Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Alexander Sandau

ARM-cortex-basiertes Plugin-System für MIDI-Synthesizer

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Sandau

ARM-cortex-basiertes Plugin-System für MIDI-Synthesizer

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Meisel
Zweitgutachter: Prof. Dr. Fohl

Eingereicht am: 20. Februar 2014

Alexander Sandau

Thema der Arbeit

ARM-cortex-basiertes Plugin-System für MIDI-Synthesizer

Stichworte

Mikroprozessor, MIDIbox, Plug-in, Audioeffekte, Synthesizer, Signalverarbeitung, ARM

Kurzzusammenfassung

Um einen Klangerzeuger oder Audioeffekt auf einem Microprozessor wie dem ARM-cortex-m3 zu implementieren wird Detailwissen über die Hardware benötigt. Diese Arbeit soll es ermöglichen Klangerzeuger und Audioeffekte zu implementieren, ohne ein Vorwissen von der Hardware zu haben. Dazu wurden einfache Schnittstellen dafür entwickelt, die es ermöglichen einen Klangerzeuger und mehrere Audioeffekte gleichzeitig auf einem ARM-cortex-Mikrocontroller laufen zu lassen. In dieser Arbeit werden die Hard- und Software die als Grundlage dienen, sowie das Framework und die Schnittstellen die entwickelt wurden, vorgestellt.

Alexander Sandau

Title of the paper

ARM-cortex-based Plugin-System for MIDI-Synthesizers

Keywords

microprocessor, MIDIbox, plug-in, audio-effects, synthesizer, signal-processing, ARM

Abstract

Impementing a synthesizer or audio-effect on a microprocessor such as the ARM-cortex-m3 requires detailed knowledge about the hardware. This thesis enables the implementation of sythesizers and audio-effects without having prior knowledge of the hardware. Therefore simple interfaces have been developed, making it possible to run a synthesizer and several audio-effects simlutaniously on an ARM-cortex-microporcessor. In this thesis the hardware and software that serve as a basis, as well as the framework and the interfaces that were developed will be presented.

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Aufbau 2

2. Grundlagen **3**

2.1. MIDI 3

2.2. Audioverarbeitung 6

2.2.1. Klangsynthese 7

2.2.2. Audioeffekte 15

2.3. MIDIbox 17

2.3.1. MIOS 19

3. Konzept **20**

3.1. Überblick 21

3.2. Main-Framework 21

3.3. Input-Manager 24

3.4. CC-Map 27

3.5. Plugin-Interfaces 28

3.5.1. Wavetable-Synthesizer 30

3.5.2. Modulations-effekt 33

3.5.3. Faltungshall 34

4. Implementierung **36**

4.1. Fremde Bibliotheken 36

4.2. Platformabhängige Entscheidungen 37

4.3. Input-manager 38

4.4. CC-map 40

4.5. Wavetable-Synthesizer und Modulationseffekt 40

5. Zusammenfassung und Ausblick **43**

A. Anleitung zur Entwicklung von Plugins **44**

B. Glossar **49**

Abbildungsverzeichnis

1.1. Aufbau des Systems 1
1.2. Hardware 2

2.1. Beispielhafte Verbindung von MIDI-Kompatiblen Systemen 3
2.2. Aufbau der Bytes der MIDI-Nachrichten 4
2.3. variierende Wellenlänge, je nach gespielter Note. Verschiedene Wellenformen. 7
2.4. Subtraktive Synthese, vereinfachter Aufbau 8
2.5. simple Wellenformen, und ihre Frequenzspektren 8
2.6. Tiefpass Filter, Bandpass Filter, Hochpass Filter 9
2.7. verschiedene Arten von Hüllkurven im Vergleich 10
2.8. Wavetable Synthesizer mit linearer Interpolation 12
2.9. oben: linearer Waveshaper, verändert Signal nicht.
unten: durch Waveshaper verzerrtes Signal 13
2.10. verschiedene Interpolationsarten 14
2.11. Gewichtung eines digitalen Audiosignals mit einem Faktor 15
2.12. Modulation eines digitalen Audiosignals mit einem Modulator 15
2.13. Faltung eines digitalen Audiosignals mit einem Faltungskern 16
2.14. Blockschaltbild der Hardware 18
2.15. MIOS-Studio. GUI zum Hochladen und Testen von Applikationen 19

3.1. Komponentendiagram des Plugin-Systems 20
3.2. Voice-stealing 25
3.3. ADSR-Hüllkurven-Statemachine 26
3.4. Arbeitsweise der CC-map 27
3.5. Wavetable-Datei 30
3.6. Beispiel für die Schrittweiten und Interpolation bei einer Wavetable in x-Richtung 31
3.7. Beispiel für die Schrittweiten und Interpolation bei einer Wavetable in y-Richtung 31
3.8. Wavetable-strategie 32
3.9. Ringmodulator und Tremolo 33
3.10. Arrays-längen des Faltungsalgorithmus 35

4.1. Aufbau der Notenliste im Input-Manager 39
4.2. Anpassung der Hüllkurve bei monophonen Synthesizern 39

1. Einleitung

Im Rahmen dieser Bachelorarbeit soll ein Grundgerüst für digitale Audioverarbeitung auf einer ARM-cortex-Plattform realisiert werden. Dazu wird ein Plugin-Konzept entwickelt und implementiert, welches es ermöglicht Synthesizer und Effekte auf dieser Plattform laufen zu lassen. Zusätzlich werden demonstrativ ein Wavetable-Synthesizer und ein Modulationseffekt, welche auf Basis dieses Plugin-Konzepts entwickelt wurden, vorgestellt.

