Entwurf und Implementierung eines Soundsystems für das Java-Betriebssystem JX

Studienarbeit im Fach Informatik

vorgelegt von

Gernot Payer

geb. am 17.10.1977 in Fürth/Bay.

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	***Dipl-Inf. Meik Felser*** ***Dipl-Inf. Christian Wawersich***
Beginn der Arbeit:	1. Juni 2003
Abgabe der Arbeit:	6. Februar 2004

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 7. November 2005 _____

Kurzzusammenfassung

Diese Studienarbeit beschreibt den Weg von der direkten Programmierung der Soundhardware über die Evaluierung verschiedener existierender Soundarchitekturen bis hin zur konkreten Implementierung einer Soundarchitektur (JX SoundEngine) in Java. Weiterhin wird auf die Möglichkeiten und Einschränkungen eingegangen, die sich durch die Systemarchitektur von JX [4] und die Tatsache, dass der weitaus größte Teil von JX direkt in Java implementiert ist, ergeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Um eine Soundarchitektur zu entwerfen, muß zuerst eruiert werden, welche Fähigkeiten aktuelle Soundhardware besitzt, und welcher Teilmenge dieser Fähigkeiten unterstützt werden soll. Zusätzlich muß zunächst festgelegt werden, wie das Design der Soundarchitektur aussehen soll, d.h. über welche Interfaces die Hardwaretreiber und der Benutzer mit der Vermittlungsschicht kommunizieren.

Nach dieser Analyse wurden die folgenden Anforderungen an die JX SoundEngine festgelegt. Es muß möglich sein, digitalen Sound in CD-Qualität abzuspielen und aufzunehmen. Monokanalsound sollte auch möglich sein. Falls die Karte Surroundsound oder 20 Bit Auslösung bietet, sollte dies auch unterstützt werden.

Ein zentraler Dienst muß existieren, der beim Booten die Hardwaretreiber initialisiert und deren Interfaces registriert. Über diesen Dienst sind dann die Namen der Sound- und Mixerdevices abfragbar, unter denen diese beim JX Naming Service registriert sind.

Die über den JX Naming Service herausgegebenen Interfaces sind eigenständige Interfaces, die die Fähigkeiten der JX SoundEngine widerspiegeln. Unterstützt werden müssen das Einstellen des verwendeten PCM Formats, das blockweise Abspielen und Aufnehmen und das streamorientierte Abspielen und Aufnehmen. Das Mixerinterface muß die Möglichkeit bieten, jedwede Einstellung des Mixers ändern zu können. Zusätzlich müssen noch Klassen existieren, die die konkreten Fähigkeiten eines Hardwaretreibers beschreiben, so dass sich die ihn benutzende Software daran anpassen kann.

Die Interfaces und Klassen des `javax.sound.sampled` Packages der J2SE 1.4 Spezifikation [2] sollen auch nachgebildet werden, so dass auch externe Programme mit der JX SoundEngine zusammenarbeiten können. Dieses Package setzt als High-Level Abstraktionsschicht auf den Interfaces der JX SoundEngine auf, da es sehr viel zusätzliche Funktionalität bietet, die sinnvollerweise nicht direkt im Hardwaretreiber implementiert wird.

Somit bietet diese Studienarbeit die vollständige Bandbreite an Schichten einer Soundarchitektur. Es werden Hardwaretreiber implementiert, es wird eine abstrakte Mittelschicht eingezogen, auf der dann die High Level Interfaces des `javax.sound.sampled` aufsetzen.

Um dem wissenschaftlichen Anspruch gerecht zu werden, wird das Endergebnis des praktischen Teils dieser Arbeit mit existierenden Soundarchitekturen, in diesem Fall OSS/Free und Alsa, in Hinblick auf gebotene Funktionen verglichen.

2 Soundhardware

Die Programmierung der gesamten Soundhardware gliedert sich in drei Hauptteile, die voneinander nur bedingt abhängig sind. Diese drei Teile sind, erstens, die Steuerbefehle für die sounderzeugenden Bereiche, zweitens die Befehle an den Mixer, und drittens schließlich die Programmierung der DMA-Controller. Von diesen drei Teilen ist der Mixer programmiertechnisch unabhängig von den beiden anderen, er kann in jeder Situation

angesprochen werden. Da er in der Sounderzeugungskette ganz am Schluß steht, entscheidet er zwar, welche Kanäle in welcher Lautstärke/Klangprägung den Lautsprecher erreichen, jedoch ändert diese signaltechnische Zusammenschaltung nichts daran, dass der Mixer ein unabhängiger Chip ist, der über eigene Befehlsports angesteuert wird.

Die Soundchips und die DMA-Controller sind zwar unabhängig voneinander ansprechbar, da aber ihr konzertiertes Zusammenspiel für den Datenfluß vom Hauptspeicher zum Mixer essentiell ist, gibt es einen engen zeitlichen Rahmen für deren Programmierung.

Im folgenden werden nun wichtige Einzelelemente der sounderzeugenden Hardware vorgestellt, außerdem wird auf ihre Programmierschnittstellen und auf ihre Kommunikation mir der Außenwelt eingegangen.

In weiteren Unterkapiteln werden dann die SoundBlaster 16 Karte und der Intel i810 Chipsatz inklusive angeschlossenem AC97 Codec im Detail vorgestellt. Für diese wurden im Rahmen dieser Arbeit Low-Level-Treiber geschrieben, außerdem eignen sie sich gut, um den Fortschritt in der PC-Architektur darzustellen.

2.1 Allgemeines

- **D/A und A/D Wandler**

  Ein D/A Wandler [1] wandelt einen digitalen Bitstrom in eine analoge Wellenform um. Dabei gibt es mannigfaltige Umrechnungsparameter zu beachten. So muß festgelegt werden, wieviele Bits einen Amplitudenwert ergeben, welche Zeitdifferenz zwischen den einzelnen Amplitudenmessungen existiert, ob die einzelnen Amplituden als *signed* oder *unsigned* zu interpretieren sind und welche Endianess vorliegt. Ein gängiger Wert ist z.B. 44 kHz Abtastfrequenz, 16 Bit Little Endian, signed. Das bedeutet, dass 44000 Amplitudenmessungen pro Sekunde existieren und jede dieser Messungen einen von 2^{16} Werten annehmen kann. Bei der Übertragung eines dieser Werte kommt erst das Lowbyte, dann das Highbyte. Zusätzlich ist dieser 16 Bit Wert als *signed* zu interpretieren. Ein A/D Wandler ist die exakte Umkehrung eines D/A Wandlers: er akzeptiert ein analoges Signal und gibt einen digitalen Bitstrom aus. Für den A/D Wandler existieren die gleichen Umrechnungsparameter, wie für den D/A Wandler, nur dass ersterer dieses Datenformat erzeugt. Für eine anschauliche Beschreibung des Digitalisierungsvorgangs siehe Abbildung 1.

- **Andere Sounderzeuger**

  Ein weiteres Standardmittel zur Sounderzeugung sind die Synthesizer. Da diese jedoch im Rahmen dieser Studienarbeit nicht unterstützt werden, werden sie hier nur der Vollständigkeit halber erwähnt.

  Als PC-Hardware existieren folgende Synthesizertypen: AM-, FM- und Wavetablesynthesizer. Von diesen wird jedoch nur noch der Wavetablesynthesizer unterstützt, da er die beiden älteren Systeme in ihren Möglichkeiten weit schlägt.

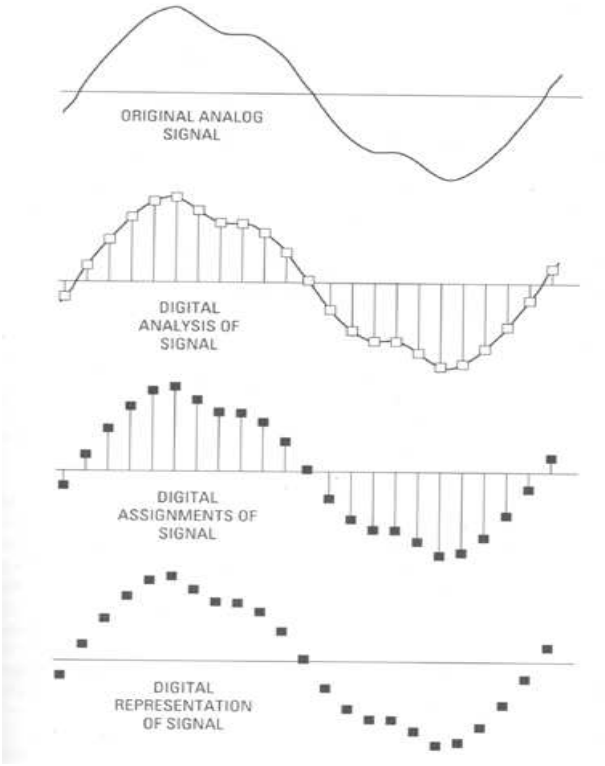

Abbildung 1: Digitalisierung eines analogen Signals

Bei der AM-Synthese werden Sinuswellen verschiedener Frequenz mittels Koeffizienten amplitudenmoduliert, um dann zum Ausgangssignal addiert zu werden.

Bei der FM-Synthese wird in sogenannten Operatoren eine Carrierfrequenz mit einem Modulator frequenzmoduliert. Das Ausgangssignal eines solchen Operators kann wiederum als Carrier oder Modulator eines anderen Operators dienen. Das Ausgangssignal jedes Carriers oder Modulators wird zusätzlich mit einer Hüllkurve amplitudenmoduliert, so dass endliche Klänge erzeugt werden können. Jedoch wurde in keiner PC Soundkarte die FM-Synthese in dieser Allgemeinheit und Komlexität umgesetzt, mehr als vier Operatoren mit wenigen vordefinierten Verschaltungsmöglichkeiten gab es nie.

Die Wavetable-Synthese ist von ihrer Funktionsweise am einfachsten zu verstehen, hat aber auch die höchsten Anforderungen an die Hardware. Hier werden fertige kurze Soundstückchen (z.B. Aufnahme eines echten Instruments, künstlich erzeugter Sound) auf die gewünschte Frequenz transponiert, aneinander gehängt, um die gewünschte Länge zu erzeugen, und schließlich mittels einer Hüllkurve naturalisiert. Mittels mehrerer solcher Wavetableuntereinheiten lassen sich extrem realistische Instrumentenklänge erzeugen.

- **Ports**

Ältere ISA PC Hardware besitzt üblicherweise nur sehr wenige Ports: einen Leseport, einen Schreib-, bzw. Befehlsport, einen Statusport und möglicherweise noch einen weiteren Hilfport oder einen Port um erhaltene Interrupts zu bestätigen.

Das Standardprotokoll zum Abschicken eines Befehls schaut folgendermaßen aus: Zuerst wird gewartet, bis der Schreibport bereit ist. Dazu wird solange vom Statusport gelesen, bis das entsprechende Statusbit den korrekten Wert besitzt. Ein ähnliches Bit gibt es auch für die Bereitschaft des Leseports. Nun wird der gewünschte Befehlscode in den Schreibport geschrieben. Je nach Befehlsart werden nun weitere Daten erwartet, oder es können Daten am Leseport abgeholt werden.

Modernere PC-Erweiterungskarten für den PCI Bus besitzen oftmals deutlich größere I/O Addressräume. Somit ist es möglich, jede Einstellmöglichkeit direkt zugreifbar zu gestalten. Bei einem solchen Interface ist auch nicht mehr unbedingt nötig, einen speziellen Befehlsport zu besitzen, da z.B. Soundmodi bequem per Setzen von Flags eingestellt werden können.

- **Hardwareinterrupts**

  Damit die Hardware ansynchron mit dem Hauptprozessor (bzw. dem darauf laufenden Betriebssystem) kommunizieren kann, gibt es die *Hardwareinterrupts*. Löst eine Hardwarekomponente einen solchen Interrupts aus, wird der aktuelle Befehlsfluß im Hauptprozessor unterbrochen und dieser springt die diesem Interrupt zugeordnete Subroutine an. In modernen Betriebssystemen auf moderner Hardware gehört diese Subroutine üblicherweise zum Betriebssystem und läuft mit der höchsten Berechtigungsstufe. Hier wird nun per systemüblicher Kommunikation der entsprechende Treiber, Userspaceprozess usw. über den Interrupt informiert, so dass entsprechend reagiert werden kann.

- **Beschreibung der wichtigsten Befehlstypen**

  Da es drei Hauptkomponenten in einer Soundhardwareeinheit gibt, gibt es auch drei Hauptgruppen von Befehlen und Optionen: die der DMA Controller, die des Mixers und die der sounderzeugenden Chips.

  So besitzt der DMA Controller Einstellmöglichkeiten, die ihm mitteilen, welche Speicherbereiche er zur Soundhardware übertragen soll, bzw. welche Speicherbereiche mit aufgenommenem und digitalisiertem Sound gefüllt werden sollen. Zusätzlich gibt es noch Optionen, bei welchen Ereignissen (z.B. letzter Puffer abgearbeitet) ein Interrupt ausgelöst werden soll, ob am Ende des (letzten) Puffers gestoppt oder wieder ab Anfang übertragen werden soll.

  Im Mixer lassen sich für jeden Ein- und Ausgangskanal die Lautstärke (bzw. bei Mehrkanalton auch die einzelnen Subkanäle) und evtl. auch Klangcharakteristiken einstellen. Bei neuerer Hardware gibt es da nicht nur die Regelung von linkem und rechtem Kanal, sondern auch die Möglichkeit Surroundsound auszugeben oder 3D-Klang anzuschalten. Eingangskanäle sind z.B. Line-In, der Mikrophoneingang (mono), bei aktueller Soundhardware ein S/PDIF Anschluß (*Sony/Philips Digital*

Interface) oder der CD-Eingang, der es erlaubt beim Abspielen einer Audio-CD im CD-ROM deren Audiosignale direkt an die Soundkarte weiterzuleiten. An Ausgangskanälen besitzt normalerweise nur Line-Out, bei neuerer Hardware existiert aber auch oftmals ein S/PDIF Anschluß.

Die Sounderzeugerhardware sich läßt in drei weitere Subkomponenten unterteilen: die D/A und A/D Wandler, die Syntheseeinheit (AM, FM oder Wavetable) und das MIDI Interface. Letzteres ist ein spezieller Fall, da ein dem MIDI Standard [3] entsprechender Befehlsstrom erwartet wird, so daß lediglich die Angabe, wie denn diese Befehle zu bekommen seien (z.B. via DMA), herstellerspezifisch ist. Bei dem D/A bzw A/D Wandler sind die Samplerate, die Bits pro Sample und das Byteformat einstellbar, bei den Syntheseeinheiten gibt es pro Synthesekanal spezifische einstellbare Optionen, z.B. bei der FM-Synthese u.a. die Carrier- und Modulatorenfrequenz und die Hüllkurvenparameter.

- **ISA und PCI**

  Auch wenn aktuelle PCs meistens nur noch über einen PCI-Bus verfügen, ist der ISA-Bus doch noch sehr weit verbreitet. Somit gibt es noch ziemlich viele Soundkarten (und andere Erweiterungskarten) für den ISA Bus.

