedeutet, dass auch keine Programmstartautomatik mit diesen Einstellungen durchgeführt wird.

5.3.4.2 Logging und Debugging

Auch der *AVPlayer* liefert viele Informationen während des Programmablaufs, die über den Status des Programms Bescheid geben. Diese werden entweder in das Konsolenfenster geschrieben, mit welchem der Player geöffnet wurde, oder in eine Log-Datei. Zu den Debug Levels und deren Informationen siehe 5.2.3.2.

Beispielhaft ist im folgenden der Logging Output von *AVPlayer* ohne Debugmeldungen angezeigt. Zuerst werden die Archive geöffnet und deren Start- und Endzeitpunkte automatisch eruiert. Danach wird die Wiedergabe gestartet und wieder pausiert. Der Benutzer führte drei Sprünge vorwärts durch, jeweils zu zehn Sekunden. Danach wird die Wiedergabe wieder gestartet und nach ca. acht Sekunden gestoppt. Schlussendlich wird das Programm beendet.

```
INFO:    AVPLAYER starting up ...........
INFO:    SIGFPE disabled
INFO:    Detected operating system LINUX
```

```
INFO:    AVArchiveAdapter: arcdir:/, verbose:0, raw:1
INFO:    AVTimeReference:globalResyncCounter: creating SHM access
INFO:    AVShm:openSegment: got ID (0x6103c0a8) from ftok
INFO:    AVTimeReference:globalResyncCounter: got SHM with ID=5832708
INFO:    AVArchiveAdapter: arcdir:/, verbose:0, raw:1
INFO:    AVArchiveReaderAdapter:openArchiveFile(fn):
/usr/home/users/markus/recordings/audiorec_20100211-141755.ara
INFO:    Opened audio file audiorec_20100211-141755.ara from day 2010-02-11,
start time: 14:18:04.787, end time: 15:12:49.757
INFO:    AVArchiveReaderAdapter:openArchiveFile(fn):
/usr/home/users/markus/recordings/videorec_20100211-141757
INFO:    Opened video file 127.0.0.1-5901 from day 2010-02-11,
start time: 14:17:57.566, end time: 15:18:14.580
INFO:    Replayer::slotStartPlaying(): started at time 14:17:57.632
INFO:    Replayer::slotStartPlaying(): started at time 14:18:01.631
INFO:    Audio device successfully opened
INFO:    Replayer::slotStopPlaying(): paused at 14:18:12.251
INFO:    Forward jump to 14:18:22.251
INFO:    Forward jump to 14:18:32.251
INFO:    Forward jump to 14:18:42.251
INFO:    AVArchiveFileAccess:seekToTime: 2010-02-11 14:18:39
INFO:    Replayer::slotStartPlaying(): started at time 14:18:42.251
INFO:    Audio device successfully opened
INFO:    Replayer::slotStopPlaying(): paused at 14:18:50.459
INFO:    AVPLAYER terminating ...
```

Der letzte Abschnitt des Kapitels widmet sich noch den durchgeführten Tests.

5.4 Tests

Die Entwicklung der Programme war evolutionär und zielte auf eine hohe Testbarkeit ab. Jeder Meilenstein in der Roadmap (siehe 5.1.3) wurde überprüft. Es existierte schon sehr früh eine lauffähige Version von *AudioRec*, was die praktische Durchführung von diversen Tests wesentlich vereinfachte. Mit Hilfe des Hilfsprogramms *AHWSim* (siehe 5.2.3.3), welches schon sehr früh fertig gestellt wurde, konnten die Entwicklung von *AudioRec* unterstützt werden. So musste für *AHWSim* eine Klasse geschrieben werden, welche die zu verschickenden Audiodaten laut dem Protokoll (siehe 5.2.1) codiert. Das Gegenstück dazu ist natürlich der Parser von *AudioRec* (Klasse `RecorderTCPDataReader`). Als der Simulator dann gegen die tatsächlich Hardware ausgetauscht wurde, funktionierte das Zerlegen des Datenstrom tadellos.