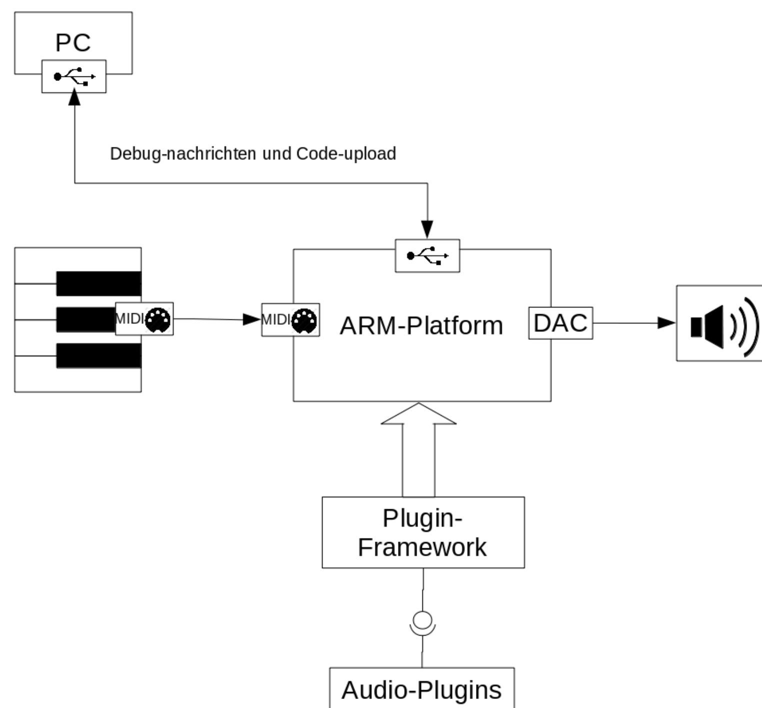

Abbildung 1.1.: Aufbau des Systems

Das Ziel der Arbeit ist es eine einfach nutzbare Schnittstelle für Audioverarbeitungs-Plugins auf einer low-cost-Hardware zu entwickeln. Dieses Plugin-Konzept soll in einem zukünftigem Hochschulprojekt genutzt werden. Die dafür entwickelten Beispiel-Plugins haben nicht den

Anspruch professionelle Synthesizer oder Audioeffekte zu sein, sondern Sie sollen die Nutzbarkeit der Schnittstelle zeigen.

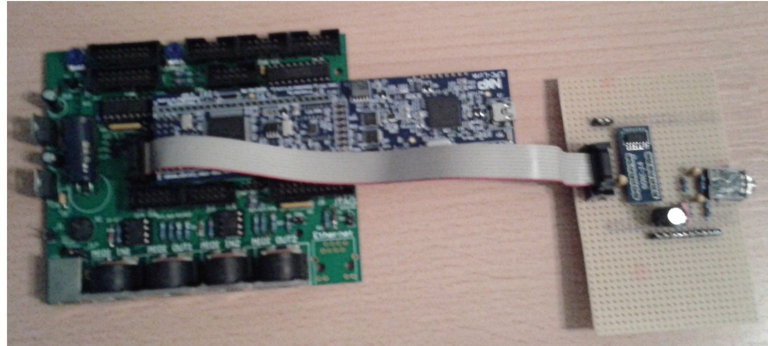

Abbildung 1.2.: Hardware

Es wird ein Microcontroller benötigt welcher auf einem ARM-cortex-Prozessor basiert. Zusätzlich werden MIDI-Schnittstellen für eingehende Steuerdaten und eine Audio-Ausgabe benötigt. Es wird der Microcontroller LPC1769 genutzt, welcher einen ARM-cortex m3 besitzt. Die MIDI-Schnittstelle wird mit UART betrieben, und die Audio-Ausgabe wird durch den DAC PCM1725 realisiert.

1.1. Aufbau

Im ersten Kapitel gibt es eine Einführung in das Thema und Ziel der Arbeit. Im zweiten Kapitel werden die Grundlagen beschrieben, die nötig sind um die Arbeit zu verstehen. Im dritten Kapitel wird beschrieben wie das System aufgebaut ist und die Aufgaben der einzelnen Komponenten. Im vierten Kapitel wird die Implementierung vorgestellt und beschreibt wie die Funktionalitäten realisiert werden. Das letzte Kapitel schließt die Arbeit mit einer Zusammenfassung und einem Ausblick ab.

2. Grundlagen

In diesem Kapitel wird ein Überblick über die für diese Bachelorarbeit relevanten Themen gegeben. Es werden eingesetzte Technologien beschrieben und Begriffe erklärt.

2.1. MIDI

MIDI ist ein Standard zum Austausch von Steuerinformationen für elektronische Instrumente. Vor der Entwicklung von MIDI wurden meist analoge Spannungen genutzt, um verschiedene Soundmodule zu verbinden. Die Schnittstellen waren uneinheitlich, und die Verbindung mehrerer Geräte war mit großem Verkabelungsaufwand verbunden. Mit der Entwicklung von MIDI wurde ein einheitlicher Standard für den Austausch von Audioinformationen geschaffen, der eine einfache Verbindung mehrerer Geräte ermöglicht. Es ist ein Hardwareinterface (MIDI-Schnittstelle) und ein Protokoll spezifiziert. Mit diesem Protokoll können zwei oder mehr Geräte über die MIDI-Schnittstelle kommunizieren. Es ist auch möglich andere Übertragungswege zu nutzen. Möglich sind zum Beispiel USB, Firewire und sogar Wireless-MIDI.

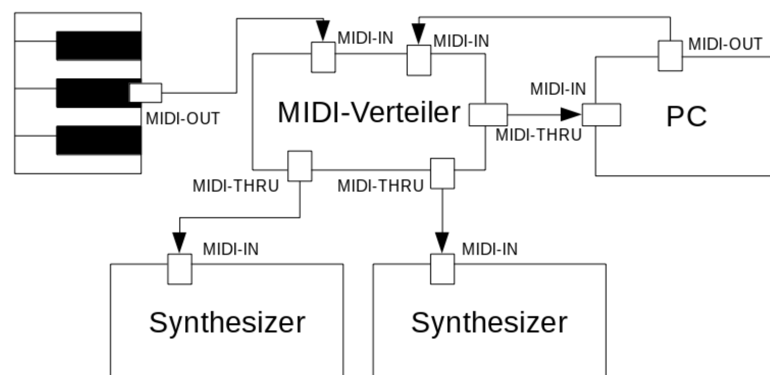

Abbildung 2.1.: Beispielhafte Verbindung von MIDI-Kompatiblen Systemen

Masterkeyboards sind beispielsweise Erzeuger von MIDI-Nachrichten. Es gibt aber auch andere Arten wie reine MIDI-Controller, die nur Drehregler und Knöpfe besitzen oder auch alternative

Notenerzeuger wie MIDI-Gitarren oder MIDI-Schlagzeug. Letztere sind aber nicht weitverbreitet, da das MIDI-Protokoll ursprünglich für Keyboards ausgelegt wurde. Synthesizer können diese Nachrichten auswerten und erzeugen daraus kontinuierliche Audiodaten. Synthesizer die kein eigenes Keyboard besitzen und ihre Befehle extern über MIDI beziehen heißen Expander.