  Der ISA-Bus war der erste PC-Standard für Erweiterungskarten und hat trotz seiner Nachteile (u.a. langsame Datenübertragungsraten, keine automatische Konfiguration der Hardware, d.h. manuelle Konfiguration mittels Jumper, 16 Bit I/O Addressraum) bis heute überlebt, da seine Vorteile für gewisse Anwendungen dir Nachteile überwiegen: er ist extrem billig und Hardware dafür zu entwerfen ist sehr einfach. Eine Vertreterin dieser Gattung Hardware ist die SoundBlaster 16 Erweiterungskarte, auf die in späteren Abschnitten noch genauer eingegangen wird.

  Der PCI-Bus ist der aktuelle Standard für PC-Erweiterungskarten. Er erlaubt weitaus höhere Datenübertragungsraten als der ISA-Bus, hat einen 32 Bit Addressraum, ermöglicht selbstständige Konfiguration der Hardware durch das BIOS und unterstützt Interruptsharing. Der einzige Nachteil ist ein höherer Entwicklungsaufwand bei Hardware (komplexes PCI-Protokoll) und teilweise bei der Software (PCI Konfiguration muß vom Betriebssystem ausgelesen werden). Benutzerfreundlicher wird jedoch der für eine PCI-Karte entwickelte Hardwaretreiber, da dieser nur die Konfiguration “seiner” Hardware beim Betriebssystem erfragen muß und dann gleich direkt zugreifen kann.

2.2 Besonderheiten des Mixers

Die eigentliche Besonderheit des Mixers ist seine Unabhängigkeit vom Rest des Soundsystems, und die programmiertechnische Unabhängigkeit seiner einstellbaren Optionen. Dies liegt daran, dass er Mixer lediglich die bereits analogen Signale von und zu den Signalgeneratoren mittels zustandsloser Funktionen modifiziert. So kann für jeden der Ausgangskanäle Lautstärke bzw. Gain und möglicherweise auch Baß und Höhen unabhängig

geregelt werden. Das selbe gilt für die Eingangskanäle. Hier kann jedoch zusätzlich geregelt werden, welche Eingangskanäle an den A/D Wandler weitergeleitet werden. Bei moderner Hardware können hier auch R- und L-Kanal unterschiedlicher Eingänge gemischt werden.

Zusätzlich gibt es möglicherweise noch Regelungsmöglichkeiten für das sogenannte *3D Stereo Enhancement* oder auch Power Management. Somit ist es dem Hardwaretreiber möglich, die Hardware in einen stromsparenden Modus zu versetzen, wenn das Betriebssystem in einen Sleep-Modus wechselt. Um Sound abspielen zu können, muß das Soundsystem natürlich im normalen Betriebsmodus sein, das sonst nicht genügend Ausgangsleistung erreicht werden kann.

2.3 Besonderheiten des DMA Controllers

DMA Controller für PCs gibt es seit 1981. Ihre Aufgabe ist es, Daten direkt zwischen Hauptspeicher und Hardware auszutauschen, d.h. ohne Beteiligung der CPU. Für diesen Zweck stellt die Mainboardhardware DMA Kanäle zur Verfügung, welche für den Datentransfer verwendet werden. Je nach verwendetem Bus (z.B. ISA oder PCI), muß der Hardware entweder explizit mitgeteilt werden, welche DMA-Kanäle sie benutzen darf, oder sie besitzt eigene DMA Controller, die über das Bus Mastering Feature des PCI Bus den Direktzugriff starten.

Der erste Schritt zur Programmierung eines DMA-Controllers ist es, ihm den zu verwendenden Speicherbereich mitzuteilen. Bei diesem Schritt sind mögliche Beschränkungen des Controllers zu beachten. So können ältere DMA-Controller, die für ISA Hardware entworfen wurden, z.B. nur auf die untersten 16 MB des Hauptspeichers zugreifen, was spezielle Mechanismen der DMA-Puffer-Allozierung im Betriebssystem notwendig macht. Auch kann man solchen Controllers nur einen einzigen zusammenhängenden Puffer von begrenzter Länge (z.B. 128 KB) übergeben, was wiederum spezielle Mechanismen im Treiber erfordert, falls dieser mehr als eine Pufferlänge an Daten verarbeiten soll. Modernere DMA-Controller, die am PCI-Bus angeschlossen sind, können weitaus mehr. Sie können deutlich mehr Arbeitsspeicher direkt adressieren, haben eine komplexere Pufferverwaltung (mehrere getrennte Puffer, die über eine Liste verwaltet werden) und erlauben, schon alleine wegen des schnelleren PCI Busses, weitaus höhrere Datenübertragungsraten. Im Zusammenhang mit Sound fällt letzterer Vorteil allerdings kaum ins Gewicht.

2.4 Konkrete Hardware

Anhand zweier Karten, für die im Rahmen des JXSound Studienarbeitprojektes Treiber entwickelt wurden, wird nun das Zusammenspiel der Einzelkomponenten demonstriert. Da das Timing der Befehle an die Subkomponenten oftmals sehr kritisch ist, soll hier auch hervorgehoben werden, dass man wichtige Erkenntnisse über die Bedienung der Hard-

ware aus den entsprechenden Treiberquelldateien des Linux-Kernels gewinnen kann[1].

2.4.1 Sound Blaster 16

Die Sound Blaster 16 Karte wurde von Creative Labs 1992 als Nachfolgerin der äußerst erfolgreichen Sound Blaster Pro und Sound Blaster Karten auf den Markt gebracht. Mit der originalen Sound Blaster hatte Creative Labs bereits 1989 den de facto Standard für PC Soundkarten gesetzt. Dieser Erfolg sollte sich auch mit der Sound Blaster 16 einstellen, wodurch der Standard für 16 Bit PC Soundhardware wiederum von Creative Labs gesetzt wurde.

Da diese Karte aus Treibersicht recht einfach zu benutzen ist, und viele, auch aktuelle Karten, einen Sound Blaster 16 Kompatbilitätsmodus besitzen und zudem diese Karte auch als virtuelle Soundkarte in PC-Emulatoren existiert (u.a. bochs[2], VMWare[3]), war es naheliehend, einen ersten einfachen Treiber für diese Hardwareplattform zu schreiben.

- **Features** Die Sound Blaster 16 Karte [5] ist für die 16 Bit Version des ISA-Busses gemacht und unterstützt in ihren neueren Inkarnationen (Sound Blaster 16 PnP) auch die Plug"n"Play Ergänzung des ISA-Protokols.

  Die Karte unterstützt 8 und 16 Bit Sampling mit Raten von 5 kHz bis 44.1 kHz, zwanzigstimmige FM-Synthese mit 4 Operatoren und den MIDI Standard.

  An Soundschnittstellen besitzt die SB16 Line-in (Stereo), einen Mikrophoneingang (Mono), Line-out (Stereo), einen Eingang für den PC Lautsprecher und einen Anschluß für den Audioausgang des CD-ROMs.

  Zusätzlich besitzt die SB16 noch einen Anschluß für eine Wavetable Synthese Erweiterungskarte, einen Port für ein IDE CD-ROM und einen Joystick Port.

  Zur Datenübertragung verwendet die SB16 die für den ISA-Bus zuständigen DMA-Controller des Mainboards, welche, bedingt durch das Alter des ISA-Busses, deutliche Einschränkungen haben. So können sie nur die unteren 16 MB des Hauptspeichers direkt zugreifen, zusätzlich darf der Lese-/Schreibpuffer keine 64 KB (128 KB beim 16 Bit DMAC) Speicherseitengrenze überschreiten.

- **Befehlsübersicht Sound**

  Die Sound Blaser 16 Karte stellt eine Reihe von Ports bereit, über die sie programmiert werden kann. Da für diese Studienarbeit nur die Ports des DSP (*Digital Signal Processor*) relevant sind, werden hier auch nur diese dokumentiert (siehe Tabelle 1). Die Ports sind hier als Offset zu der per Jumper (oder PnP) konfigurierbaren Basisadresse der Karte angegeben. Ein üblicher Wert wäre z.B. $260h$, hier läge der Resetport dann auf Adresse $266h$.

[1] z.B. `linux-2.4.24/drivers/sound/i810_audio.c`
[2] *Bochs IA-32 Emulator Project*, `http://bochs.sourceforge.net`
[3] *VMWare Workstation*, VMware Inc., `http://www.vmware.com`

Offset	Beschreibung
06h	DSP Reset
0Ah	DSP Leseport
0Ch	DSP Schreib- und Befehlsport, Schreibpufferstatus (Bit 7)
0Eh	DSP Lesepufferstatus (Bit 7), Interruptbestätigung
0Fh	16 Bit Interruptbestätigung

Tabelle 1: SB16 DSP Portoffsets zur Portbasisadresse

Kommando	Beschreibung
41h	Output Sample Rate setzen (erwartet High-, Low-Byte am Schreibport)
42h	Input Sample Rate setzen (erwartet High-, Low-Byte am Schreibport)
D0h	8 Bit Sound I/O Pausieren
D4h	8 Bit Sound I/O Fortsetzen
D5h	16 Bit Sound I/O Pausieren
D6h	16 Bit Sound I/O Fortsetzen
D9h	16 Bit Sound I/O am Ende des aktuell programmierten Blockes stoppen
DAh	8 Bit Sound I/O am Ende des aktuell programmierten Blockes stoppen
E1h	DSP Versionsnummer (2 Bytes am Leseport: Major, Minor)
Bxh	16 Bit DMA Sound I/O Programmieren
Cxh	8 Bit DMA Sound I/O Programmieren

Tabelle 2: SB16 DSP Befehle

Um einen Befehl an die SB16 zu übermitteln, muß solange gewartet werden, bis Bit 7 am Befehlsport $0Ch$ den Wert 0 annimmt, dann kann ein Byte übermittelt werden. Im Falle eines Befehls, der zusätzliche Angaben erfordert, muß dieser Schritt solange wiederholt werden, bis alle Bytes übermittelt sind. Für eine Liste der SB16 DSP Befehle siehe Tabelle 2

Lesen vom Leseport erfolgt analog. Es wird solange gewartet, bis Bit 7 am Statusport $0Eh$ gesetzt ist, dann kann ein Byte vom Leseport $0Ah$ gelesen werden.

Das Format der Befehle Bxh und Cxh ist ein Spezialfall, hier werden spezifische Angaben in den Kommandocode integriert und zusätzlich ein Modusbyte und zwei Bytes Längenangabe am Schreibport erwartet (siehe Abbildung 2). Die Begriffe SC (*single-cycle DMA*) und AI (*auto-initialized DMA*) werden später noch im Detail behandelt.

Offset	Beschreibung
04h	Indexport
05h	Schreib-/Leseport

Tabelle 3: SB16 Mixer Portoffset zur Basisportadresse

Addr	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
00h	Reset (Zurücksetzen aller Mixerregister)							
04h	PCM Volume links (alt)				PCM Volume rechts (alt)			
0Ah						Mic Volume (alt)		
0Ch			Input-Filter			ADC		
0Eh			Dfni			Vstc		
22h	Master Volume links (alt)				Master Volume rechts (alt)			
26h	FM Volume links (alt)				FM Volume rechts (alt)			
28h	CD Volume links (alt)				CD Volume rechts (alt)			
2Eh	Line Volume links (alt)				Line Volume rechts (alt)			
30h	Master Volume (L)							
31h	Master Volume (R)							
32h	PCM Volume (L)							
33h	PCM Volume (R)							
34h	MIDI Volume (L)							
35h	MIDI Volume (R)							
36h	CD Volume (L)							
37h	CD Volume (R)							
38h	Line Volume (L)							
39h	Line Volume (R)							
3Ah	Mic Volume							
3Bh	PC Speaker Volume							
Output Mixer Switches:								
3Ch				Line (L)	Line (R)	CD (L)	CD (R)	Mic
Input (L) Mixer Switches:								
3Dh		MIDI (L)	MIDI (R)	Line (L)	Line (R)	CD (L)	CD (R)	Mic
Input (R) Mixer Switches:								
3Eh		MIDI (L)	MIDI (R)	Line (L)	Line (R)	CD (L)	CD (R)	Mic
3Fh	Input Gain (L)							
40h	Input Gain (R)							
41h	Output Gain (L)							
42h	Output Gain (R)							
43h								AGC
44h	Treble (L)							
45h	Treble (R)							
46h	Bass (L)							
47h	Bass (R)							

Tabelle 4: SB16 Mixer Register

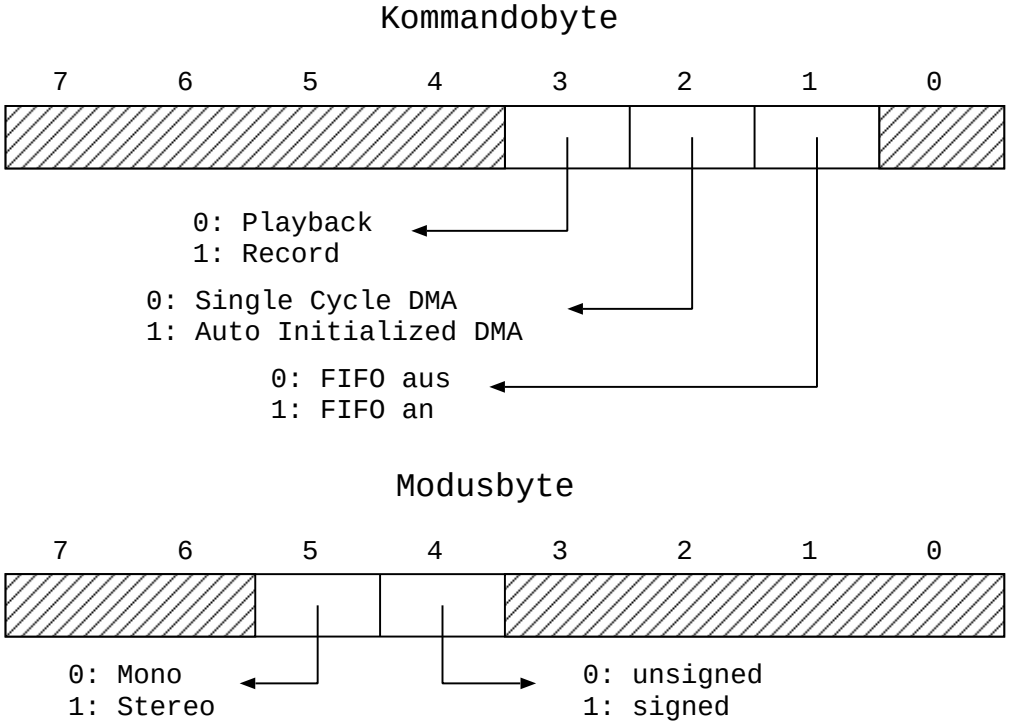

Abbildung 2: Bitbelegung des SB16 Abspiel-/Aufnahmekommandos

- **Befehlsübersicht Mixer**

  Die Einstellungen des Mixers werden in zwei Schritten geändert. Zuerst wird der Index des gewünschten Mixerregisters an die an die Adresse des Indexports geschrieben (siehe Tabelle 3), dann kann, nach einer Wartezeit von $20\mu s$, das Mixerregister ausgelesen und neu geschrieben werden. In Tabelle 4 sind alle Mixerregister aufgelistet. Um gegenüber den Vorgängerkarten kompatibel zu sein, sind einige Einstellungsmöglichkeiten doppelt vorhanden.