Die Roadmap sah auch zwei Gesamttests vor. Der erste erfolgte im Februar 2010. Zu diesem Zeitpunkt war *AudioRec* fertig gestellt und schrieb für jeden Tag ein passendes Archiv. *AudioRec* verbindete sich zu *AHWSim*, welcher die Hardware simulierte. Als Audio Input wurde ein Hörbuch verwendet, wobei jeder Kanal ein Kapitel darstellte. Damit konnten die vollen 24 Kanäle ausgenutzt werden. Der erste Gesamttest verlief wie zu erwarten nicht problemlos und musste öfters neu gestartet werden. Das Ziel war es, zumindest 48 Stunden lang Audio Recordings zu erstellen und diese dann stichprobenartig zu kontrollieren. Über die lange Laufzeit wurden einige Programmfehler gefunden und behoben, basierend auf der Analyse der erstellten Logfiles. Die schlussendlich erfolgreich aufgezeichneten Archive wurden zusammen mit Videomaterial im *AVPlayer* kontrolliert. Dieser war auch schon so weit, dass ein Audio Playback unter Linux möglich war.

Der zweite Gesamttest des finalen Systems folgte dann im Mai 2010. Statt des Simulators wurde die Hardware XC-1124A verwendet, welche Daten zu *AudioRec* verschickte. Damit man nicht 24 Geräte an das XC-1124A anschließen musste, wurde davon ausgegangen, dass sich ohnehin alle Kanäle gleich verhalten. Es wurde also nur ein Kanal angeschlossen und auf den restlichen Kanälen Rauschen aufgezeichnet. Dieser Kanal konnte kontrolliert werden. Seitens des *AudioRec* Programms gab es dann keine Änderungen mehr. Lediglich der *AVPlayer* wurde noch erweitert und verändert, unter anderem mit der Möglichkeit einer Soundausgabe unter Windows. Zusätzlich wurden einige Performance Probleme behoben. Der programmierte Simulator leistet jedoch auch in diesem Abschnitt noch gute Dienste, da er

leichter zu bedienen war, mehr aktive Kanäle verschicken konnte und gezielt konfigurierte werden konnte.

Mit diesen Tests konnte das Projekt im Rahmen der Diplomarbeit abgeschlossen werden. Das nächste Kapitel widmet sich neben der Zusammenfassung auch einem Ausblick auf verbleibende Arbeiten. Es ist davon auszugehen, dass beim Testen noch weitere Fehler auftretenden können und gewisse Designentscheidungen noch nachträglich geändert werden. Schlussendlich kommt es jedoch vor allem auf die Wünsche des Kunden an, wie die Programme zu funktionieren haben.

6 Zusammenfassung und Ausblick

Die vorliegende Diplomarbeit beschreibt eine softwarebasierte Lösung zum Aufzeichnen von Voice over IP Daten, welche mit Hilfe von Zeitstempeln einem exakten Zeitpunkt zugeordnet werden können. Die aufgezeichnet Daten können später in einem eigens entwickelten Media Player zusammen mit archivierten Videomaterial wiedergegeben werden. Das Projekt wurde bei der Firma **AviBit data processing GmbH** in Graz abgewickelt.

Die Arbeit beschreibt im theoretischen Abschnitt den Weg von der Schallquelle bis hin zum digitalisierten und codierten Audiosignal, welches auf einem PC gespeichert und weiterverarbeitet werden kann. Kapitel 2 widmet sich zuerst den Grundlagen der Akustik. Dazu gehört Schall, dessen Erzeugung und Ausbreitung, sowie die Erklärung der physikalische Eigenschaften. Das analoge (Schall-)Signal kann erfasst und digitalisiert werden. Kapitel 3 beschreibt die Signalverarbeitung, welche für die Digitalisierung verantwortlich ist. Dafür sind Analog-Digital Wandler notwendig, welche eine Abtastung des Signals, eine Quantisierung und eine Codierung durchführen. Das Ergebnis ist digitaler Code, welcher als Binärdatei gespeichert werden kann. Zusätzlich sind in diesem Kapitel Verfahren zur Analog-Digital Wandlung beschrieben. Kapitel 4 geht nun davon aus, dass die Audiodaten digital vorhanden sind und beschreibt Möglichkeiten zur weiteren Verarbeitung, Speicherung und Übertragung. Ein wichtiger Aspekt dabei ist die Komprimierung (siehe Abschnitt 4.2), die nicht nur ermöglicht, die Audiodaten platzsparend abzulegen, sondern auch für eine geringere Übertragung an Datenmaterial sorgt. Dies hat den Vorteil, dass Audiodaten schneller übertragen werden können und somit eine Echtzeitkommunikation ermöglicht wird. Diese setzt eine möglichst geringe, also vom Menschen nicht mehr wahrnehmbare, Verzögerung voraus. Das ist einer der wichtigsten Aspekte der Voice over IP Telefonie, auch bekannt als die Internet-Telefonie.