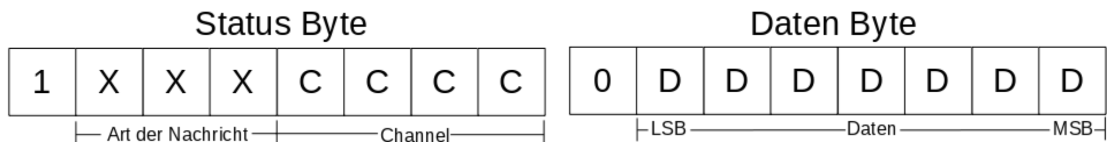

Abbildung 2.2.: Aufbau der Bytes der MIDI-Nachrichten

Über das MIDI-Protokoll lassen sich MIDI-Nachrichten verschicken. Eine MIDI-Nachricht besteht aus einem Statusbyte und Datenbytes. Um ein Einsteigen in einen kontinuierlichen MIDI-Datenfluss an jeder Stelle zu ermöglichen, ist das erste Bit eines Statusbytes immer 1, und bei Datenbytes 0. Somit liegt der nutzbare Wertebereich für die Datenbytes bei 7 Bit [0,127]. MIDI-Nachrichten sind in der Regel 3 Bytes lang. Das erste Byte (Statusbyte) legt fest von welcher Art die Nachricht ist, sowie den Kanal für den die Nachricht bestimmt ist. MIDI-Geräte können Bus-artig zusammengeschaltet werden. Eine MIDI-Nachricht erreicht somit alle verbundenen Teilnehmer. Um eine Nachricht an ein bestimmtes Gerät schicken zu können ist eine Kanal-ID notwendig. Die Bedeutung des 2ten und 3ten Bytes sind abhängig von der Art der Nachricht. Es sind folgende Arten für die Nachrichten festgelegt.

Nachricht	**Statusbyte**	**Datenbyte 1**	**Datenbyte 2**	**Beschreibung**
Note-off	0x8c	0xaa	0xbb	Deaktivierung einer Note. Versendet wird die Notennummer aa und die Release-velocity bb. Ist vorher kein Note-on eingetroffen, wird diese Nachricht ignoriert.
Note-on	0x9c	0xaa	0xbb	Aktivierung einer Note. Versendet wird die Notennummer aa und die Velocity (Anschlagsstärke) bb . Eine Note-on-Nachricht mit einer Velocity von 0 wird als Note-off interpretiert.
Polyphonic-aftertouch	0xAc	0xaa	0xbb	Druckstärke einer Taste nach dem Drücken. Es wird die Notennummer aa und der Druckwert bb gesendet.
Controll-change	0xBc	0xaa	0xbb	Änderung eines Kontrollparameters. Versendet wird Controler-Nummer aa und der Wert bb.
Program-change	0xCc	0xaa	-	Ändert das Instrument. Versendet wird die Instrument-Nummer aa.
Global-aftertouch	0xDc	0xaa	-	Gleiche Wirkung wie beim Polyphonic-aftertouch, allerdings bezogen auf alle Noten. Versendet wird Druckwert aa.
Pitch-bending	0xEc	0xaa	0xbb	Ändert die Tonhöhe der aktuell gespielten Töne feinstufig. Versendet wird der Änderungswert der sich aus beiden Datenbytes (aa LSB, bb MSB, zusammen 14 bit) zusammensetzt. 0x2000 bedeutet keine Änderung der Tonhöhe. Kleinere Werte stehen für das Senken, größere für das Anheben der Tonhöhe.
SysExclusive Message	0xFa	-	-	Steuermeldungen, die es Herstellern erlauben eigene MIDI-Nachrichten zu definieren. So können damit zum Beispiel Einstellungen für Synthesizer übertragen werden.

2.2. Audioverarbeitung

Zuerst wird beschrieben wie Schall vom menschlichen Gehör wahrgenommen wird. Dann wird auf die Erzeugung und Verarbeitung von Audiosignalen eingegangen. Schall sind Druckschwingungen die von einem Medium, beispielsweise der Luft, getragen werden. Eine Gitarrensaite oder ein Lautsprecher kann solche Schwingungen erzeugen.
Der Bereich in dem das menschliche Gehör Schall wahrnimmt ist ca. 20 Hz bis 20 kHz, wobei die Frequenzspanne im Alter besonders im höheren Bereich abnimmt. Frequenzen zwischen 500 Hz bis 5 kHz werden besonders gut vom Menschen wahrgenommen. Die Frequenzanteile der Schwingung bestimmt die Tonhöhe des Schalls. Die wahrgenommene Lautstärke wird von der Amplitude des Schalls bestimmt.
Ein Ton ist zusammengesetzt aus einem Grundton und Obertönen. Der Grundton hat die niedrigste Frequenz, und bestimmt in der Regel die Tonhöhe eines Tons. Die Anteile des Grundtons und der verschiedenen Obertöne in einem Ton bestimmt die Klangfarbe. Harmonische Obertöne habe eine ganzzahlige vielfache Frequenz des Grundtons. Reale Instrumente erzeugen neben dem Grundton immer eine gewisse Anzahl von Obertönen die mit der Zeit variieren. Desto mehr Obertöne ein Ton hat, desto heller klingt er. Wenn viele Obertöne enthalten sind, kann der Ton einen verrauschten Charakter annehmen. So enthält weißes Rauschen alle möglichen Frequenzen.