  Die Mixerswitches bestimmen, welcher interne Ausgabekanal (also bspw. der Ausgang eines D/A-Wandlers) auf welchen physikalischen Ausgabekanal abgebildet wird. Auf diese Weise können unterschiedliche Quellen (z.B. Musik von der CD und digitale Samples) zusammengemischt werden.

  Neben den Lautstärkeregelungen gibt es noch eine Reihe von Spezialschaltern:

  - **Input-Filter**: 001: Hochpass-Filter, 000: Tiefpass-Filter, 1xx: kein Filter
  - **ADC**: Auswahl der Signalquelle; 00: Mikrofon, 01: CD, 11: Line-in
  - **DNFI**: Noise-Filter am Ausgang (1=Ein)
  - **VSTC**: Stereo/Mono Control (1=Stereo)
  - **AGC**: Automatic Gain Control (1=Ein): regelt den Pegel des Mikrofoneingangssignals automatisch runter, wenn dieser zu hoch ist

- **Output Mixer Switches**: Kontrolle, welche Kanäle ausgegeben werden sollen
- **Input Mixer Switches**: Kontrolle, von welchen Kanälen aufgenommen werden soll

- **Befehlsübersicht DMA-Controller**

  Zu einem ISA-Bus gehören 2 DMA Controller, einer für 8 Bit Transfers (DMAC1) und einer für 16 Bit Transfers (DMAC2). DMAC1 kontrolliert die DMA Kanäle 0 bis 3, DMAC2 die Kanäle 4 bis 7.

Adresse		**Funktion**
DMAC1	**DMAC2**	
0A	**D4**	Maskierungsregister
0B	**D6**	Modusregister
0C	**D8**	Byte Pointer Flip-Flop
00 .. 07	**C0 .. CE**	Adress- und Längenregister für jeden Kanal

Tabelle 5: ISA DMA Kontrollregister

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Maskierungsregister						Maske	Kanal	
Modusregister	Modus		DEC	AI	Transfer		Kanal	

Tabelle 6: Beschreibung der ISA DMA Kontrollregister

Jeder DMA Controller besitzt 3 Kontrollregister: ein nur schreibbares Maskierungsregister, um einzelne DMA Kanäle zu aktivieren oder deaktivieren, ein nurschreibbares Modusregister, um die Art der Übertragung für jeden DMA Kanal festlegen zu können, und ein Byte Pointer Flip-Flop, über welches der interne Zähler des DMA Controllers zurückgesetzt werden kann (Tabelle 5).

Die Belegung der einzelnen Bits des Maskierungs- und des Modusregisters ist in Tabelle 6 dargestellt. Dabei haben die einzelnen Parameter folgende Bedeutung:

- **Maske**: 1: Maskierungsbit wird gesetzt (Kanal wird deaktiviert), 0: Maskierungsbit wird gelöscht (Kanal wird aktiviert)
- **Kanal**: binäre Kodierung der Kanalnummer (Bei 16 Bit Kanälen: Kanalnummer - 4)
- **Modus**: 00: Demand Mode, 01: Single Mode, 10: Block Mode, 11: Cascade mode (hier ist lediglich der Single Mode relevant)
- **DEC**: 0: Adresse wird hochgezählt, 1: Adresse wird heruntergezählt

- **AI**: 0: single-cycle DMA, 1: auto-initialized DMA
  Single-Cycle bedeutet, dass der angegebene Block genau einmal übertragen wird, bei auto-initialized DMA wird der Puffer immer wieder von Beginn an übertragen.
- **Transfer**: 00: Verfifizierung des Transfers, 01: Record, 10: Playback, 11: illegal (Transfermodus wird im Cascade Mode ignoriert)

Die Belegung des Modusregisters bestimmt schließlich, wie nun im Detail die Übertragung stattfinden soll. Übliche Werte für Sound I/O sind z.B. $58h+Kanalnummer$ (Single Mode, Adressinkrementierung, auto-initialized DMA, Playback) oder $54h+Kanalnummer$ (Single Mode, Adressinkrementierung, auto-initialized DMA, Record).

DMAC1 ist für die Kanäle 0,1,2 und 3 zusändig, DMAC2 betreut Kanal 4,5,6 und 7. Obwohl die Register für Kanal 4 zugreifbar sind, sollten diese nicht verwendet werden, da Kanal für interne Kommunikationszwecke zwischen den DMA Controllern verwendet wird, ähnlich dem *cascading interrupt*.

Zu jedem DMA Kanal gehören ein Adressregister, ein Längenregister (siehe Tabelle 5) und ein Seitenregister (Tabelle 7). Adress- und Längenregister liegen in dieser Reihenfolge hintereinander im angegebenen I/O-Bereich. Beim DMAC1 liegen die Register 1 Byte auseinander, beim DMAC2 beträgt das Offset 2 Byte.

Die Adresse des für den DMA zu benutzenden Speicherbereichs wird nicht linear übergeben, sondern getrennt in eine Speicherseitenzahl und die relative Adresse innerhalb dieser Seite. Als Seitengröße verwenden sowohl DMAC1 als auch DMAC2 64 KB Seiten. Das Offset innerhalb dieser Seite wird dann jedoch beim DMAC2 in 16 Bit Einheiten angegeben, gleiches gilt für die Länge des Puffers. Zusätzlich muß beachtet werden, dass im Längenregister der Wert (Pufferlänge - 1) erwartet wird.

Die Längen- und Adressregister sind nur 8 Bit breit, erwarten jedoch einen 16 Bit Wert. Dieser Atavismus wird umgangen, indem man zuerst das Low-Byte, dann das High-Byte des Wertes in das entsprechende Register schreibt.

Kanalnummer	0	1	2	3	5	6	7
Adresse	87h	83h	81h	82h	8Bh	89h	8Ah

Tabelle 7: ISA DMA Seitenregister

- **Initialisierung**

  Die Initialiserung erfolgt in 6 Schritten:

  1. Schreibe eine 1 an den Reset Port (2x6)
  2. Warte 3 μs

3. Schreibe eine 0 an den Reset Port
4. Warte bis Bit 7 am Lesepufferstatusport gesetzt ist
5. Lese vom Leseport, bis AAh empfangen wird
6. Resette den Mixer ($00h$ an Mixer Index Register, 20 μs warten, 1 an Mixer Datenregister)

Nach der Initialisierung sollte die DSP Versionsnummer (Befehlscode $E1h$) abgefragt werden, da auch die älteren Sound Blaster und Sound Blaster Pro auf diese Weise initialisiert werden. Weil diese Karten keinen 16 Bit Sound unterstützen, muss der Hardwareprogrammierer zwischen ihnen unterscheiden können. DSP Versionen ab 4.0 sind 16 Bit fähig.

- **Das Zusammenspiel am Beispiel der Soundausgabe**

  Um nun tatsächlich Sound ausgeben zu können, müssen DSP, DMA Controller und der Mixer konzertiert angesteuert werden. Insbesondere die Befehle an den DSP und den DMA Controller müssen aufeinander abgestimmt sein, damit der Datentranfer reibungslos verläuft und der Datenstrom auch richtig interpretiert wird.

  Zuerst müssen Vorarbeiten erledigt werden, die betriebssystemspezifisch sind. Ein DMA-Puffer muß alloziert werden, dieser muß in den unteren 16 MB des Hauptspeichers liegen und darf keine 64 KB (bzw. 128 KB bei 16 Bit) Seitengrenze überschreiten. Weiterhin muß ein Interrupthandler installiert werden, der die Interrupts der SB16 abfängt. Welcher Interrupt ausgelöst wird, ist per Jumper einstellbar. Interrupts werden von der SB16 außschließlich dann ausgelöst, wenn sie die im entsprechenden Sound I/O Befehl angegebene Pufferlänge übertragen wurde.

  Nun wird der gewünschte DMA Controller programmiert. Erst wird der zu verwendende DMA Kanal mittels des Maskierungsregisters deaktiviert, dann wird der Byte Pointer zurückgesetzt, indem ein beliebiger Wert an das Byte Pointer Flip-Flop geschrieben wird. Nun kann der gewünschte DMA Modus in das Modusregister geschrieben werden. Als nächstes wird die Adresse und die Länge des Puffers an die entsprechenden Register übergeben (Seite, Offset, Länge-1). Jetzt ist der DMA Controller vollständig programmiert und der gewünschte DMA Kanal kann wieder aktiviert werden.

  Um den gewählten Speicherbereich nun abzuspielen, muß noch der DSP programmiert werden. Dazu wird die Sample Rate (Befehl $E1h$) gesetzt und das entsprechende I/O Kommando gegeben (z.B. $B6h$ für 16 Bit auto-init Output, siehe Tabelle 2). Nun erwartet die Soundkarte 3 weitere Bytes am Leseport: das Datenformat und die um eins verringerte Länge des abzuspielenden Blockes (Reihenfolge: Low-Byte, High-Byte).

  Sobald die Soundkarte die mit dem I/O Kommando übergebene Anzahl an Samples verarbeitet hat, löst diese den vereinbarten Interrupt aus. Nun kann die Softwa-

re geeignet reagieren, um zu verhindern, dass der Datenfluß an die Soundkarte unterbrochen wird, was unangenehmes Knacksen im Lautsprecher zur Folge hätte.

Da die ISA-Bus DMA Controller lediglich auf einen Speicherbereich programmiert werden können, hat es sich eingebürgert, diesen in zwei Teilpuffer zu unterteilen. D.h. der DMA Controller wird auf *auto-initialize* eingestellt und bekommt die volle Länge des Puffers einprogrammiert, die Soundkarte aber nur die Hälfte der Länge. Nachdem nun die Hälfte des Puffers abgespielt ist, löst die Soundkarte einen Interrupt aus. Dieser Interrupt wird gegenüber der Soundkarte bestätigt, indem vom Interruptbestätigungsport ($2xE$ für 8 Bit, $2xF$ für 16 Bit Sound) gelesen wird. Auch gegenüber dem PIC muß das Interrupt eventuell bestätigt werden, dies wird jedoch üblicherweise vom Betriebssystem erledigt.

2.4.2 i810

Hinter dem gebräuchlichen Namen i810 Audio, der so auch als Treibername im Linuxkernel 2.4 verwendet wird, steckt etwas deutlich komplizierteres, als nur ein Soundchip. Der Audioteil ist nur ein kleiner Teil der Intel 82801 I/O Controller Hub Baureihe. Dieser Hub bietet nicht nur Zugang zu einem Audio Codec, er stellt alle grundlegenden Interfaces zur Verfügung: USB, IDE, ACPI, SMBus und DMA Controller.

Die für diese Arbeit relevanten Teile des 82801 sind der Audio Codec und die DMA Controller, deswegen werden die wichtigsten Informationen zu ihrer Programmierung im Folgenden vorgestellt.

- **Features**

  Der Audio Codec funtioniert nach dem AC'97 Standard [6] und ist somit als externer Chip an den I/O Controller Hub angeschlossen. Der Codec fungiert als PCI Device und hat somit einen weitaus breiteren I/O Raum als vergleichbare ISA Hardware, was den Zugriff auf die mannigfaltigen Register deutlich erleichtert, da nun jedes Register direkt zugreifbar ist. Im Gegensatz zur SB16 muß auch keine manuelle Konfiguration mehr vorgenommen werden, da PCI Geräte vom BIOS automatisch konfiguriert werden.

  Dem Codec stehen drei DMA Controller des ICHs zur Verfügung, einer für PCM In, einer für PCM Out, einer für das Mikrophon, welche deutlich leistungsfähiger sind als die ISA DMA Controller, die von der SB16 verwendet werden müssen. Diese für den PCI Bus designten DMA Controller können nun bis zu 32 Puffer verwenden, dank 32 Bit Adressierung können diese Puffer nun auch an beliebiger Stelle im Speicher liegen.

  Alle ICHs der 82801 Baureihe können 16 Bit Stereo Sound, haben einen S/P DIF Anschluß und unterstützen bis zu zwei AC'97 Codecs (Audio und Modem Codec) bis zur Version 2.3 [7]. Ab ICH2 [8] wird 16 Bit 6-Kanalton, ab ICH4 [9] auch 20 Bit Sound für bis zu 6 Kanäle geboten. Ab letzterer Version gibt es auch zwei zusätzliche DMA-Controller für den S/P DIF Ausgang und für die Eingänge, einen

zweiten PCM Eingang und einen zweiten Mikrofoneingang. Zusätzlich wird nun auch ein dritter AC'97 Codec und *Memory Mapped I/O* unterstützt.

Register	Name	Register	Name
00h	Reset	**16h**	AUX In Volume
02h	Master Volume	**18h**	PCM Out Volume
04h	Headphone Volume	**1Ah**	Record Select
06h	Master Volume Mono	**1Ch**	Record Gain
08h	Master Tone (R & L)	**1Eh**	Record Gain Mic
0Ah	PC_BEEP Volume	**20h**	General Purpose
0Ch	Phone Volume	**22h**	3D Control
0Eh	Mic Volume	**26h**	Powerdown Ctrl/Stat
10h	Line In Volume	**2Ch**	PCM Front DAC Rate
12h	CD Volume	**32h**	PCM LR ADC Rate
14h	Video Volume	**34h**	Mic ADC Rate

Tabelle 8: AC'97 Mixer Register (Auswahl)

Register	Name	Zugriff
00h	PCM In Pufferliste Basisadresse	R/W
04h	PCM In Aktueller Puffer	RO
05h	PCM In Letzter gültiger Index	R/W
06h	PCM In Status	R/W
08h	PCM In Position in aktuellem Puffer	RO
0Ah	PCM In Prefetched Index	RO
0Bh	PCM In Kontrollregister	R/W
10h - 1Bh	PCM Out Bus Master Register	
20h - 2Bh	Mic In Bus Master Register	
2Ch	Globales Kontrollregister	R/W
30h	Globaler Status	RO
34h	Codec Write Semaphorenregister	R/W

Tabelle 9: i810 ICH Bus Master Register

- **Ports und Befehle**

  Für den AC'97 Codec werden zwei I/O Bereiche vorkonfiguriert: der Bereich für die Mixer (siehe Tabelle 8) und die Bus Mastering Register für die DMA Controller Ansteuerung (siehe Tabelle 9). Bei Nachfolgerversionen des 82801AA ICHs (ab ICH4) gibt es auch die Möglichkeit des *Memory Mapped I/O*, welches schnelleren Zugriff auf die Register bietet, als klassisches portorientiertes I/O. Außerdem kann die Hardware so einen größeren Addressraum anbieten. Abwärtskompatibilität ist jedoch gewährleistet, die alten I/O Bereiche bleiben bestehen. So kann auch ältere

Software erfolgreich mit einem neueren ICH umgehen, auch wenn sie dessen erweiterte Möglichkeiten dann natürlich nicht nutzen kann. Allerdings müssen die alten I/O Bereiche ab dem ICH5 explizit aktiviert werden.