Für den praktischen Teil der Arbeit spielt die Echtzeitfähigkeit jedoch eine untergeordnete Rolle. Die in Kapitel 5 beschriebene Software empfängt rein passiv Audiodaten, welche dann aufgezeichnet (archiviert) werden. Eine Zuordnung zu einem realen Zeitpunkt kann auch später erfolgen. Es ist nicht notwendig, dass die empfangen Daten sofort ausgegeben werden müssen, wie bspw. bei der Internet-Telefonie. Dennoch bot die Recherche zur Übertragung von digitalen Audiodaten (siehe auch Abschnitt 4.3) eine wichtige Grundlage für die angestrebte Implementation.

Kapitel 5 widmet sich schließlich dem *AviBit Legal Recording and Replay* System. Darunter wird alles verstanden, was Video- und Audiodaten aufzeichnen und später auch wiedergeben kann. Da das Video Recording aber nicht Teil des Projekts war, spielt es in der Diplomarbeit eine untergeordnete Rolle. Am Anfang des Kapitels wird Allgemeines erläutert (Abschnitt 5.1). Dazu gehört eine Einführung zum bereits vorhanden Video Recording Programm *VideoRec*, eine Anforderungsanalyse an das Audio Recording Programm *AudioRec* und die Erweiterung des Media Player *AVPlayer*. Desweiteren ist ein detaillierter Plan - die Roadmap - für die Durchführung des Projekts angegeben. Für das Audio Recording wurde auch eine eigene Hardware names *XC-1124A* (”24-Channel Network Audio System”) entwickelt. Diese Hardware bedient als Input bis zu 24 Audiokanäle und schickt die Audiodaten komprimiert und codiert via Netzwerk aus. *AudioRec* verbindet sich zu der Hardware und beginnt mit der Archivierung. Abschnitt 5.2 widmet sich dem Programm *AudioRec*, beschreibt das zu Grunde liegende Kommunikationsprotokoll, welches von der Hardware vorgegeben wurde und erläutert die Architektur und die Konfiguration der Software. Abschnitt 5.3 beschreibt die Erweiterung des Media Players *AVPlayer*. Dieser konnte ursprünglich nur Videomaterial wiedergeben und musste eben nun zusätzlich Audiomaterial unterstützen. Beide Medien müssen natürlich synchron abgespielt werden. Zur Beschreibung gehören Aspekte des Designs der grafischen Oberfläche des *AVPlayers*, die Architektur und Konfiguration. Den Abschluss bildet der Abschnitt 5.4, in dem die Tests für die Software erläutert sind.

Die ausführlichen Tests haben gezeigt, dass die Software die Anforderungen erfüllt und stabil läuft.

Sie zeigen jedoch nicht die Praxistauglichkeit. Es wäre nützlich, an Ort und Stelle ein längeres Recording durchzuführen (Field Recording) und diesen Prozess sowie die resultierenden Archive zu analysieren. Außerdem muss das ganze Setup, also die Verbindung von Software und Hardware, erprobt und getestet werden.

Neben der Stabilität der Programme ist sicher noch Aufwand in diverse Änderungen zu investieren, welche auf Kundenwünsche basieren. Im Allgemeinen verlangen die Kunden eine Anpassung an ihre Verhältnisse und Gegebenheiten. Ein möglicher Eingriff wäre beispielsweise das Design der grafischen Oberfläche des *AVPlayers* mit zusätzlichen Schaltflächen oder erweiterten Konfigurationsmöglichkeiten.

Dennoch sind die Programme in einem Status, der eine Präsentation vor Kunden erlaubt. Es kann gezeigt werden, dass mit Hilfe der Hardware XC-1124A Audio Recordings mit *AudioRec* durchgeführt und diese später zu passendem Videomaterial im *AVPlayer* abgespielt werden können.