Die Note A4 ist definiert als 440 Hz. In der Musik werden Noten in Oktaven aufgeteilt. Eine Oktave bedeutet eine Verdopplung der Frequenz. Somit hat ein A5 eine Frequenz von 880 Hz. Oktaven werden jeweils noch in 12 Noten gleichmäßig aufgeteilt. Zwei benachbarte Noten hängen also durch einen Faktor von ca. 1.059 zusammen.
In dieser Arbeit wird mit digitalen Audiosignalen gearbeitet. Digitale Audiosignal sind eine Reihe von Zahlen, die ein kontinuierliches Signal repräsentieren. Diese Zahlen werden auch Samples genannt.
Die Samplerate f_s bestimmt den zeitlichen Abstand zwischen zwei Samples. Sie gibt die Anzahl von Samples in einer Sekunde an ($Hz = 1/s$). Nach dem Nyquist-Shannon-Sampling-Theorem muss die Samplerate mindestens doppelt so hoch sein wie die höchste im Signal vorhandene Frequenz, um eine Artefakt-freie Rekonstruktion des Signals zu ermöglichen. Bei einer Samplerate von 44.1 kHz sollten also maximal Frequenzen von 22.05 kHz im Audiosignal enthalten sein. Das ist ein Grund für die weitverbreitete (Bsp.: CD) Samplingfrequenz von 44.1 kHz, da Sie gerade hoch genug ist um alle für den Menschen hörbaren Frequenzen abzudecken.
Für die Darstellung der Samples können Floatingpoint- (Wertebereich [-1,1]) oder Ganzzahlen

(Wertebreich bei 16 bit: [-32768,32767]) verwendet werden. Um diese Daten mit einem Lautsprecher abspielen zu können müssen sie in ein analoges Signal gewandelt werden. Dazu kann ein DAC (Digital-Analog Wandler) verwendet werden. Dabei werden die Samples in analoge Spannungen gewandelt. Beispielsweise kann eine Ganzzahl im Bereich von [-32768,32767] auf eine Spannung im Bereich von [-3V,3V] abgebildet werden.

2.2.1. Klangsynthese

Klangsynthese ist eine Methode zur Generierung 'künstlicher' Klänge. Hierbei kann man unterscheiden zwischen analogen und digitalen Klangerzeugern. Erstere wurden größtenteils von den digitalen ersetzt, da diese die analogen simulieren können und zudem noch neue Möglichkeiten eröffnen.

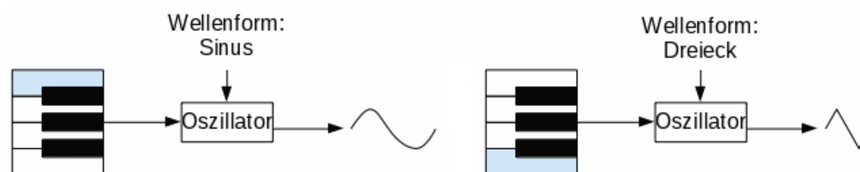

Abbildung 2.3.: variierende Wellenlänge, je nach gespielter Note. Verschiedene Wellenformen.

Die Basis eines Klangerzeugers ist ein Oszillator. Dieser kann auf verschiedene Arten realisiert sein, beispielsweise eine Analoge Schaltung oder eine digitale Tabelle. Meist stehen mehrere Oszillatoren zu Verfügung um einen komplexeren Klang zu erzeugen und/oder um eine bestimmte Polyfonie (mehrere gleichzeitig gespielte Noten) zu erreichen. Zur Verarbeitung des Klangs stehen weiterhin noch Modifikatoren wie Filter und Hüllkurven zur Verfügung. Es werden folgend einige Klangerzeugerarten vorgestellt.

Subtraktive Synthese

Subtraktive Klangsynthese ist eine der ältesten Methoden um Klang zu synthetisieren. Ursprünglich wurde Sie durch analoge Schaltungen realisiert, dabei wurden Analoge Spannungen (CV: Control Voltage) genutzt um die Parameter zu manipulieren. Es ist aber auch möglich diese Art der Klangerzeugung digital zu imitieren. Die subtraktive Synthese ist stark beschränkt in der Vielfalt von Klängen die erzeugt werden können. Aber auf Grund der leicht verständlichen Art der Klangerzeugung, und die dadurch schnelle Erlernbarkeit der Bedienung ist die Subtraktive Synthese dennoch weit verbreitet. Ein Beispiel für subtraktive Synthese ist der Minimoog.

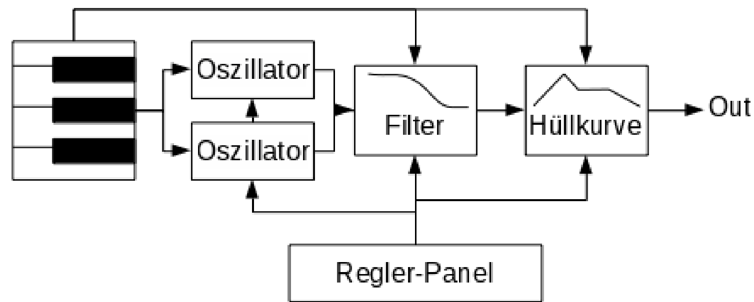

Abbildung 2.4.: Subtraktive Synthese, vereinfachter Aufbau

In Abbildung 2.4 ist ein vereinfachter Aufbau von subtraktiven Synthesizern zu sehen. Es werden obertonreiche Oszillatoren genutzt um den Grundton zu erzeugen, und mit Filtern werden ungewollte Obertöne wieder entfernt. Es werden zudem Hüllkurven genutzt um den Klang dynamischer zu gestalten.

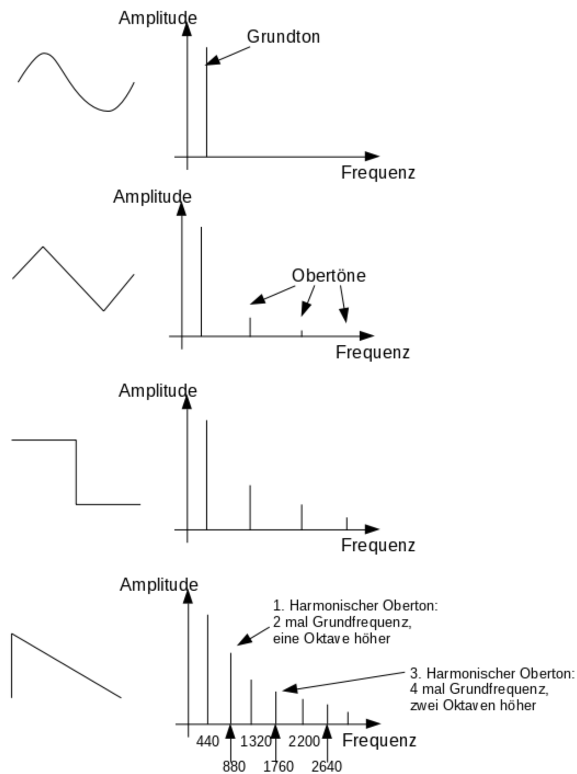

Abbildung 2.5.: simple Wellenformen, und ihre Frequenzspektren

Beispiele für Grundwellenformen die Oszillatoren erzeugen können sind Sägezahn, Rechteck, Dreieck und Sinus. Diese simplen Wellenformen haben nur Harmonische Obertöne, das heißt Töne mit einer ganzzahligen Vielfachen der Grundfrequenz. Eine Sinus-Wellenform besteht nur aus der Grundfrequenz und besitzt keine Obertöne. Somit lässt sich die Klangfarbe dieser auch nicht mit Filtern bearbeiten.