Der Registersatz zur Ansteuerung eines DMA Controllers bietet Möglichkeiten, die weit über die der ISA DMA Controller hinausgehen. So gibt es nicht mehr nur einen Puffer, sondern bis zu 32, die hintereinander abgearbeitet werden. Den einzelnen Puffern können Attribute beigefügt werden (u.a. Interruptverhalten bei erfolgter Abarbeitung), die aktuelle Position kann abgefragt werden (Pufferindex + Position im aktuellen Puffer), und es existieren ein Status- und ein Kontrollregister, die umfangreiche Einstellungen ermöglichen und genauen Aufschluß über den momentanen Zustand einer Übertragung geben. Zusätzlich wird dem DMA-Controller der letzte vollständig gefüllte Puffer mitgeteilt, so dass dieser mit einem Interrupt und entsprechendem Setzen von Statusbits reagieren kann.

Die Behandlung von Interrupts gestaltet sich prinzipiell anders als bei der SB16. So muß erstens beachtet werden, daß PCI Interruptsharing erlaubt, d.h. es muß in den Statusregistern geprüft werden, ob das Interrupt überhaupt vom ICH ausgelöst wurde. Stammt das Interrupt vom ICH, muß zweitens geprüft werden, warum das Interrupt ausgelöst wurde. So kann ein Puffer abgearbeitet sein (ob hier ein Interrupt ausgelöst werden soll, ist für jeden Puffer einstellbar) oder auch der letzte gültige Puffer erreicht sein.

Die Einstellmöglichkeiten des AC'97 Mixers ähneln stark denen des SB16 Mixers, auch wenn ersterer im Detail mehr erlaubt (u.a. 16 Bit Mic, 20 Bit PCM, mehr Lautstärkeabstufungen). Die größten Unterschiede sind die Zugriffmethoden (Index- und Schreib/Leseregister vs. alle Register im I/O Raum) und die zusätzliche Powermanagementfunktionalität bei AC'97.

- **Initialisierung**

  Zuerst muß geprüft werden, ob die i810 + AC'97 Hardware überhaupt existiert. In modernen Betriebsystemen gestaltet sich dies ziemlich einfach, da diese Serviceroutinen besitzen, die Informationen über alle PCI Geräte herausgeben. Diese Geräteliste muß nun nur noch nach der entsprechenden Hersteller- und Geräte-ID durchsucht werden. Ist die passende Hardware gefunden, bietet der PCI Service noch zusätzliche Informationen über die I/O Bereiche der DMA Controller und des AC'97 Codecs und über das verwendete Interrupt.

  Ist der Treiber von der Existenz der Hardware überzeugt, wird diese nun neu initialisiert. Dazu wird entweder ein *Warm Reset* durchgeführt oder es wird, falls die Karte gerade einen *Cold Reset* hinter sich hat, beispielsweise direkt nach dem Einschalten des PC, bestätigt, dass der PC den normalen Betriebsmodus erreicht hat. Abschließend wird noch der *ACLink* (Verbindung zwischen ICH und AC'97 Codec) aktiviert und es wird kurz gewartet, bis sich der ICH erfolgreich neu gestartet hat.

  Nachdem nun der ICH erfolgreich aktiviert ist, kann geprüft werden, wieviele

Kanäle er unterstützt und welche Codecs an ihn angeschlossen sind. Dies kann über Flags im globalen Statusregister ($34h$) des ICH entschieden werden. Im Falle der Revisionen ICH4 und ICH5 muß über das SDM Register [9] getestet werden, welche ID der Codec hat, der gerade detektiert wurde, da hier Codec ID und Verbindungsnummer nicht übereinstimmen müssen. Die I/O Bereiche der einzelnen Codecs sind innerhalb des AC'97 I/O Bereichs um das Offset $80h$ gegeneinander verschoben.

Falls mehrere Codecs existieren, müssen die folgenden Schritte für jeden davon ausgeführt werden.

Der AC'97 Codec wird mittels Schreibens des Werts $00h$ an das Resetregister zurückgesetzt, anschließend können die internen Einheiten des Codecs mittels Löschen der entsprechenden Bits im Powerdown Register aktiviert werden. Sind diese aktiv, können die Fähigkeiten des Codecs abgefragt werden. Diese sind im Reset- und im Extended-ID-Register gespeichert (z.B. Anzahl unterstützer Kanäle, 3D Enhancement etc.). Nun kann auch der externe Verstärker über das Powerdown Register aktiviert werden.

Abschließend wird geprüft, welche Mixerein- und ausgänge existieren und in welcher Auflösung (5 oder 6 Bit) deren Lautstärke geregelt werden kann.

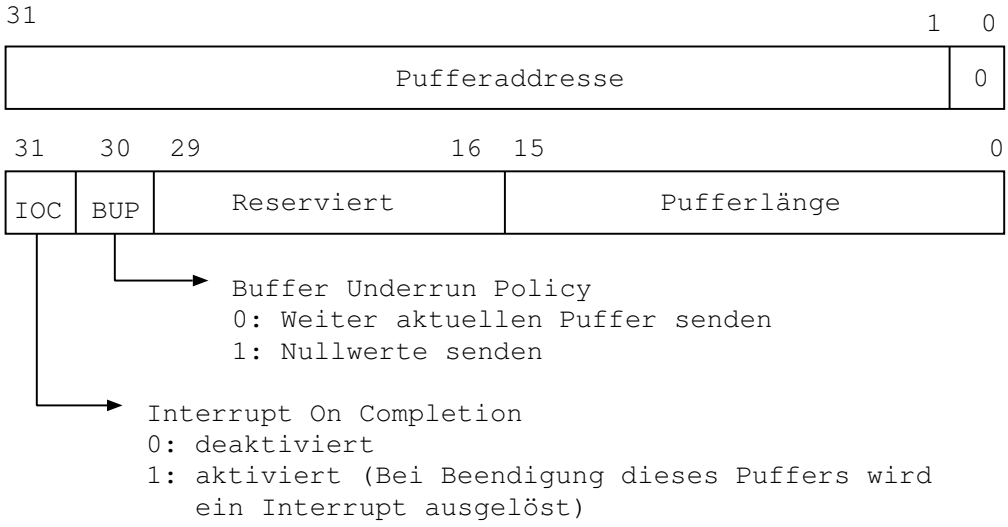

Abbildung 3: i810: Aufbau eines Pufferdeskriptors

- **Sound I/O** Der Akt der Audioausgabe findet programmiertechnisch fast ausschließlich auf dem ICH statt, im AC'97 Codec wird lediglich die gewünschte Sampling Rate über das PCM Front DAC Register gesetzt.

  Als erstes muß der DMA Puffer eingerichtet werden. Für kontinuierlichen Sound hat es sich als praktisch erwiesen einen zusammenhängenden Speicherbereich zu allozieren und die Pufferdeskriptoren (Abbildung 3) in der Pufferdeskriptorliste auf

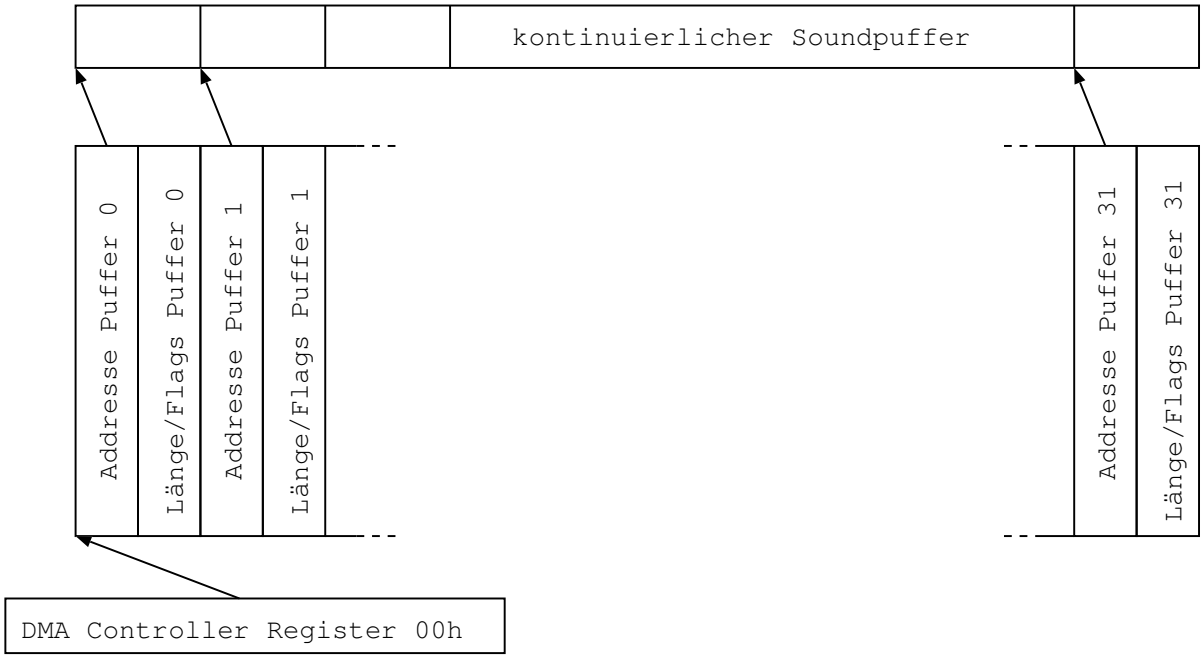

Abbildung 4: i810: Organisation der Pufferdeskriptorliste

die einzelnen Abschnitte des Gesamtpuffers zeigen zu lassen (siehe Abbildung 4). Mittels der Register des jeweiligen DMA Controllers kann so jederzeit festgestellt werden, welche Stelle des Puffers gerade abgearbeitet wird, so dass bekannt ist welche Bereiche Puffers der Treiber nun neu beschreiben bzw. auslesen kann.

Ist nun die Adresse der Pufferdeskriptorliste ins Register $00h$ des entsprechenden DMA Controllers geschrieben, muß nur noch der letzte gültige Puffer im Register $05h$ und der DMA Transfer selbst im Kontrollregister $0B$ aktiviert werden. Falls mehr Daten abgespielt werden sollen, als in den Puffer passen, so sollten auch die Interrupts im Kontrollregister aktiviert werden.

Tritt nun ein Interrupt auf, muß zuerst im globalen Statusregister geprüft werden, durch welchen DMA Controller das Interrupt ausgelöst wurde. Da PCI Interruptsharing erlaubt, kann es auch sein, dass das Interrupt gar nicht für den Soundtreiber bestimmt ist.

Ist die interruptauslösende Komponente bestimmt, wird der aktuelle Status des DMA Controllers bestimmt und entsprechend der Puffer neu befüllt bzw. weiter ausgelesen. Stehen keine Daten mehr zur Verfügung, wird der DMA Controller über sein Kontrollregister angehalten.

3 Realisierung

Dieser Teil nun beschreibt den praktischen Teil dieser Studienarbeit, die Entwicklung er Soundtreiber, des zentralen `SoundManager` Dienstes und des `javax.sound.sampled` Packages, mit welchem Kompatibilität zur J2SE 1.4 Spezifikation hergestellt wird.

Ausgehend von den gestellten Anforderungen wird im folgenden erst das Design entwickelt, welches das Soundsystem in überschaubare Einheiten und Schichten unterteilt. Danach wird auf die speziellen Probleme und Herausforderungen der Hardwaretreiber eingegangen.

3.1 Design

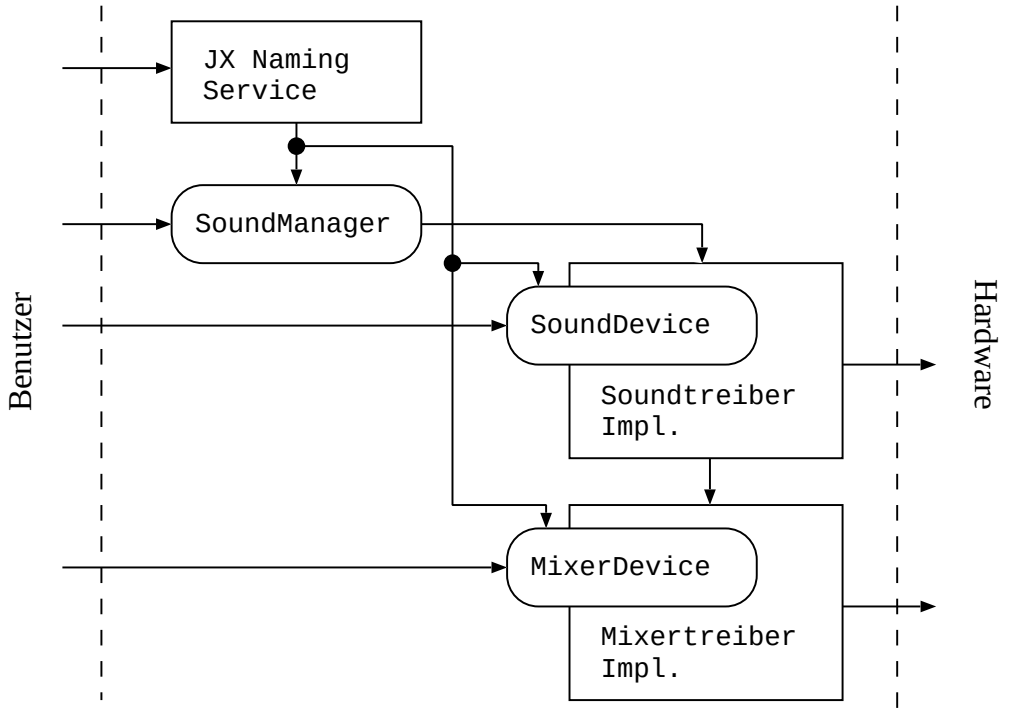

Abbildung 5: Schichtdesign der JX SoundEngine

Mit der konkreten Implementierung stellen sich einige Probleme, die nun gelöst werden müssen. So müssen die JX SoundEngine Klassen die gestellten Anforderungen erfüllen, sich den Hardwaregegebenheiten anpassen und außerdem die Möglichkeiten und Grenzen von JX beachten.

In Abbildung 5 sind die Zusammenhänge zwischen JX, Interfaces und den eigentlichen Treibern skizziert.

Das Gesamtbild der implementierten Klassen und Interfaces ist als UML-Diagramm in Abbildung 6 dargestellt.