Literaturverzeichnis

[AH03] AKESTER, Richard; HAILES, Stephen: A New Audio Skew Detection and Correction Algorithm / Department of Computer Science, University College London. Version: 2003. http://www.cs.ucl.ac.uk/staff/r.akester/skew2.pdf. 2003. – Paper (Zitiert auf Seite 26.)

[Ahl07] AHLERS, Heinfried: Theoretische Verfahren der Elektrotechnik / Fachhochschule Wilhelmshaven/Oldenburg/Elsfleth. Version: 2007. http://www.fh-oow.de/fbi/we/el/eg/DOWNLOAD/TVE/TVE_Skript_Kap_06_Diskrete_Systeme.pdf. 2007. – Skriptum (Zitiert auf Seite 13.)

[Ast10] ASTERISK DEVELOPMENT TEAM: *Asterisk Reference Information*. Version 1.6.1.6, 2010. http://www.asterisk.org (Zitiert auf Seite 28.)

[BS06] BASET, Salman A.; SCHULZRINNE, Henning: An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol / Department of Computer Science, Columbia University, New York. Version: 2006. http://www1.cs.columbia.edu/~salman/publications/skype1_4.pdf. 2006. – Paper (Zitiert auf Seite 28.)

[CHW09] CARÔT, Alexander; HOHN, Torben ; WERNER, Christian: Netjack - Remote music collaboration with electronic sequencers on the Internet / Institute of Telematics, University of Lübeck. Version: 2009. http://lad.linuxaudio.org/events/2009_cdm/Saturday/22_Hohn/22.pdf. 2009. – Paper (Zitiert auf Seite 26.)

[Cob] *CobraNet*. http://www.cobranet.info (Zitiert auf Seite 33.)

[DEG06] *Akustische Wellen und Felder - DEGA-Empfehlung 101*. http://www.dega-akustik.de/publikationen/DEGA_Empfehlung_101.pdf. Version: März 2006 (Zitiert auf den Seiten 4 und 7.)

[DPB+06] DAVIDSON, Jonathan; PETERS, James; BHATIA, Manoj; KALIDINDI, Satish ; MUKHERJEE, Sudipto: *Voice over IP Fundamentals*. Second Edition. Cisco Press, 2006. – ISBN 1–58705–257–1 (Zitiert auf den Seiten 27, 29 und 31.)

[Fle98] FLEISCHMAN, Eric: RFC 2361 / IETF. Version: 1998. http://www.ietf.org/rfc/rfc2361.txt. 1998. – Forschungsbericht (Zitiert auf Seite 20.)

[Grü08] GRÜNINGEN, Daniel Ch. V.: *Digitale Signalverarbeitung*. 4. Auflage. Carl Hanser Verlag München, 2008. – ISBN 978–3–446–41463–1 (Zitiert auf Seite 9.)

[Haz] *Hazelware: Mu-Law and A-Law Compression Tutorial*. http://hazelware.luggle.com/tutorials/mulawcompression.html (Zitiert auf Seite 24.)

[HBG05] HERING, Ekbert; BRESSLER, Klaus ; GUTEKUNST, Jürgen: *Elektronik für Ingenieure und Naturwissenschaftler*. Fünfte, aktualisierte Auflage. Springer Berlin Heidelberg, 2005 http://dx.doi.org/10.1007/b137683. – ISBN 978–3–540–26487–3 (Zitiert auf den Seiten 13, 14 und 16.)

[Hir03] HIRT, Norbert: Interfacetechnik (AD- und DA-Umsetzer) / Technische Universität Ilmenau. Version: 2003. http://www.tu-ilmenau.de/fakia/fileadmin/template/startIA/mhe/lehre/itechnik/V_ADgesamt_n.pdf. 2003. – Skriptum (Zitiert auf den Seiten 13, 15 und 16.)

[Hof] HOFBAUER, Gerhard: *XC-1124A HW-SW Interface*. Draft B (Zitiert auf den Seiten 41 und 42.)

[HR00] HÖSS, Thomas; RIECK, Tobias: *WAV-Audio-Format*. http://www.it.fht-esslingen.de/~schmidt/vorlesungen/mm/seminar/ss00/HTML/node107.html. Version: 2000 (Zitiert auf Seite 20.)