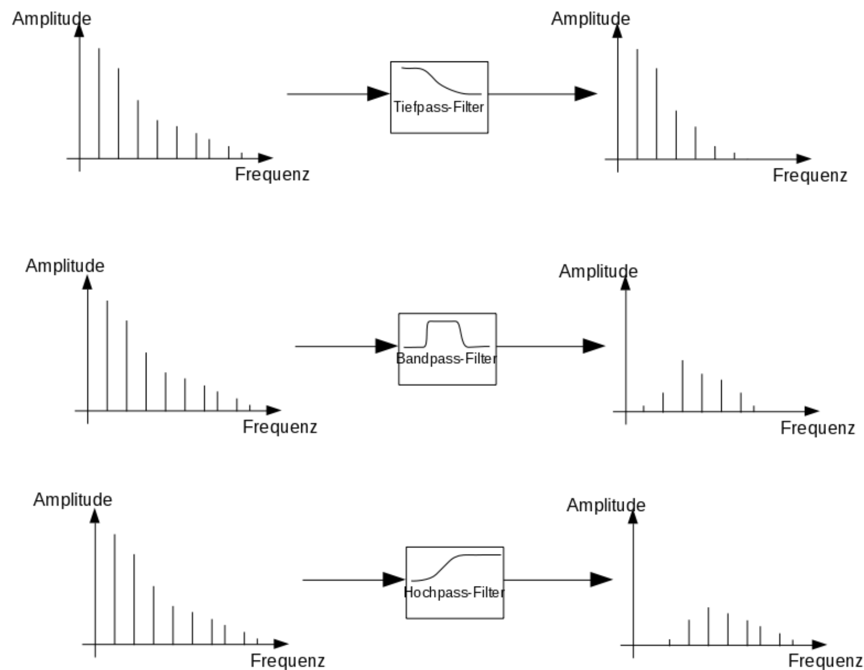

Abbildung 2.6.: Tiefpass Filter, Bandpass Filter, Hochpass Filter

Ein Filter ist eine Komponente die es ermöglicht bestimmte Frequenzanteile aus einem Signal zu entfernen. Als Filter werden unter anderem Tiefpass, Hochpass und Bandpass-Filter angewendet. Parameter für die Filter lassen sich in Echtzeit verstellen.

Ein Tiefpass-Filter ermöglicht es hohe Frequenzen ab einer Cutoff-Frequenz zunehmend abzuschwächen. Ein Durchlaufen der Cutoff-Frequenz von hoch nach tief erzeugt einen für analone Synthesizer charakteristischen Klang. Wenn die Cutoff-Frequenz gerade über der Grundfrequenz liegt, haben alle Wellenformen die gleiche Klangfarbe, da nur die Grund-Sinuswelle durchgelassen wird. Bei einem Erhöhen der Cutoff-Frequenz kommen dann die Obertöne hinzu, welche die jeweiligen Wellenformen charakterisieren.

Ein Hochpass-Filter hingegen lässt hohe Frequenzen durch, aber blockiert tiefe Frequenzen bis zu einer Cutoff-Frequenz. Dadurch kann sich die wahrgenommene Tonhöhe des Signals ändern, da die Grundfrequenz zuerst abgeschwächt wird.

Ein Bandpass-Filter kann als Kombination von Hoch- und Tiefpass-Filter gesehen werden, da nur Frequenzen in einem bestimmten Frequenzband durchgelassen werden. Enge Bandpass-Filter können für eine Frequenzanalyse verwendet werden. Eine Frequenz ist in einem Signal enthalten wenn Sie als Sinuston gehört wird bei einer Einstellung eines entsprechenden Frequenzbands.

Filter lassen auch ein Offset zu, so dass man zum Beispiel die aktuelle Tonhöhe mitgeben kann. So kann man für jede Tonhöhe das selbe relative Ergebnis erhalten.

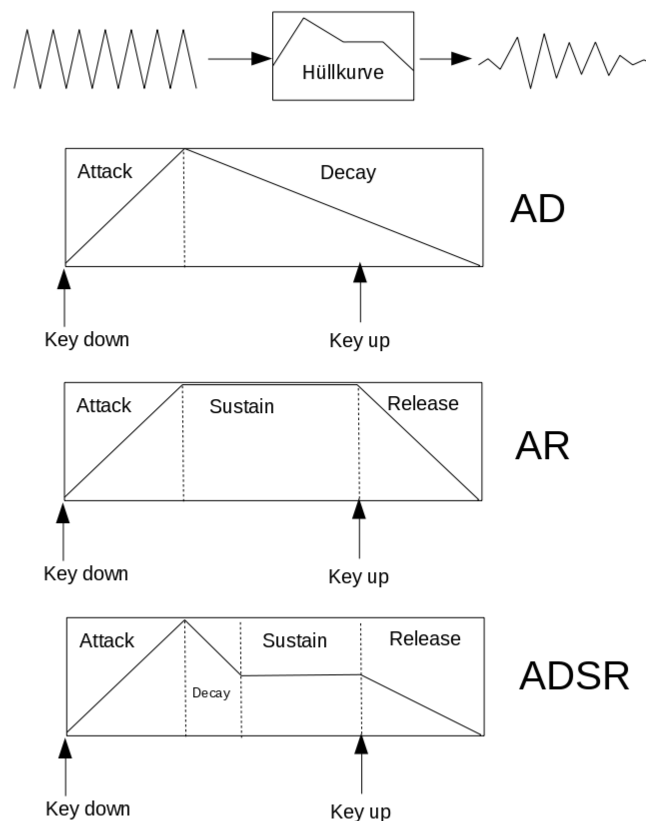

Abbildung 2.7.: verschiedene Arten von Hüllkurven im Vergleich

Mit einer Hüllkurve kann die Lautstärke einer Note vom Anschlag einer Taste bis zum Loslassen variiert werden. Es gibt verschiedene Arten von Hüllkurven. Einige Beispiele sind: AR, AD und ADSR. Hüllkurven sind in verschiedene Teile aufgeteilt. Die Zeit von Anschlag bis zum lautesten Punkt wird Attack genannt. Die Zeit von dem lautesten Punkt bis zum stabilen Wert wird Decay genannt. Der Abschnitt in der sich die Lautstärke nicht verändert wird Sustain genannt. Die Zeit vom Loslassen der Taste bis der Ton wieder abklingt wird Release genannt.