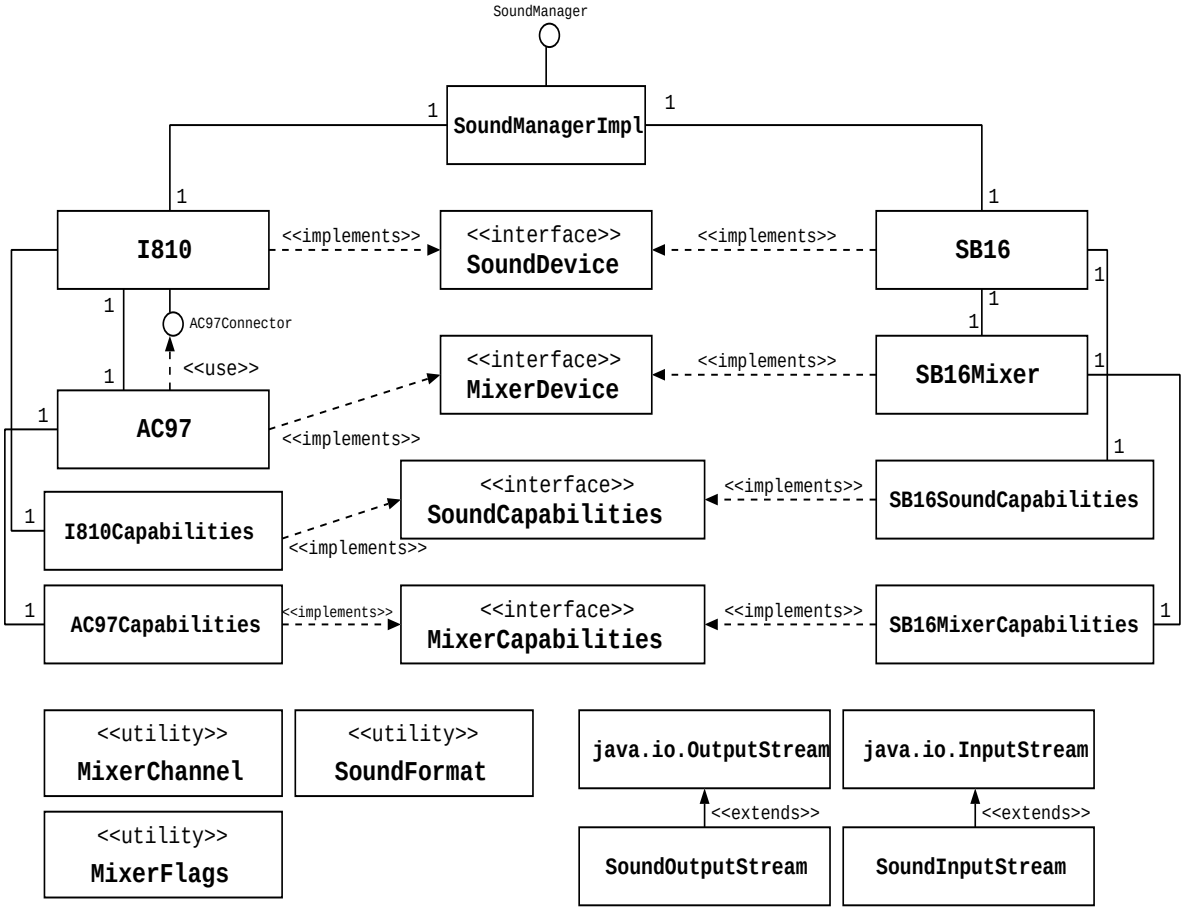

Abbildung 6: UML Klassendiagramm der JX SoundEngine

Die zentrale Klasse der JX SoundEngine ist `SoundManager`. Sie enthält eine statische `main` Methode und kann somit als eigenständige Komponente in JX betrieben werden. Jeder Komponente in JX können über die `boot.rc` Konfigurationsdatei Parameter übergeben werden. `SoundManager` nutzt diese, um sich Angaben über die vorhandene Soundhardware machen zu lassen.

Der `SoundManager` ist der zentrale Anlaufpunkt für den Benutzer, der Sound ausgeben will. Diese Klasse bietet Informationen über verfügbare Soundinterfaces und gibt auch Referenzen auf diese mittels der Methode `getSoundDevices()` zurück. Um bequem verfügbar zu sein, registriert sich der `SoundManager` beim zentralen JX Namingservice unter dem Namen “SoundManager”.

Nach ihrem Start legt diese Klasse als erstes eine Instanz der entsprechenden Hardwaretreiberklasse an und läßt so die Hardware testen und initialisieren. War dieser Schritt erfolgreich, speichert `SoundManager` eine Referenz auf das `SoundDevice` Interface des Hardwaretreibers.

Die Soundtreiberklasse wiederum erzeugt eine Instanz ihrer zugehörigen Mixertrei-

berklasse und speichert ein `MixerDevice` Interface auf diese Klasse.

Das Interface `SoundDevice` bietet eine abstrakte Sicht auf die Soundhardware und erlaubt momentan mittels der Methode `setProperties(..)` Eigenschaften, wie das verwendete Audioformat und die Samplerate zu setzen. Sound abgespielt wird entweder über `play(..)`, der ein fester Puffer übergeben wird, oder über `getPlayStream()`, die einen von `OutputStream` abgeleiteten Stream zurückgibt, mittels dem die Sounddaten byteweise an den Treiber übergeben werden können.

`MixerDevice` bietet über die Methode `setLevel(..)` die Möglichkeit die Lautstärke jedes Mixerkanals zu setzen. Zusätzliche Einstellmöglichkeiten, wie z.B. Mehrkanalsound, werden über die Methode `setFlags(..)` aktiviert.

Die genaue Dokumentation der JX SoundEngine API findet sich in Anhang A.

3.2 Codebeispiele

Im ersten Beispiel wird demonstriert, wie sich der Benutzer ein Sound- und ein Mixerinteface über den `SoundManager` holt:

```
1    Naming naming = InitialNaming.getInitialNaming();
2    SoundManager sm = (SoundManager)naming.lookup("SoundManager");
3    SoundDevice[] devs = sm.getSoundDevices();
4    if(devs.length == 0) {
5        Debug.out.println("Keine Sounddevices gefunden!");
6        return;
7    }
8    SoundDevice sd = devs[0];
9    MixerDevice md = sd.getMixerDevice();
```

Nachdem nun die Soundinterfaces verfügbar sind, werden die Eigenschaften der Soundhardware über die Capabilitiesobjekte abgefragt:

```
10   SoundCapabilities soundCap = sd.getCapabilities();
11   MixerCapabilities mixerCap = md.getCapabilities();
12   if( !soundCap.canDoFormats(SoundFormat.AFMT_S16_LE) ) {
13       Debug.out.println("16 Bit signed wird nicht unterstützt!");
14       return;
15   }
16   if( !mixerCap.hasChannels(MixerChannel.Volume) ||
17       !mixerCap.hasFlags(MixerFlags.STEREO) ) {
18       Debug.out.println("Mixer kann nicht genug!");
19       return;
20   }
```

Nun kann der Mixer konfiguriert und der Soundpuffer abgespielt werden. Dazu wird ein `SoundOutputStream` vom `SoundDevice` geholt, über welches die Bytes des Soundpuffers ausgegeben werden:

```
21   int maxlevel = mixerCap.getMaxLevel();
22   md.setLevel(MixerChannel.Volume, maxlevel/2, maxlevel/2);
23   md.setFlags(STEREO);
24   sd.setProperties(SoundFormat.AFMT_S16_LE, 44100);
25   SoundOutputStream soundOut = sd.getPlayStream();
26   for(int i = 0; i < soundBuf.length; i+=2)
27      soundOut.write(soundBuf, i, 2);
28   soundOut.flush();
```

Da das gewählte Soundformat 16 bittig ist, werden hier jeweils zwei Bytes auf einmal geschrieben, um zu gewährleisten, dass nur vollständige Samples beim Treiber ankommen.

3.3 Implementierung der Hardwaretreiber

Insgesamt wurden drei Soundtreiber im Rahmen dieser Studienarbeit implementiert, ein Treiber für die Sound Blaster 16, ein Treiber für i810-artige Chipsätze und ein Soundemulationtreiber für die Emulationsversion von JX. Während die beiden echten Hardwaretreiber komplett in Java geschrieben werden konnten, erforderte der Emulationstreiber einen komplett anderen Ansatz, da dieser teilweise im C-Kernel von JX residiert, um direkten Zugriff auf die Linuxsoundinterfaces zu haben.

Als Abstraktionsschicht besteht die Aufgabe des Treibers nicht nur darin, die jeweilige Hardware zu steuern und ein Benutzerinterface zu besitzen, sondern dieses auch über das jeweilige Betriebssystem tatsächlich verfügbar zu machen. Zusätzlich benutzt der Treiber diverse Dienste dieses Systems, seien es Hilfsroutinen oder Wrapper, die den konkreten Zugriff auf die Hardware vereinfachen oder erst ermöglichen.

Im Falle von JX als Java Betriebssystem sind beispielsweise alle Portzugriffe oder auch die IRQ Verwaltung im C-Kernel gekapselt, welcher diese Funktionalität über ein Javainterface dem Rest des Systems zur Verfügung stellt.

Im Folgenden werden nun alle für das JX SoundSystem relevanten JX Services kurz vorgestellt:

- `InitialNaming`: diese Klasse bildet die Wurzel des Servicesystems, sie ist als einzige statisch in jede Komponente gelinkt. Über sie können Interfaces zu allen registrierten Services abgerufen und neue Portale registriert werden.

- `Ports`: dieses Intefaces ermöglicht direkte Lese- und Schreibzugriffe auf die Ports.

- `IRQ`: hierdurch wird es Treiberklassen ermöglicht sich als Interrupthandler zu registrieren und einzelne Interrupts zu aktivieren und deaktivieren.

- `MemoryManager`: dieser Dienst ermöglicht es `Memory` Objekte zu allozieren, welche die Möglichkeit bieten, größere Datenmengen zwischen *Domains* auszutauschen, ohne den ganzen Speicherbereich kopieren zu müssen. Zusätzlich wurde im Rahmen dieser Studienarbeit dieser Service um die Möglichkeit erweitert, Speicher

unterhalb der 16 MB Grenze zu allozieren. Erst so ist es möglich, zuverlässig DMA Puffer zu reservieren, die von den ISA DMA Controllern auch verwendet werden können.

- `CPUManager`: hinter diesem Interface sind alle möglichen Subdienste versteckt, die mit Scheduling etc. zu tun haben. So ist es hier möglich, sich einen Mutex zu besorgen, der im Rahmen der Treiber dazu verwendet wird, den möglichen gleichzeitigen Zugriff zweier Programme auf den Treiber zu kontrollieren.

- `PCIAccess`: wie der Name schon nahelegt, verbirgt sich hinter diesem Interface die Möglichkeit die Konfiguration der PCI Geräte zu untersuchen und zu verändern. Dieser Service wurde nur für den i810 Treiber benötigt.

3.3.1 Soundblaster 16

Der Hardwaretreiber für die Soundblaster 16 Erweiterungskarte wurde im Zuge des praktischen Teils dieser Studienarbeit als erstes entwickelt. Aufgrund der Einfachkeit der SB16 Hardware eignete sie sich hervoragend als prototypischer Treiber, mit dem die Möglichkeiten von JX und die abstrakten JX SoundEngine Interfaces getestet werden konnten.

Es gab aber auch einen weiteren Grund. Eine SB16 existiert als emulierte Hardware sowohl in *VMWare Workstation* als auch in *bochs*, einem Open-Source PC-Emulationsprojekt.

Der Hardwaretreiber wurde in die Hauptklassen `SB16Finder`, `SB16` und `SB16Mixer` und in die Hilfsklassen `SB16MixerCapabilities` und `SB16SoundCapabilities` unterteilt. Da letztere lediglich über die sich nicht verändernden Fähigkeiten der SB16 Auskunft geben, enthalten sie nur Konstanten, die auf Nachfrage herausgegeben werden.

Die Klasse `SB16Finder` ist lediglich eine Wrapperklasse um die `open` Methode von `SB16`, da die SB16 Karte als ISA-Steckkarte nicht gesucht werden muß, es wird lediglich an der übergebenen Portadresse getestet, ob dort eine SB16 existiert. Um jedoch mit dem Treibermodell von JX (und der JX SoundEngine) konform zu gehen, ist eine solche Unterteilung durchaus sinnvoll.

Die Hauptklasse, die nahezu alle Arbeit erledigt, ist `SB16`. So wird in der `open` Methode getestet, ob die Karte existiert. Wenn dies zutrifft, wird sie nach Vorschrift initialisiert, die DSP Version wird abgefragt und der DMA Puffer wird initialisiert. Die Verwaltung des DMA Puffers ist die eigentliche Herausforderung, da selbst bei geschickter Programmierung der Hardware lediglich zwei Halbpuffer bereitstehen (siehe 2.4.1), die abwechselnd befüllt werden können. D.h. sobald die Hälfte des DMA Puffers abgearbeitet wurde, löst die SB16 ein Interrupt aus, welches dazu führt, dass die `interrupt` Methode von `SB16` aufgerufen wird. Diese Methode sorgt nun dafür, dass die gerade abgespielte Hälfte des Puffers mit weiteren Daten befüllt werden. Da die maximale Größe des Puffers mit 128 KB im Falle von 16 Bit Sound nicht sehr groß ist, muessen die abzuspielenden Daten anderweitig aufgehoben werden, beispielweise in einer Warteschlange. Auf diese Weise ist es auch möglich streamartig Daten zu akzeptieren. Im Falle von

SB16 existiert die Methode streamWrite, über die im Vergleich zum DMA Puffer kleine Datenblöcke übergeben werden können. Diese werden in einer FIFO-Liste gespeichert, so dass interrupt mittels dieser Daten den Puffer wieder auffüllen kann.

Die restlichen Methoden, die von SoundDevice geerbt wurden, sind entweder simple Verwaltungsfunktionen, die den internen Zustand von SB16 ändern oder Kommandofunktionen, die einen Befehl direkt an die Hardware weiterleiten. Um diese Registerzugriffe so einfach wie möglich so halten, existieren innerhalb des Treibers zwei private Methoden, die den bei der SB16 nötigen Zugriff auf Lese- bzw. Schreibstatusflags und die damit verbundenen Wartezeiten kapseln.

Die Klasse SB16Mixer erlaubt den Zugriff auf den Mixer der SB16. Da dieser lediglich aus Registern besteht, deren Wert jederzeit verändert werden kann, ist auch diese Klasse sehr einfach. Am unteren Ende stehen zwei Wrappermethoden, die den Lese- und Schreibzugriff auf ein einzelnes Register kapseln. Diese werden ihrerseits von den getLevel und setLevel Methoden aufgerufen, die sich ihrerseits lediglich die Übersetzung zwischen den Konstanten der MixerChannel Klasse und den korrespondierenden Mixerregistern übernehmen.

3.3.2 i810

Der Hardwaretreiber für i810-artige Soundhardware mit einem AC'97 Codec ist in die Klassen I810 und AC97 eingeteilt. Die Klasse I810 ist zuständig für die Initialisierung des ICH, für die DMA Controller und die Ein- und Ausgabe von Daten über die I/O Bereiche der Hardware. Sie implementiert das Interface AC97Connector, über welches die AC97 Klasse Informationen über den ICH abfragen kann und Daten in ein AC'97 Register schreiben oder aus einem lesen kann. Auf diese Weise kann die Klasse AC97 mit jedem Hardwaretreiber zusammenarbeiten, der das AC97Connector Interface implementiert. Somit wird abgebildet, dass der ICH für die I/O Bereiche zuständig ist und der AC'97 Codec ein untergeordneter externer Baustein ist.

Da jeder DMA Controller des ICH identisch funktioniert und sich lediglich über den verwendeten I/O Bereich von den anderen unterscheidet, wurde die Klasse DMAEngine als Subklasse von I810 angelegt. Diese enthält Kopien der wichtigsten DMA Register, um die Anzahl direkter Hardwarezugriffe zu minimieren, initialisiert den DMA Puffer und besitzt Methoden, um die Größe des freien Speicher im DMA Puffer zu erfragen, um neue Daten in den DMA Puffer zu schreiben und um die DMA Register an den neuen Zustand des DMA Puffers anzupassen.