[HSF08] HENN, Hermann; SINAMBARI, Gh. R. ; FALLEN, Manfred: *Ingenieurakustik*. 4., überarbeitete und erweiterte Auflage. Vieweg+Teubner, 2008 http://dx.doi.org/10.1007/978-3-8348-9537-0. – ISBN 978–3–8348–9537–0 (Zitiert auf den Seiten 6 und 8.)

[ITUa] *ITU-T Recommendations*. http://www.itu.int/itu-t/recommendations/index.aspx (Zitiert auf Seite 22.)

[ITUb] *ITU-T Recommendations: G Series*. http://www.itu.int/rec/T-REC-g (Zitiert auf Seite 22.)

[ITUc] *ITU-T Recommendations: H Series*. http://www.itu.int/rec/T-REC-h (Zitiert auf Seite 29.)

[ITWa] *ITWissen: A-Law-Verfahren*. http://www.itwissen.info/definition/lexikon/A-Law-Verfahren-A-law-method.html (Zitiert auf Seite 22.)

[ITWb] *ITWissen: Mu-Law-Verfahren*. http://www.itwissen.info/definition/lexikon/micro-Law-Verfahren-micro-law-method.html (Zitiert auf Seite 22.)

[JAC] *JACK Audio Connection Kit*. http://jackaudio.org (Zitiert auf Seite 25.)

[Kab06] KABAL, Peter: *WAVE Audio File Format Specifications*. http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html. Version: 2006 (Zitiert auf Seite 21.)

[Kre] KREMER, Martina: *Akustik-Gehör-Psychoakustik*. http://www.dasp.uni-wuppertal.de/ars_auditus/index.html (Zitiert auf den Seiten 4, 5 und 6.)

[Lat98] LATTMANN, Marcel: Digitaltechnik / SwissEduc. Version: 1998. http://www.swisseduc.ch/informatik/hardware/analog_digital_wandler/docs/script.pdf. 1998. – Skriptum (Zitiert auf den Seiten 11, 14 und 15.)

[Ler08] LERCH, Alexander; WEINZIERL, Stefan (Hrsg.): *Bitdatenreduktion*. Springer Berlin Heidelberg, 2008. – 849 – 884 S. http://dx.doi.org/210.1007/978-3-540-34301-1. – ISBN 978–3–540–34301–1 (Zitiert auf Seite 21.)

[Loh08] LOHNINGER, H.: *Angwandte Mikroelektronik*. http://www.vias.org/mikroelektronik/. Version: 2008 (Zitiert auf den Seiten 11 und 14.)

[LSS+07] LAZARUS, Hans; SUST, Charlotte A.; STECKEL, Rita; KULKA, Marko ; KURTZ, Patrick: *Akustische Grundlagen sprachlicher Kommunikation*. Springer Berlin Heidelberg, 2007 http://dx.doi.org/10.1007/978-3-540-49986-2. – ISBN 978–3–540–49986–2 (Zitiert auf Seite 6.)

[Mös07] MÖSER, Michael: *Technische Akustik*. 7., erweiterte und aktualisierte Auflage. Springer Berlin Heidelberg, 2007 (VDI-Buch). http://dx.doi.org/10.1007/978-3-540-71387-6. – ISBN 978–3–540–71387–6 (Zitiert auf den Seiten 4, 5, 7 und 8.)

[Net] *NetJack*. http://netjack.sourceforge.net (Zitiert auf Seite 26.)

[Pet08] PETERMICHL, Karl; WEINZIERL, Stefan (Hrsg.): *Dateiformate für Audio*. Springer Berlin Heidelberg, 2008. – 649 – 718 S. http://dx.doi.org/210.1007/978-3-540-34301-1. – ISBN 978–3–540–34301–1 (Zitiert auf den Seiten 19, 20 und 21.)

[Pro] *Protocols.com*. www.protocols.com (Zitiert auf den Seiten 29 und 30.)

[Ria02] RIAT, Martin: *Grundlagen der Musik*. http://www.riat-serra.org/akustik.html. Version: 2002 (Zitiert auf den Seiten 4, 5 und 6.)

[Rot09] ROTHENBERG, Alexander: *Audionetzwerk mit EtherSound*. http://www.amazona.de/index.php?page=26&file=2&article_id=2516&page_num=1. Version: 2009 (Zitiert auf Seite 33.)