Bei einer AR (Attack Release) Hüllkurve kann nur die Anschlagzeit und die Abklingzeit angegeben werden. Bei einer AD (Attack Decay) Hüllkurve wird ebenfalls die Anschlagzeit aber statt der Abklingzeit wird ein Decay angegeben, welcher das abklingen des Tons sofort nach dem Erreichen des höchstens Punkts einleitet. Damit lassen sich nur perkussive Hüllkurven erzeugen. Eine flexiblere und trotzdem simple Hüllkurve ist ADSR (Attack Decay Sustain Release). Sie ist die weit verbreitetste Hüllkurve. Hier kann zusätzlich zur Attack- und Release-Zeit noch eine Decay-Zeit und ein Sustain-Wert angegeben werden. Somit steigt die Lautstärke in der Attack-Zeit vom Notenanschlag bis zum höchsten Wert. Danach sinkt die Lautstärke in der Decay-Zeit wieder bis zum Sustain-Wert ab. Nach dem Loslassen einer Taste sinkt der Wert dann von Sustain nach 0 in der Release-Zeit.
Es gibt noch weitere Arten von Hüllkurven, die eine noch feinere Kontrolle über die Lautstärke ermöglichen. Die Hüllkurven können oft nicht nur von Keyboards, sondern auch von anderen Quellen gesteuert werden, wie z.B. einem LFO.

Additive Synthese

Im Gegensatz zur subtraktiven Synthese werden hier Klänge durch das Mischen von verschiedenen harmonischen Teiltönen erzeugt. Hierzu werden einfache Sinustöne genutzt. Basierend auf der Fourieranalyse kann gezeigt werden, das jeder Ton aus einer Mischung von verschiedenen Sinustönen synthetisiert werden kann. Es gibt für jeden Oberton eine Einstellungsmöglichkeit der Hüllkurven-Parameter und der Amplitude. Aus diesem Grund sind die Anzahl der Regler oft um einiges höher als bei subtraktiven Synthesizern. Es gibt allerdings auch die Möglichkeit die Parameter zu gruppieren, so dass man die Parameter für eine Hüllkurve beispielsweise direkt für mehrere Obertöne nutzt.

Filter sind für diese Art von Synthese nicht nötig, da eine Filterung durch die feine Obertoneinstellung erzielt werden kann. Die Vielfalt der Klänge ist bei der additiven Synthese um einiges größer als bei der subtraktiven Synthese, allerdings ist die Steuerung der vielen Parameter sehr aufwändig. Da man für jeden Oberton einen eigenen Oszillator benötigt ist die

technische Umsetzung oft nicht einfach. Dies sind Gründe für die geringe Verbreitung von additiven Synthesizern. Ein Beispiel für diese Art der Klangsynthese ist die Zugriegelorgel.

Wavetable Synthese

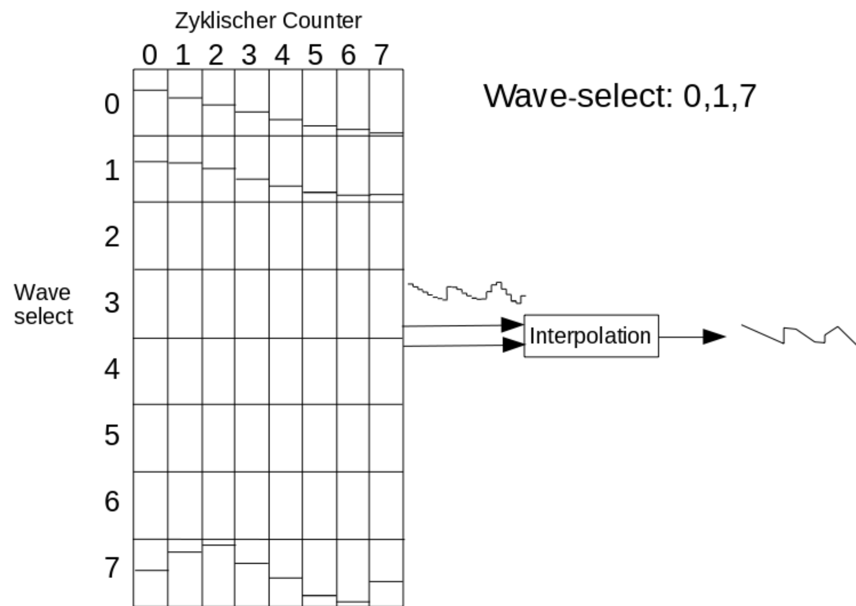

Abbildung 2.8.: Wavetable Synthesizer mit linearer Interpolation

Wavetable Synthesizer haben als Oszillatoren anstelle von Grundwellenformen ganze Tabellen die mit Wellenformen gefüllt sind. Es gibt verschiedene Arten wie mit diesen Tabellen gearbeitet werden kann, deshalb gibt es auch viele verschiedene Definitionen für die Wavetable Synthese.

Eine Möglichkeit ist es die Tabelle von Anfang bis Ende zu durchlaufen. Eine andere Möglichkeit ist es den Index der Wellenform per Zufall auszuwählen. Eine dritte Möglichkeit ist es den Index von der Hüllkurve abhängig zu machen, so dass man für die jeweiligen Segmente in einem anderen Bereich der Wavetable abspielt.

Es besteht die Möglichkeit zwischen den verschiedenen Wellenformen zu interpolieren, um einen glatteren Klang zu erhalten. Da es möglich ist eine große Menge von Wellenformen in den Tabellen unterzubringen, haben die Samples einen großen Einfluss auf den Klang des Synthesizers. Somit ist es beispielsweise auch möglich ähnliche Ergebnisse wie ein Sampler zu

erzielen. Bei Wavetable Synthesizern werden aber oft kleinere Tabellen genutzt, und der Klang der Samples wird durch die Wavetable Synthese deutlich verändert. Wavetables werden oft genutzt um lange, sphärische Klänge zu erzeugen.