Die Klasse AC97 ist dafür zuständig die Fähigkeiten des AC'97 Codecs zu prüfen, da hier die AC'97 Spezifikation recht breite Variationen zuläßt. Zusätzlich muß AC97 über den AC97Connector prüfen, inwiefern die Fähigkeiten des ICH zu denen des AC'97 Codecs kompatibel sind, beispielsweise kann die Anzahl der unterstützen Kanäle differieren.

Um die unterstützten lautstärkeregelbaren Mixerkanäle herauszubekommen, existiert die Subklasse MC, welche die Konstanten der Klasse MixerChannel auf die korrespondierenden AC'97 Register abbildet und testet, ob der jeweilige Mixerkanal existiert. Sollte der Kanal vorhanden sein, kümmert sich MC auch um die Lautstärkeeinstellungen.

3.3.3 Soundemulationstreiber

Neben der normalen JX Version, bei der der Kernel direkt auf der Hardware läuft, gibt es noch die Möglichkeit JX als OS Emulation zu kompilieren. In dieser Variante läuft JX als normaler Unixprozess, d.h. es kann auch nur die normalen Unix Systembefehle verwenden. Folglich müssen alle Teile von JX, die sonst direkt auf die Hardware zugreifen, in C neu geschrieben werden, da der Java Teil von JX nur den JX Kernel sieht, unabhängig von dessen Betriebsmodus.

Somit musste die untere Hälfte des Emulationstreibers ebenfalls in C geschrieben werden, um auf `/dev/dsp` und `/dev/mixer` zugreifen zu können. Dieser Low-Level Treiber macht nichts anderes, als Gerätedateien öffnen, ioctls auf diesen auszuführen, um die Soundparameter passend zu setzen, und `Memory` Objekte entgegenzunehmen, um deren Inhalt nach `/dev/dsp` zu schreiben. Insofern ist dieser Treiber lediglich ein JX-SoundEngine-nach-OSS/Free Mapper und die Tatsache, dass dies so einfach vonstatten geht, zeigt die starke Ähnlichkeit zwischen dem JX SoundEngine Ansatz und OSS/Free auf.

Der High-Level Teil dieses Treibers ist in Java geschrieben und besteht hauptsächlich aus zwei Wrapperklassen, die das `SoundDevice` und `MixerDevice` implementieren, und die übergebenen Befehle passend an den Low-Level Emulationstreiber weitergeben.

3.4 J2SE Interface

Die im Package `javax.sound.sampled` auftauchenden Klassen und Interfaces sind eine auf einem höheren Abstraktionslevel als die JX SoundEngine angesiedelte Soundarchitektur. Allerdings gibt im Design dieser beiden Ansätze auch Gemeinsamkeiten, so dass eine (teilweise) Neuimplementierung im Rahmen dieser Studienarbeit größtenteils unproblematisch ist. Im Folgenden wird nun auf die wichtigsten Gemeinsamkeiten und Unterschiede dieser beiden Ansätze eingegangen. Zusätzlich wird auch auf mögliche Synergieeffekte hingewiesen.

Die zentrale Verwaltung geschieht durch die Klasse `AudioSystem`, welche über eine Reihe von statischen Methoden Informationen über verfügbare Audiointerfaces und unterstützte Datei- und Soundformate gibt. Die verfügbaren Interfaces können über entsprechende Methoden auch angefordert werden. So gibt die Methode `getMixerInfo()` eine Liste von `Mixer.Info` Objekten zurück, welche Informationen über die vorhandenen Mixer liefern und dazu verwendet werden können über `getMixer(..)` das Interface zu einem konkreten Mixer anzufordern. Analog dienen die Methoden `getSourceLineInfo()` und `getTargetLineInfo()` dazu abzufragen, welche Soundeingabe und -ausgabe Objekte angefordert werden können. Diese von `Line` abgeleiteten Objekte exisitieren in unterschiedlicher Ausprägung. Sie können als `Clip` einen festen Soundpuffer besitzen oder als `SourceDataLine` einen Strom von Bytes abarbeiten.

Ein Mixer stellt eine Menge von `Line` Objekten zur Verfügung, über die Sound aufgenommen und abgespielt werden kann. Zusätzlich bietet ein Mixer die Möglichkeit Lines zu synchronisieren. Da ein Mixer ebenfalls von `Line` abgeleitet ist, bietet er alle Steue-

rungsmöglichkeiten, die von `Line` angeboten werden.

Jede `Line` bietet die Möglichkeit zu ihr gehörende `Control` Objekte zu erhalten. Diese gliedern sich in `FloatControl`, `BooleanControl`, `EnumControl` und `CompoundControl` auf und dienen als Regler für Lautstärke, Gain usw. dieser `Line`.

`AudioSystem` bietet noch weitere komfortable Möglichkeiten für den Benutzer. So lassen sich Sounddatein diverser Formate direkt öffnen und über einen `AudioInputStream` auslesen. Über letztere Klasse ist auch möglich Konversionen zwischen verschiedenen Soundformaten durchführen zu lassen.

Die Existenz einer zentralen Verwaltung läßt `javax.sound.sampled` und JX SoundEngine recht ähnlich erscheinen, aber während letztere ausschließlich Interfaces herausgibt, die direkt zu den Hardwaretreibern reichen, arbeiten die `javax.sound.sampled` Interfaces auf einem höheren Abstraktionslevel. Bei ihnen hat jede `Line` Zugriff auf höchsteigene Regler, `Line`s können flexibel mit Mixern und Datenquellen verbunden werden. Dies soll natürlich auch in der Implementierung berücksichtigt werden. Somit greifen die im Rahmen dieser Studienarbeit entwicklten Klassen des `javax.sound.sampled` Packages auf die JX SoundEngine zurück, führen aber einen zusätzlichen Abstraktionslevel ein, so daß beispielsweise über einen Softwaremixer mehrere Lines gleichzeitig Sound ausgeben können, auch wenn die Hardware dies von sich aus nicht unterstützt. In gewisser Weise sind diese Implementierungen also eine Vervollständigung der JX SoundEngine, letztere stellt den unkomplizierten Zugriff auf die Hardware sicher, erstere den Komfort und die Flexibilität.

4 Andere Soundsysteme

Nachdem nun die JX SoundEngine in allen Details vorgestellt worden ist, soll nun eruiert werden, wie es um ihre Konkurrenzfähigkeit im Vergleich mit anderen bereits existierenden Soundarchitekturen bestellt ist.

Da beide sehr bekannt und im breiten Einsatz sind, und zusätzlich deutlich unterschiedlichen Designprinzipien folgen, wird die JX SoundEngine im folgenden nun an dem *Open Sound System* [10] und der *Advanced Linux Sound Architecture* [11] gemessen. Besonderes Augenmerk wird dabei auf das Verhältnis von Benutzungskomplexität zu verfügbaren Möglichkeiten gelegt. Das heißt es wird sowohl positiv gewertet, wenn eine einfache Aufgabe, bespielsweise digitalen Sound in CD-Qualität ausgeben, einfach umgesetzt werden kann, als auch wenn alle Feaures der Soundhardware, unabhängig davon wie sie in ein Klassifikationschema passen, unterstützt werden.

4.1 Open Sound System (OSS)

OSS ist ein kommerzielles Treibersystem für fast alle Unixvarianten - sowohl kommerzielle als auch Open Source Entwicklungszweige - welches von 4Soft Technologies entwickelt und vertrieben wird. Unter Linux war seit Kernelversion 2.0 bis Version 2.4 die unter der GPL stehende Variante OSS/Free im Kernelquellcode enthalten, die bis 1997 unter

Federführung von 4Front Technologies entwickelt und danach von der Linux Community übernommen wurde.

Die kommerzielle Variante von OSS unterstützt deutlich mehr Soundhardware als die Open Source Variante, da sie jedoch das gleiche Benutzerinteface wie OSS/Free besitzt, ist dieser Unterschied für den Vergleich mit JX SoundEngine unerheblich.

4.1.1 Design

Die gesamte Funktionalität von OSS befindet sich im Kernel und wird über Gerätedateien angesprochen. Über spezielle `ioctl`-Aufrufe werden die Parameter der einzelnen Submodule gesetzt, wie z.B. die Sample Rate, oder die Lautstärke eines Mixerkanals.

OSS geht davon aus, dass es drei grundsätzlich unterschiedliche Soundgeräte gibt, die es in unterschiedlicher Ausprägung und Anordnung auf jeder Soundkarte gibt.

Da wäre zunächst das Gerät für *Digitized Voice* (auch DSP, Codec, PCM oder DAC/ADC Gerät genannt), welches digitalen Sound aufnehmen und abspielen kann.

Der nächste wichtige Baustein ist der *Mixer*, welcher die Lautstärke der Ein- und Ausgabekanäle und deren interne Zusammenschaltung regelt.

Der *Synthesizer* dient dazu Musik abzuspielen oder Soundeffekte zu erzeugen. OSS unterstützt momentan zwei Typen von Synthesizern. Zum einen werden FM-Synthesizer unterstützt, die vor allem noch in älteren Soundkarten zu finden sind. Zum anderen werden auch die moderneren Wavetable Synthesizer unterstützt, die sehr realistische Instrumentenklänge zu erzeugen vermögen. Zusätzlich besitzt OSS ein MIDI Interface, über das Befehle nach dem MIDI Standard gegeben und angeschlossene externe MIDI-Geräte gesteuert werden können.

OSS unterstützt lediglich die Features der Soundhardware, die auch tatsächlich von dieser angeboten werden. So wird im *OSS Programmer's Guide* [10] immer wieder darauf hingewiesen, dass man erst testen solle, ob etwas, das man benutzen möchte, auch tatsächlich existiert. Oder um den Leitfaden zu zitieren: "Not all mixers have a main volume control. For some reason almost all mixer programs written for the OSS API make this assumption". Dies gilt natürlich auch für andere Hardwarebestandteile, wie beispielsweise den DAC. So unterstützen Soundkarten (oder deren Treiber) möglicherweise nicht alle in OSS vorgesehenen Samplingraten, was dazu führen kann, dass der Benutzer sich selbst um ein Resampling seines digitalen Sounds kümmern muß. Ähnliches gilt für das Soundformat und die Anzahl der Soundkanäle (Mono, Stereo, etc.).

Aufgrund dieses Designs ist es relativ einfach, einen Hardwaretreiber zu schreiben, da dieser lediglich die Behandlung von Zugriffen auf die Gerätedateien OSS-konform gestalten muß und sich sonst nicht um Abstraktion kümmern muß. Letzlich muß sogar nur die Behandlung der `ioctl` Befehle gemäß dem OSS Standard sein, da der Benutzer sowieso prüfen muß, ob das von ihm gewünschte auch funktioniert.

Aber auf diese Weise wird die Abstraktionsschicht lediglich verschoben. Eine Applikation, die beispielsweise jedes mögliche PCM Soundformat abspielen können will, muß entweder einiges an Soundtransformationscode selbst mitbringen oder über externe Bibliotheken einbinden.

Nur in der kommerziellen Variante von OSS gibt es gegen Aufpreis einen *Virtual Mixer*, der das gleichzeitige Abspielen mehrere Quellen erlaubt und auch Soundformatumwandlungen in Software ausführt. Analog gibt es auch einen *Virtual Wavetable*, der einen qualitativ hochwertigen Wavetabel Synthesizer emuliert. Da jedoch im Rahmen dieser Studienarbeit ausschließlich Open Source Soundarchitekturen behandelt werden, fließen diese Erweiterungen von OSS nicht in die Bewertung mit ein.

4.1.2 Interface

Im folgenden werden nun die wichtigsten von OSS verwendeten Gerätedateien erläutert. Jede einzelne Gruppe von Gerätedateien repräsentiert dabei einen Baustein der Soundhardware.

- `/dev/dsp, /dev/audio`

  Über diese beiden Gerätedateien kann digitaler Sound ausgegeben und aufgenommen werden. Beide Geräte verhalten sich gleich, nur die Standardeinstellungen sind unterschiedlich. So ist `/dev/dsp` laut OSS Spezifikation auf 8 Bit unsigned Samples mit einer Samplingrate von 8 kHz eingestellt, während `/dev/audio` *mu law* codierte Samples erwartet. So soll eine minimale Kompatibilität mit der Sound API von Sun hergestellt werden.

  `/dev/dsp` ist dabei nur ein Link auf eine der "echten" Gerätedateien `/dev/dsp0`, `/dev/dsp1`, usw. Auf diese Weise können mehrere unabhängige DAC/ADC Bausteine betrieben werden.

- `/dev/mixer`

  Über diese Gerätedatei wird auf den Mixer zugegriffen. Auch hier ist diese Datei lediglich ein Link auf eines der Mixergeräte `/dev/mixer0`, `/dev/mixer1`, usw. (gewöhnlich `/dev/mixer0`). Auf diese Weise können mehrere Mixer betrieben werden, was beispielsweise bei der Intel 82801 Baureihe verwendet werden kann, da man an diesen ICH inzwischen bis zu drei Mixercodecs anschließen kann.

- `/dev/music, /dev/sequencer, /dev/midi`

  Ein weiterer wichtiger Teil der OSS Interfaces ist der Zugang zum Synthesizer der Soundhardware. Wie schon angesprochen findet sich heutzutage meistens ein Wavetable Synthesizer in den Soundkarten, während ältere PC Soundhardware meistens einen FM Synthesizer enthielt. Zusätzlich enthalten viele Soundkarten auch einen MIDI Port, an den externe MIDI Geräte angeschlossen werden können.

  Das Interface mit dem höchsten Abstraktionsniveau ist `/dev/music`. Es unterscheidet nicht zwischen internen Synthesizern und externen MIDI Geräten, auch die OSS Kommandos an diesen Port lehnen sich sehr stark an den MIDI Standard an. `/dev/sequencer` erlaubt einen direkteren Zugriff auf die internen Synthesizer, aber auch an externe MIDI Geräte können Befehle geschickt werden. Da jedoch die

API dieser Gerätedatei zwischen internen und externen Synthesizern unterscheidet, kann die Programmierung recht trickreich werden. Deswegen rät auch der *OSS Programmer's Guide* von der Verwendung dieses Interfaces ab. Am unteren Ende der Abstraktionsskala sitzt `/dev/midi`, welches den direkten Zugriff auf den MIDI Port erlaubt. Befehle, die hierüber gegeben werden, werden sofort an den MIDI Port weitergegeben, d.h. für das Timing ist die dieses Interface verwendende Software selbst verantwortlich. `/dev/midi` ist wiederum ein Link auf eine der echten Gerätedateien `/dev/midi00`, `/dev/midi01` usw.

Da ein derartiges Interface nicht in den Anforderungen an diese Studienarbeit beinhaltet war, wird es nicht für den Vergleich zwischen OSS und JX SoundEngine herangezogen.

4.1.3 Vergleich: JX SoundEngine vs. OSS

Auch wenn es auf den ersten Blick nicht so scheint, so sind sich die JX SoundEngine und OSS vom Design her sehr ähnlich. Zwar werden auf der einen Seite Java Interfaces verwendet, während OSS über Gerätedateien und `ioctl` Befehle angesprochen wird, jedoch sind sich die einzelnen Teilinterfaces in ihrer Funktionalität sehr ähnlich.