[RSC+02] ROSENBERG, Jonathan; SCHULZRINNE, Henning; CAMARILLO, Gonzalo; JOHNSTON, Alan; PETERSON, Jon; SPARKS, Robert; HANDLEY, Mark ; SCHOOLER, Eve: RFC 3261 / IETF. Version: 2002. http://www.ietf.org/rfc/rfc3261.txt. 2002. – Forschungsbericht (Zitiert auf Seite 31.)

[RTR06] RTR: Voice over IP - Grundlagen, Regulierung und erste Erfahrungen / Rundfunk und Telekom Regulierungs-GmbH. Version: 2006. http://www.rtr.at/de/komp/SchriftenreiheNr12006/Schriftenreihe_01_2006.pdf. 2006. – Paper (Zitiert auf den Seiten 29 und 31.)

[SCFJ03] SCHULZRINNE, Henning; CASNER, Stephen L.; FREDERICK, Ron ; JACOBSON, Van: RFC3550 / IETF. Version: 2003. http://www.ietf.org/rfc/rfc3550.txt. 2003. – Forschungsbericht (Zitiert auf Seite 31.)

[Sch03] SCHLATTER, C.: Basic Architecture of H.323 / The Swiss Education and Research Network. Version: 2003. http://hive.packetizer.com/users/packetizer/papers/h323/h323_basics_handout.pdf. 2003. – Presentation (Zitiert auf den Seiten 29 und 30.)

[Sch09] SCHUBERT, Matthias: *Mathematik für Informatiker*. Vieweg+Teubner, 2009 http://dx.doi.org/10.1007/978-3-8348-9585-1. – ISBN 978–3–8348–9585–1 (Zitiert auf Seite 10.)

[Set] SETZER, Max J.: *Der Schall*. http://www.uni-due.de/ibpm/BauPhy/Schall/indexschall.htm (Zitiert auf den Seiten 5, 6, 7 und 8.)

[Shu99] SHUVALOV, Andrew: *Real-Time TV Program Distribution and Storage Server with Keyword Access Capability*, State University of New York at Stony Brook, Diplomarbeit, 1999. http://www.ecsl.cs.sunysb.edu/~andrew/VideoServer/videoserver/thesis/book1.html (Zitiert auf Seite 25.)

[SKM+06] SCHMITZ, Roland; KIEFER, Roland; MAUCHER, Johannes; SCHULZE, Jan ; SUCHY, Thomas: *Kompendium Medieninformatik*. Springer Berlin Heidelberg, 2006 http://dx.doi.org/10.1007/3-540-30226-3. – ISBN 978–3–540–30226–1 (Zitiert auf den Seiten 17, 21 und 22.)

[Tel] *Tele-Communication (Telecom) Terms Glossary and Dictionary*. http://www.networkdictionary.com/telecom/dictionary.php (Zitiert auf Seite 9.)

[Tro05] TROLLTECH: *Qt Documentation*. Version 3.3, 2005. http://doc.trolltech.com/3.3/ (Zitiert auf den Seiten 44, 47 und 57.)

[Uns00] UNSER, Michael: Sampling - 50 Years After Shannon. In: *Proceedings of the IEEE* Bd. 88, 2000, 569 - 587 (Zitiert auf Seite 12.)

[Voi] *Voip Think: G.711 codec process*. http://www.voipthink.com/codec/codecs-g711-alaw.php (Zitiert auf Seite 22.)

[Wol04] WOLF, Ludwig: Multimedia Netz Praxis / Technische Universität Chemnitz. Version: 2004. http://www.tu-chemnitz.de/urz/lehre/mmn/scripte/html/. 2004. – Skriptum (Zitiert auf Seite 32.)

[WU07] WOITOWITZ, Roland; URBANSKI, Klaus: *Digitaltechnik*. Fünfte, neu bearbeitete und erweiterte Auflage. Springer Berlin Heidelberg, 2007 http://dx.doi.org/10.1007/978-3-540-73673-8. – ISBN 978–3–540–73673–8 (Zitiert auf den Seiten 9, 11, 12, 14, 15 und 16.)

[WW01] WATKINSON, John; WILKINSON, John: *The Art Of Digital Audio*. 3rd edition. Butterworth-Heinemann, 2001. – ISBN 0–240–51587–0 (Zitiert auf den Seiten 17 und 21.)