Filter und Hüllkurven können ähnlich wie bei der subtraktiven Synthese auf den Output der Wavetable angewandt werden um die Klangfarbe und Dynamik zu verändern. Ein weiteres Mittel die Klangfarbe anzupassen sind Waveshaper.

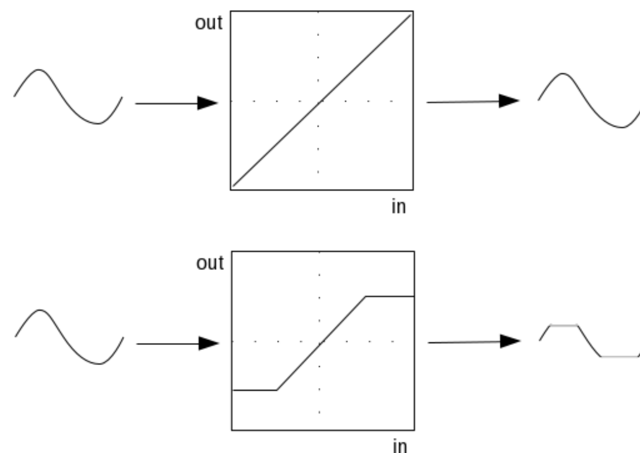

Abbildung 2.9.: oben: linearer Waveshaper, verändert Signal nicht.
unten: durch Waveshaper verzerrtes Signal

Waveshaper sind im Grunde Lookuptables, die Werte von -1 bis 1 auf Werte von -1 bis 1 abbilden. Damit lassen sich Wellenformen nicht-linear verzerren. Wenn der Waveshaper eine gerade Linie ist wie in Abbildung 2.9 oben, dann wird der Eingang nicht verändert. Auf Abbildung 2.9 unten sieht man eine durch den Waveshaper verzerrte Wellenform.

Das Verzerren einer Wellenform ändert ihre Oberton-Anteile. In den meisten Fällen werden neue Obertöne erzeugt. Waveshaper sind vielseitig einsetzbar. Sie können direkt auf einen Oszillator angewandt werden um die Klangfarbe zu ändern, oder aber auch auf das bereits gemischte polyphone Signal, um eine Verzerrung zu realisieren.

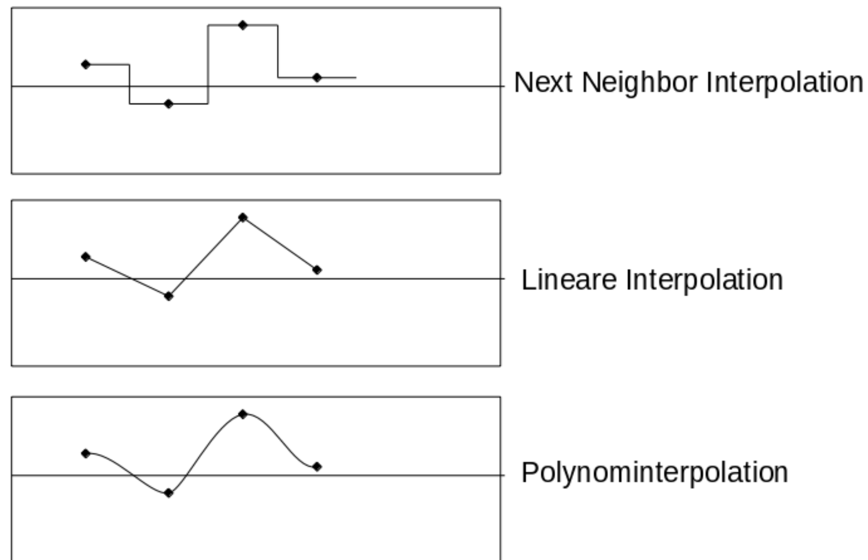

Abbildung 2.10.: verschiedene Interpolationsarten

Interpolation

Da bei Wavetable-Synthesizern im Gegensatz zu Samplern die Wellenformen nur in einer Periode gespeichert sind, ist es sinnvoll zu Interpolieren, um eine ähnliche Klangqualität über das gesamte Frequenzspektrum zu erhalten.

Es gibt unterschiedliche Arten der Interpolation. Die einfachste und Ressourcen schonendste Methode ist Nearest Neighbour. Dabei wird einfach der nächste ganze Wert genommen, der am nächsten zu dem Index liegt. Der Index wird also einfach gerundet. Das Ergebnis ist aber oft nicht zufriedenstellend. Eine weitere Methode ist lineare Interpolation. Dabei wird zwischen zwei Werten eine Gerade gezogen. Die Nachkommastellen des Index' werden hier als Gewichtung genutzt um die beiden Werte zu verrechnen. Diese Methode ist etwas rechenaufwendiger, aber liefert schon eine gute Interpolation. Eine weitere Methode ist das Annähern durch Polynome höheren Grades. Dabei wird eine Kurve durch drei oder mehr Punkte gezogen. Der Rechenaufwand nimmt mit zunehmendem Grad des Polynoms zu, liefert dafür aber auch sehr gute Ergebnisse, da Audiosignale oft wellenförmige Strukturen haben anstatt geraden Linien.

2.2.2. Audioeffekte

Ein Audioeffekt kann man sich als Blackbox vorstellen, die Audiosignale entgegen nimmt, verarbeitet und das Ergebnis ausgibt. Bei digitalen Audioeffekten sind die Audiosignale eine Reihe von Samples.

Abbildung 2.11.: Gewichtung eines digitalen Audiosignals mit einem Faktor

Dabei wird unterschieden zwischen Sample-für-Sample und Blockverarbeitung. Ein simples Beispiel welches die Daten Sample-für-Sample verarbeitet ist eine Gewichtung. Dabei wird jedes Sample mit einem festem Wert g multipliziert. Dadurch wird nur die Amplitude des Signals verändert. Ein Beispiel für einen Blockverarbeitungs-Algorithmus ist die Fourier Transformation, welche genutzt wird um auf dem Frequenzspektrum eines Signals zu arbeiten. Hierbei werden die Samples verzögert ausgegeben, zu dem Input x(n) erhält man den Output y(n-m). Diese Verzögerung m wird Latenz genannt.