Sowohl `MixerDevice` als auch `/dev/mixer` werden dazu verwendet Informationen über die existierenden Mixerkanäle zu bekommen und um diese in ihrer Lautstärke zu regeln. Beide bieten also lediglich eine minimale Abstraktion, die dem Benutzer lediglich das Wissen über die konkrete Ansteuerung der Hardware erspart, aber nicht deren mögliche Einschränkungen.

Das gleiche gilt für `SoundDevice` und `/dev/dsp`. Auch hier muß der Benutzer zuerst erfragen, was die Hardware an Möglichkeiten bietet. Erst dann können die Soundparameter gesetzt und Samples zum Abspielen übergeben werden.

Somit kann festgehalten werden, dass sich die Interfaces der JX SoundEngine und der OSS Treiber trotz ihrer unterschiedlichen Ansprechweise sowohl in ihren Möglichkeiten, als auch in ihren Einschränkungen sehr stark ähneln.

Der größte Unterschied zwischen den beiden Architekturen besteht darin, wie der Benutzer an ein Interface herankommt. Während unter OSS der Benutzer eine Gerätedatei öffnet, für die sich eventuell ein Treiber registiert hat, der dann die Erledigung der verschiedenen Dateioperationen übernimmt, muß diese Aufgabe der Zuordnung von Hardware zu Interface unter JX der `SoundManager` übernehmen. Somit kann OSS alle Standardoperationen verwenden, die das VFS Layer unterstützt, während diese unter JX explizit in die Interfaces mit aufgenommen werden müssen.

Ein weiterer Unterschied betrifft die zugrunde liegende Sprache, in der die Treiber geschrieben sind. Da die JX SoundEngine in praktisch allen Teilen in Java geschrieben ist, können Fehlbedienungen oder Fehlerzustände per Exception mitgeteilt werden. Somit ist die Behandlung von Fehlern bereits durch die Semantik von Java gesichert und undefinierte Zustände von Hardware und Treiber können leicht vermieden werden. Unter OSS jedoch müssen die Rückgabewerte der Systemaufrufe explizit auf Fehlerzustände geprüft werden, da es keine andere Möglichkeit gibt, diese mitzuteilen.

4.2 Advanced Linux Sound Architecture (Alsa)

Das Alsa-Projekt begann 1998 mit der Arbeit einem neuartigen Soundsystem für Linux, welches OSS/Free ersetzen sollte. Mittels einer Aufteilung in Kerneltreiber, die nur exakt das unterstützen, was die Hardware kann, und einer Userlandbibliothek, die einen vereinfachten Zugriff auf die Kerneltreiber und Funktionalitätserweiterungen mittels Plugins bietet, sollten die Fähigkeiten moderner Soundhardware komplett ausgenutzt werden können. Da OSS/Free dies nicht könne [11], wird nun seit Kernelversion 2.6 Alsa auch im Mainstreamkernel angeboten.

Ein Unterschied zwischen Alsa und OSS, der sofort ins Auge sticht, hat die Ausführungen zu Alsa im Rahmen dieser Studienarbeit deutlich erschwert: zu Alsa existiert praktisch keine Dokumentation, ja nicht einmal eine Spezifikation. Die einzige existierende API Dokumentation wurde direkt aus dem Quellcode generiert und enthält oftmals Aussagen, dass sich ein bestimmter Funktionsprototyp jederzeit ändern könne. Aussagen über Vor- und Nachbedingungen, über Seiteneffekte und über die tatsächliche Bedeutung einer Funktion fehlen jedoch nahezu komplett.

Interessanterweise ist die Dokumentation, wie man einen Hardwaretreiber schreibt, weitaus ausführlicher und in Form eines Tutorials auch gut verständlich. Da jedoch in Abschnitt 4.2.3 die JX SoundEngine mit Alsa auch aus Benutzersicht verglichen werden soll, helfen diese Informationen nicht sehr viel weiter.

4.2.1 Design

Alsa ist in zwei getrennte Schichten unterteilt. Zum einen gibt es die Kerneltreiber, zum anderen die Alsa Bibliothek, die im Userspace residiert. Die Treiber im Kernel bilden lediglich die Fähigkeiten der Hardware nach außen ab, die Abstrahierung findet in der Benutzerbibliothek statt. Zusätzlich sind die Kerneltreiber so weit wie möglich modularisiert, so dass nur die Module für die tatsächlich gewünschten Audiofunktionen geladen werden müssen.

Dadurch, dass der größte Teil der Funktionalität außerhalb des Kernels angesiedelt ist, ist Alsa sehr flexibel. So ist das System über Konfigurationsdateien einstellbar, in denen auch die Plugins definiert werden. Diese Plugins ermöglichen beispielsweise die Konversion des Sounddatenstroms in Hinsicht auf Samplerate und Kanalzahl, das softwareseitige Mixen der Ein- und Ausgabekanäle, das "Abhören" des fertigen Soundsignals, das verlinkte Betreiben mehrere Soundkarten oder auch das "Verschmelzen" mehrere Karten zu einem virtuellen Gerät. Diese Liste an Plugins kann jederzeit erweitert werden. Letztlich ist im Alsa-Konzept jedes Soundausgabegerät, das über die Alsa Bibliothek angesprochen wird, ein Plugin, lediglich die geforderten Eigenschaften, die notfalls von der Bibliothek erledigt werden müssen, variieren.

Somit stehen dem Benutzer zwei Möglichkeiten offen Alsa zu benutzen. Zum einen kann er direkt auf die Kerneltreiber zugreifen, die, wie unter Unix üblich, über Gerätedateien angesprochen werden, oder er benutzt die komfortableren Möglichkeiten der Alsa Bibliothek. Da sich das Interface zwischen Kernel- und Userspacepart allerdings ohne

Vorankündigung ändern kann, und die Dokumentation der Alsa Bibliothek umfangreicher ist, ist letztere vorzuziehen.

4.2.2 Interface

Im folgenden wird nun kurz auf die Interfaces eingegangen, über die die Alsa Bibliothek mit den Alsa Kerneltreibern kommuniziert.

Über das Informationsinterface (`/proc/asound`) werden Informationen über die Soundhardware im System in menschenlesbarer Form ausgegeben.

Das Reglerinterface (`/dev/snd/controlCX`) erlaubt es, Regler, wie zum Beispiel die Lautstärkeregler des Mixers, zu bedienen.

Mittels des PCM Interfaces (`/dev/snd/pcmCXDX`) werden sowohl Einstellungen wie Soundformat etc. getätigt, als auch die einzelnen Samples mittels der Standarddateioperationen übergeben.

Das Raw MIDI Interface (`/dev/snd/midiCXDX`) erlaubt den direkten Zugriff auf MIDI Geräte und auf den MIDI Bus. Wie der Name schon andeutet, erfolgt dieser Zugriff über dieses Interface ohne jeglichen Komfort, wie beispielsweise vom Treiber gesteuertes Timing.

Das Sequencer Interface (`/dev/snd/seq`) hingegen bietet all dieses Features und erlaubt somit einen komfortablen Zugriff auf die MIDI Hardware.

Das Timer Interface (`/dev/snd/timer`) schließlich ist ein Hilfsinterface, welches exakte Zeitinformation liefert.

Da die nun folgenden API Funktionsgruppen in C geschrieben sind, liegen sie im gleichen Namespace und sind somit lediglich über differierende Präfixe gegliedert. Jedes dieser Interfaces besitzt eine `open` Funktion, die unter anderem einen Gerätenamen erwartet. Standardmäßig sind hier `hw` und `default` definiert, wobei ersteres vom Benutzer weitreichende Einstellungen erwartet, bevor es tatsächlich verwendet werden kann, während letzteres mit sinnvollen Standardeinstellungen aufwartet.

- Reglerinterface (`snd_ctl_*`)
  Mittels dieser Funktionen wird auf die in der Konfiguration definierten Regler zugegriffen. Zusätzlich bietet dieses Interface die Möglichkeit Callbacks zu registrieren, über die der Benutzer über Events, wie beispielsweise eine Änderung der Reglerkonfiguration, informiert wird.

- High Level Reglerinterface (`snd_hctl_*`)
  Über dieses Interface lassen sich erweiterte Regelungsmöglichkeiten, die auf dem Standardreglerinterface aufsetzen, definieren und nutzen. So ist es beispielweise möglich, mehrere Regler zusammenzufassen und über einen einzigen Funktionsaufruf einzustellen.

- PCM Interface (`snd_pcm_*`)
  Diese Interfaces dient dazu PCM-Streams auszugeben und aufzunehmen, und die Hardware- und Softwareseite von Alsa geeignet zu konfigurieren. Es ist hierbei

wichtig zu beachten, dass die Funktion `snd_pcm_open` einen virtuellen Soundkanal zurückliefert, d.h. es können bei geeigneter Soundgerätewahl beliebig viele Streams gleichzeitig offen sein, deren Sounddaten dann in einem Softwaremixer abgemischt werden. Um die Eigenschaften des Streams anzupassen existieren die Funktionsuntergruppen `snd_pcm_hw_*` und `snd_pcm_sw_*`. Nachdem diese Einstellungen per `snd_pcm_prepare` an die Bibliothek und somit letztlich an die Hardware weitergeleitet wurden, kann nun mittels `snd_pcm_write[in]` bzw. `snd_pcm_read[in]` Sound ausgegeben oder aufgenommen werden. Das Suffix `i` steht hierbei for *interleaved*, d.h. die Subsamples für die einzelnen kommen hintereinander in einem Sample, und das Suffix `n` steht für *non-interleaved*, d.h. die Subsamples für die einzelnen Kanäle sind in getrennten Bereichen des Soundpuffers. Neben diesen Grundfunktionen bietet das PCM Interface einige weitere hilfreiche Funktionen, beispielsweise das Synchronisieren mehrere PCM-Streams.

- RawMidi Interface (`snd_rawmidi_*`)
  Mittels dieser Funktonen ist es möglich direkt MIDI Befehle an die Soundhardware zu schicken. Zusätzlich gibt es noch Hilfsfunktionen zur Verwaltung des I/O Puffers. Da somit das gesamte Timing vom Benutzer erledigt werden muß, ist es in den meisten Fällen sinnvoller das Sequencer Interface zu benutzen.

- Timer Interface (`snd_timer_*`)
  Viele Soundkarten besitzen einen eigenen Zeitgeber, der eine weitaus feinere Auflösung als die Systemuhr bietet. Damit dies auch vom Benutzer genutzt werden kann, existiert das Timer Interface. Intern wird dieses Interface u.a. dazu genutzt PCM-Streams zu synchronisieren.

- Sequencer Interface (`snd_seq_*`)
  Einen weitaus komfortableren und flexibleren Zugang als das RawMidi Interface bietet das Sequencer Interface. Intern befasst sich es ausschließlich mit dem korrekten Zustellen von MIDI Messages, d.h. es beachtet Quelle und Ziel und den Zeitstempel. Eine gültige Quell- oder Zieladdresse besteht aus der Client ID und der Port ID. Ein Client ist entweder ein MIDI Gerät oder ein Benutzerprogramm. Jeder Client kann mehrere Ports haben. Messages können nicht nur direkt an Ports geschickt werden, sondern auch automatisch geroutet werden. Im einfachsten Fall ist das eine Verbindung zwischen einem Port einer Applikation und einem Port eines MIDI Gerätes, es können aber auch MIDI Messages zwischen Applikationen geroutet werden, um beispielsweise Filter implementieren zu können. Zusätzlich gibt es noch die Möglichkeit spezielle Sequencer Messages von Client zu Client zu schicken.

- Hardware abhängiges Interface (`snd_hwdep_*`)
  Dieses Interface existiert, um Spezialfeatures der Hardware ausnutzen zu können. Es wird dazu genutzt spezialle Software zu schreiben, die nur mit bestimmter Hardware funktioniert.

4.2.3 Vergleich: JX SoundEngine vs. Alsa

Wie auch im Vergleich zwischen OSS und der JX SoundEngine, wird in diesem Abschnitt lediglich das PCM Interface von Alsa in den Vergleich mit einbezogen. Da Alsa jedoch aus einem Kernel- und einem Userspaceteil besteht, ist ein eindeutiger Vergleich kompliziert, da die JX SoundEngine eher einem reinen Kernelteil gleicht, der erst in Verbindung eines High Level Packages, wie beispielsweise `javax.sound.sampled`, mit Alsa vergleichbar scheint. Somit werden im ersten Schritt die Teile verglichen, die minimalen Einsatz im Userspace erfordern, und im zweiten Schritt wird die gesamte PCM API der Alsa Bibliothek mit `javax.sound.sampled` verglichen.

Der Hauptunterschied, der sofort ins Auge sticht, ist die Möglichkeit bei Alsa mehrere PCM-Streams unabhängig von der Hardware zu erhalten. Dies ist mit den bisherigen JX Soundtreibern nicht möglich. Auch die Unterstützung sowohl des auch bei JX verwendeten *interleaved* PCM Formates als auch des *non-interleaved* Formates unterscheidet Alsa von der JX SoundEngine. Gleichwertig sind sich die Architekturen darin, dass ohne Einsatz im Userspace nur die tatsächlichen Fähigkeiten der Hardware genutzt werden können, welche vorher explizit abgefragt werden müssen.

Wird jedoch eine vollständige `javax.sound.sampled` Implementierung an die JX SoundEngine angeschlossen, dann gleichen sich deren Fähigkeiten stark an Alsa an und übertreffen diese auch in manchen Bereichen. So bieten beide die Möglichkeit zur Konversion des Soundformates, der Samplingrate und anderer Eigenschaften des Streams, beide bieten auch die Umleitung des Audiostreams in eine Datei oder durch benutzerdefinierbare Filter, beide bieten die Synchronisierung von Audiostreams. Während aber `javax.sound.sampled` hier, dadurch dass z.B. `AudioOutputStream` von `OutputStream` abgeleitet ist, sich die Möglichkeit offen hält, Streams von beliebigen Quellen zu belieben Zielen zu leiten, muß bei Alsa, da diese C verwendet, der Benutzer diese Datentransfers selbst verwalten. So ist es mit `javax.sound.sampled` trivial einen Audiostream über TCP/IP zu beziehen, während dies unter Alsa manuell erledigt werden muß. Auch bietet ersteres die Möglichkeit der automatischen Formaterkennung, so dass die Trivialaufgabe, einen vordefinierten Stream aus einer Datei abzuspielen hier tatsächlich trivial erledigt werden kann, während dies unter Alsa manuelle Arbeit erfordert. Somit ist es für den Benutzer nach Aussage eines Alsa Entwicklers (Takashi Iwai) bequemer, eine auf Alsa aufsetzende Bibliothek, wie beispielsweise *portaudio* oder `libao`, für Trivialaufgaben zu verwenden.

Somit kann abschließend subsummiert werden, dass die JX SoundEngine alleine gegenüber Alsa klar unterlegen ist, wird sie jedoch mit einer vollständigen `javax.sound.sampled` Implementierung kombiniert, zieht dieses Gespann gegenüber Alsa gleich und übertrifft diese auch in einigen Bereichen.

5 Zusammenfassung, Kritik und Ausblick

In diesem Kapitel wird noch einmal kurz zusammengefasst, welche verwert- und benutzbaren Ergebnisse diese Arbeit hervorgebracht hat. Außerdem wird darauf eingegangen welche systemimmanenten Probleme bei der Implementierung aufgetreten sind und welche Verbesserungen der gewählten Architektur noch möglich sind.

5.1 Ergebnisse

Das Ziel der Arbeit, eine Soundarchitektur unter JX zu haben, über die man PCM-Samples abspielen und aufnehmen kann, wurde erreicht. Natürlich kann der überwiegende Teil der existierenden Soundhardware deutlich mehr, z.B. Wavetable Synthese und die Verarbeitung von MIDI Befehlen, aber da diese Funktionalität unabhängig von den Sampleverarbeitungseinheiten ist, könnten die Hardwaretreiber unkompliziert erweitert werden, um diese Fähigkeiten zusätzlich zu unterstützen.

Schon bei der Entwicklung der abstrakten Interfaces wurde darauf geachtet, möglichst viele unterschiedliche Fähigkeiten der zugrundeliegenden Hardware unterstützen zu köninen. So können die Klassen `MixerChannel`, `SoundFormat` und `MixerFlags` jederzeit erweitert werden, wenn ein neuer Hardwaretreiber für Soundhardware implementiert wird, die Fähigkeiten besitzt, die noch nicht in diesen Klassen aufgelistet sind.

Selbstverständlich ist die aktuelle Implementierung nicht die beste aller möglichen. So ist beispielsweise die Behandlung paralleler Zugriffe auf die Soundhardware verbesserungswürdig. Zwar ist eine Synchronisation vorgesehen, diese führt aber dazu, dass nur ein Benutzer gleichzeitig die Soundhardware benutzen kann. Erschwerend kommt hinzu, dass die Synchronisation kaum getestet ist. Auch könnte das Schreiben von Treibern für sehr spezielle Soundhardware problematisch werden. Momentan wird angenommen, dass eine Soundkarte auch immer einen Mixer besitzt, dies muß aber nicht zwangsläufig zutreffen.

5.2 Probleme bei der Arbeit

Die Entwicklung fand unter Linux auf Intel x86 Rechner mit installiertem aktuellen JDK statt. Getestet wurde im JX Emulationsmodus, unter *VMWare Workstation* und auf einem PC mit einem i810 kompatiblen Chipsatz, der mittels Nullmodemkabel an den Entwicklungsrechner angeschlossen war.

Aus dieser Konstellation läßt sich schon ein Problem der Entwicklung einer JX Komponente ablesen. JX läßt sich momentan nur monolithisch bauen, so dass Entwicklung direkt unter JX nicht möglich ist. Das hat auch zur Folge, dass die Konfiguration einer kompletten Entwicklungs- und Testumgebung sehr aufwändig ist. So muß der Testrechner den JX Kernel und den Modulcode über das Netzwerk laden können, die Debugausgabe über die serielle Schnittstelle des Testrechners muß auf dem Entwicklungsrechner abgefangen werden und die Entwicklungsumgebung muß an die lokalen Gegebenheiten

angepasst werden. Diese Entwicklungsmethode hat zur Folge, dass die Turnaroundzeiten sehr hoch sind und auch klassisches Einzelschrittdebugging nicht möglich ist.

Das größte Problem bei der Implementierung der JX SoundEngine war die Tatsache, dass sich JX immer noch in einer sehr frühen Entwicklungsphase befindet. Daher wurden im Laufe dieser Arbeit einige Fehler in allen möglichen JX Komponenten entdeckt, die natürlich erst einmal zeitaufwändig als Fehler identifiziert werden mussten. Erschwerend kam hinzu, dass der Bytecodecompiler bei weitem nicht alle Features von Java unterstützt, so dass teilweise das Testen von Code nicht möglich war. Auch die Implementierung der J2SE API weist noch deutliche Lücken und auch ungetesteten und somit fehlerbehafteten Code auf, so dass das vorgesehene Demoprogramm immer mehr zusammenschrumpfte, bis es auf einen nichtinteraktiven Teil beschränkt war, der lediglich die JX SoundEngine und die Debugausgabe verwendet.

Bedingt durch die frühe Entwicklungsphase ist auch die Dokumentation zu JX sehr lichte, so dass immer wieder intensive Beratungsgespräche mit den Betreuern nötig wurden, die oft genug damit endeten, dass langwierig nach einem Fehler in JX gesucht werden musste, der auf einer der vielen Ebenen lauerte (Bytecodecompiler, JX Kernel, J2SE API Implementation).

Aber auch unabhängig von JX gab es das Problem ungenügender Dokumentation. Um die JX SoundEngine mit anderen Soundarchitekturen vergleichen zu können, müssen diese anderen Soundarchitekturen erst einmal verstanden werden, was nunmal ohne Dokumentation sehr aufwändig ist. So war die Evaluierung von OSS sehr einfach, die diese Architektur vorbildlich dokumentiert ist, während das nun unter Linux zum Standard erklärte Alsa System unter massiven Dokumentationsdefiziten leidet. Dies hat die Evaluierung deutlich kompliziert.

Somit lagen die Schwierigkeiten dieser Arbeit ganz anders, als vom Bearbeiter zu Anfang vermutet wurde. Nicht fehlende Dokumentation der Hardware war das Problem, sondern fehlende Dokumentation der Software.

5.3 Mögliche Erweiterungen

Im Folgenden werde ich auf einfach realisierbare Erweiterungen der JX SoundEngine eingehen.

Der größte Nachteil der JX SoundEngine ist, dass die Soundinterfaces zwar abstrakt sind, man jedoch nur die tatsächlich vorhandenen Möglichkeiten der Hardware nutzen kann. Besonders fällt das auf, wenn mehrere Nutzer gleichzeitig Sound abspielen wollen.

Dieses Problem läßt sich am einfachsten dadurch lösen, dass man einen Softmixer einführt, der beliebig viele PCM-Streams zu einem einzigen Stream zusammenführen kann. Der Hardwaretreiber hat nun pro DMA Controller einen echten Stream, auf die er die virtuellen Streams gleichmäßig aufteilt.

Weiterhin wäre es wohl auch für zukünftige Erweiterungen einfacher, wenn das `SoundDevice` Interface ledglich allgemeine Informationen über die Soundhardware und Interfaces auf deren spezielle Funktionen herausgeben würde. In diesem Szenario würden alle Funktionen aus `SoundDevice`, die sich konkret mit PCM-Sound beschäftigen, beispiels-

weise in ein Interface namens `PCMSound` umziehen. Die MIDI Funktionen einer Soundkarte wären dann über das Interface `MIDI` erreichbar, welches ebenfalls über `SoundDevice` zu beziehen wäre.

Vervollständigt man zusätzlich die Implementierung des `javax.sound.sampled` Packages, hat man eine moderne und flexible Soundarchitektur, mit der einfache Aufgaben auf einfache Weise erledigt werden können, aber auch komplexe Soundverarbeitungsprozesse möglich sind.

A JX SoundEngine API

Um möglichst direkt die Hardware bedienen zu können, stellt die JX SoundEngine Interfaces bereit, die teilweise direkt durch die Hardwaretreiber implementiert wurden. Im Folgenden werden nun diese Interfaces und alle für den Endnutzer relevanten Methoden vorgestellt.

SoundManager

- `SoundDevice[] getSoundDevices()`
  Über diese Methode des zentralen `SoundManager` Dienstes werden alle registrierten `SoundDevice` Interfaces zurückgeliefert. Eine Anwendung, die unter JX über die hauseigenen Interfaces Sound ausgeben will, kommt somit ohne diese Methode nicht aus.

SoundDevice

- `MixerDevice getMixerDevice()`
  Diese Methode gibt das zu diesem `SoundDevice` gehörende `MixerDevice` zurück.

- `SoundCapabilities getCapabilities()`
  Über die von dieser Methode zurückgegebenen Klasse können die Fähigkeiten der Hardware abgefragt werden, so dass sich der Benutzer darauf einstellen kann.

- `String getName()`
  Liefert den Namen der verwendeten Soundhardware zurück.

- `void setProperties(int samplefmt, int samplerate)`
  Mittels dieser Methode werden die Eigenschaften des Sounddatenstroms an den Soundtreiber übermittelt. Das Datenformat wird mittels der Konstanten der Klasse `SoundFormat` übergeben, die Samplerate ist ein Integerwert deren Wertebereich von der zugrundeliegenden Hardware abhängt.

- `int[] getProperties()`
  Diese Methode liefert ein Array aus zwei Integern zurück, dessen erstes Element das aktuell eingestellte Soundformat und dessen zweites die Samplerate ist.

- `void play(Memory src, int length)`
  Um einen Puffer fester Länge mit Samples abzuspielen, wird diese Methode verwendet. Sie kehrt sofort zurück, und der Soundpuffer wird im Hintergrund mit den aktuell eingestellten Soundeigenschaften abgespielt.

- `void pause()`, `void cont()`, `void stop()`
  Diese Kontrollbefehle erlauben es, den aktuellen Abspielvorgang zu pausieren, wieder fortzusetzen, oder abzubrechen.

- `Memory record(int length)`
  Analog zu `play` wird hier ein voreingestellte Menge an Samples aufgenommen und als `Memory` Objekt zurückgeliefert.

- `SoundOutputStream getPlayStream()` Liefert einen `SoundOutputStream` zurück, über den ein Stream von Samples ausgegeben werden kann.

- `SoundInputStream getRecordStream()` Liefert einen `SoundInputStream` zurück, über den ein Stream von Samples aufgenommen werden kann.

MixerDevices

- `MixerCapabilities getCapabilities()`
  Diese Methode liefert eine Instanz der Klasse `MixerCapabilities` zurück, über die die Fähigkeiten der zugrundeliegenden Mixerhardware abgefragt werden können.

- `void reset()`
  Setzt die Einstellungen des Mixers auf die Standardeinstellungen zurück.

- `int[] getLevel(int mixchannel)`
  Liefert die aktuellen Lautstärkeeinstellungen des Mixerkanals mixchannel zurück. Dieser wird in Konstanten der Klasse `MixerChannel` ausgedrückt. Das erste Element des zurückgelieferten Arrays enthält die Lautstärke des linken Teilkanals, das zweite Element die des rechten Teilkanals.

- `void setLevel(int mixchannel, int left, int right)`
  Setzt die Lautstärke des Kanals mixchannel auf [left, right].

- `void setFlags(int mixerflags)`
  Mittels dieser Funktion können speziell Funktionen der Mixerhardware aktiviert werden, wie zum Beispiel Mehrkanalsound, 3D Enhancement und mehr. Die Konstanten hierzu sind in `MixerFlags` definiert.

- `int getFlags()`
  Liefert die aktuell eingestellten Mixerflags zurück.

SoundCapabilities

- `boolean canDoFormats(int formats)`
  Dieser Methode werden geoderte Konstanten aus `SoundFormat` übergeben. Der Rückgabewert sagt aus, ob die Soundhardware diese Formate unterstützt oder nicht.

- `int getMinSampleRate()`
  Gibt die minimale von der Soundhardware unterstützte Samplingrate zurück.

- `int getMaxSampleRate()`
  Gibt die maximale Samplingrate zurück, die die Soundhardware unterstützt.

SoundFormat

Diese Klasse enthält die Konstanten, mittels deren das Format des Sounddatenstroms an `SoundDevice` übermittelt werden.

- `AFMT_S16_LE`: 16 Bit signed Little Endian
- `AFMT_S18_LE`: 18 Bit signed Little Endian
- `AFMT_S20_LE`: 20 Bit signed Little Endian

MixerCapabilities

- `boolean hasChannels(int channels)`
  Dieser Methode werden geoderte Konstanten aus `MixerChannel` übergeben. Der Rückgabewert sagt aus, ob diese Kanäle im Mixer existieren.
- `boolean hasFlags(int mixerflags)`
  Mittels dieser Methode wird abgefragt, ob ein Mixer bestimmte zusätzliche Features unterstützt. Dazu werden ihr geoderte Konstanten aus `MixerFlags` übergeben. Der Rückgabewert drückt dann aus, ob diese Features unterstützt werden, oder nicht.
- `int getMaxLevel()`
  Diese Methode liefert den maximalen Lautstärkewert zurück, der in `setLevel` und `getLevel` in `MixerDevice` verwendet werden kann. Dieser Wert hängt davon ab, welche Anzahl an Bits der Mixer für die Lautstärkeeinstellungen verwendet.

MixerFlags

In dieser Klasse sind die Konstanten definiert, die bei den Methoden `getFlags` und `setFlags` des Interfaces `MixerDevice` verwendet werden.

- `STEREO`
- `FOUR_CHANNELS`
- `SIX_CHANNELS`

MixerChannel

Jede der Konstanten dieser Klasse definiert einen Kanal des Mixers. Diese sind für die Verwendung bei `setLevel` und `getLevel` von `MixerDevice` gedacht.

- `Volume`: die Gesamtlautstärke
- `Voice`: der DAC/PCM Ausgang
- `Midi`: der Midi Ausgangs
- `CD`: der CD Ausgangs
- `LineIn`: der Line Eingangs
- `Mic`: der Mikrophoneingang
- `Speaker`: der Systemlautsprecher
- `SPDIF`: der digitale S/PDIF Ausgang
- `Surround`: die Surroundkanäle
- `Center_LFE`: die Surroundkanäle für 6 Kanal Sound

SoundOutputStream

Diese Klasse ist von `OutputStream` abgeleitet und bietet all deren wohldokumentierten Methoden, um bequem Daten an die Soundhardware schicken zu können.

SoundInputStream

Analog zu `SoundOutputStream` ist diese Klasse von `InputStream` abgeleitet und bietet somit einen bequemen Weg, Soundsamples aufzunehmen.

SoundException

Dies ist die Standardexception, die geworfen wird, wenn innerhalb der Treiber Fehler auftreten, oder auch wenn die Hardware nicht dazu fähig ist das zu tun, was der Benutzer von ihr verlangt.

Literatur

[1] *Sound Synthesis and Sampling*, Martin Russ, Focal Press, 1997

[2] *Java 2 Platform, Standard Edition, v1.4.2 API Specification*, Sun Microsystems, Inc., `http://java.sun.com/j2se/1.4.2/docs/api/`

[3] *Complete MIDI 1.0 Detailed Specification*, MIDI Manufacturers Association, 2001, `http://www.midi.org`

[4] *The JX Operating System*, Michael Golm, Meik Felser, Christian Wawersich, Jürgen Kleinöder, 2002 USENIX Annual Technical Conference

[5] *Developer Kit for Sound Blaster Series (Second Edition)*, Creative Technology Ltd., 1996

[6] *Audio Codec '97 Revision 2.3*, Intel Cooperation, 2002

[7] *Intel 82801AA (ICH) and Intel 82801AB (ICH0) I/O Controller Hub*, Intel Cooperation, 1999

[8] *Intel 82801BA I/O Controller Hub (ICH2), AC '97*, Intel Cooperation, 2000

[9] *Intel 82801EB (ICH5) I/O, 82801ER (ICH5R), and 82801DB (ICH4) Controller Hub: AC'97 Programmers Reference Manual*, Intel Cooperation, 2003

[10] *Open Sound System Programmers's Guide Version 1.11*, 4Front Technologies, 2000

[11] *Advanced Linux Sound Architecture*, `http://www.alsa-project.org`