Modulation

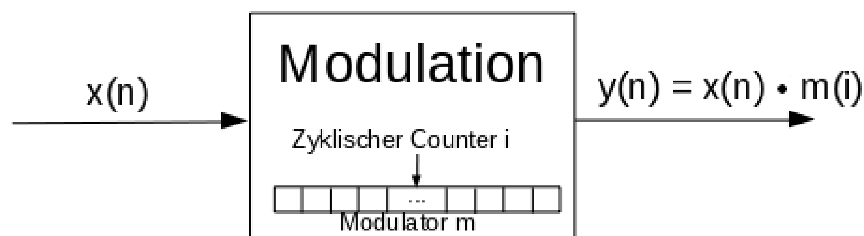

Abbildung 2.12.: Modulation eines digitalen Audiosignals mit einem Modulator

Eine Modulation ist auch eine einfache Multiplikation mit dem Eingangssignal. Allerdings wird hier ein Modulator verwendet. Ein Ringmodulator ist ein Array von Samples, das zyklisch durchlaufen wird. Wenn als Modulator ein Sinussignal verwendet wird, können damit schwingende Effekte erzielt werden. Wenn ein verschobenes Sinussignal verwendet wird, welches sich nur in dem Wertebereich oberhalb der 0 aufhält, kann damit auch ein Tremoloeffekt erzielt werden. Dafür kann folgende Formel verwendet werden, c ist dabei die Intensität des Effekts

und hat einen Wertebereich von 0...1.

$$y(n) = x(n) \cdot (\frac{1 + m(i)}{2} \cdot c + (1 - c))$$

Dieser Effekt der variierenden Amplitude ist nur bei einem Modulator im niedrigen Frequenzbereich festzustellen. Wenn ein Modulator mit höherer Frequenz verwendet wird, entstehen neue hörbare Obertöne im Signal.

Faltung

Abbildung 2.13.: Faltung eines digitalen Audiosignals mit einem Faltungskern

Bei dem Faltungshall wird die diskrete Faltung genutzt um den Hall eines Raumes zu simulieren. Dazu wird die Impulsantwort in einem beliebigem Raum aufgenommen und dann mit dem Audiosignal gefaltet. Man erhält nun ein Audiosignal welches die Klangfarbe des Raums besitzt. Faltung ist ein generischer Operator, mit dem sich zwei Signale verrechnen lassen. Dabei wird die Struktur des zweiten Signals auf das Erste übertragen. So kann z.B. ein Audiosignal mit der Impulsantwort eines Raumes gefaltet werden, um die Klangfarbe des Raumes auf das Audiosignal zu übertragen. Da die Faltung als FIR sehr rechenaufwendig ($O(n^2)$) ist, müssen Techniken wie Fast-convolution (die sich auf die Fast-Fourier-Transformation stützt) genutzt werden, um die Faltung effektiv für Echtzeit-Audioverarbeitung nutzen zu können (Komplexität $O(n \cdot log(n))$). Fast-convolution hat allerdings die Eigenschaft das eine Latenz mit der Länge der Impulsantwort entsteht. Diese Latenz kann aber durch gewisse Techniken minimiert werden. Beispielsweise indem nur kleinere Teile der Signale gefaltet werden. Dadurch wird allerdings wieder der Rechenaufwand erhöht. Dabei muss zwischen Latenz und Rechenaufwand abgewogen und das optimale Verhältnis für die jeweilige Anwendung gefunden werden.

Fourier Transformation

Die Fourier Transformation ist eine mathematische Transformation, die es ermöglicht Signale aus dem Zeitbereich in den Frequenzbereich und zurück zu transformieren. Für die digitale Signalverarbeitung wird die diskrete Fourier Transformation (DFT) genutzt. Die Ausgabewerte sind komplexe Werte.

$$S_k = \sum_{n=0}^{N-1} s_n \cdot e^{\frac{-i2\pi kn}{N}}$$

Da das Berechnen der DFT mit dieser Formel zu rechenintensiv ist, wird in der Praxis meistens ein effizienter Fast-fourier-Transformation(FFT)-Algorithmus genutzt. Ein Beispiel ist der Cooley/Tukey Algorithmus, der nach dem Teile-und-Herrsche-Prinzip die Transformationen rekursiv in kleinere Teile zerlegt.

2.3. MIDIbox

Das MIDIbox-Projekt ist ein modulares Open-source-framework für Hard- und Software rund um MIDI. Es ist möglich damit eine Vielzahl von MIDI-Applikationen zu realisieren, wie beispielsweise Sequenzer oder Synthesizer. Es gibt eine MIDIbox Hardware Platform (MBHP) und die Software Platform MIOS (MIDIbox Operating System).
Es gibt verschiedene Core-Module, die neben einem Mikrocontroller Schnittstellen für andere Module besitzen. Als 8-Bit Version gibt es ein Core-Modul das auf PIC-Mikrocontrollern basiert. Zudem gibt es noch 32-bit Core-Module, welche die Mikrocontroller STM32, STM32F4 und LPC1769 verwenden. Die verschiedenen Core-Module haben im Grunde die gleichen Funktionalitäten, so dass sie auch austauschbar sind, allerdings unterscheiden sie sich unter anderem in der Performance und Speicher den sie besitzen.

Es gibt zudem noch Hardware-Module, mit denen das Core-Module erweitert werden kann. Es gibt digitale und analoge Ein- und Ausgabe-Module, die beliebig in Reihe geschaltet werden können um beispielsweise eine große Anzahl von Schaltern und Reglern zu nutzen. Des weiteren existieren noch das LCD-Ausgabe-Modul, DAC-Modul, SD-Karten-Modul und weitere. Es sind schon einige fertige Synthesizer und andere Projekte vorhanden die auf dem MIDIbox-Framework aufbauen.

Im Rahmen dieser Bachelorarbeit wurde das Core-Module mit dem Mikrocontroller LPC1769

genutzt. Der LPC1769 besitzt 512kB Flash und 64kB RAM Speicher. Der 32-bit Prozessor ist ein ARM-Cortex m3 mit einer Taktfrequenz von bis zu 120 MHz und 1.25 DMIPS/MHz. Um eigene Applikationen auf die MIDIbox hochladen zu können muss zuerst der MIOS-bootloader in den Flash-Speicher geschrieben werden.

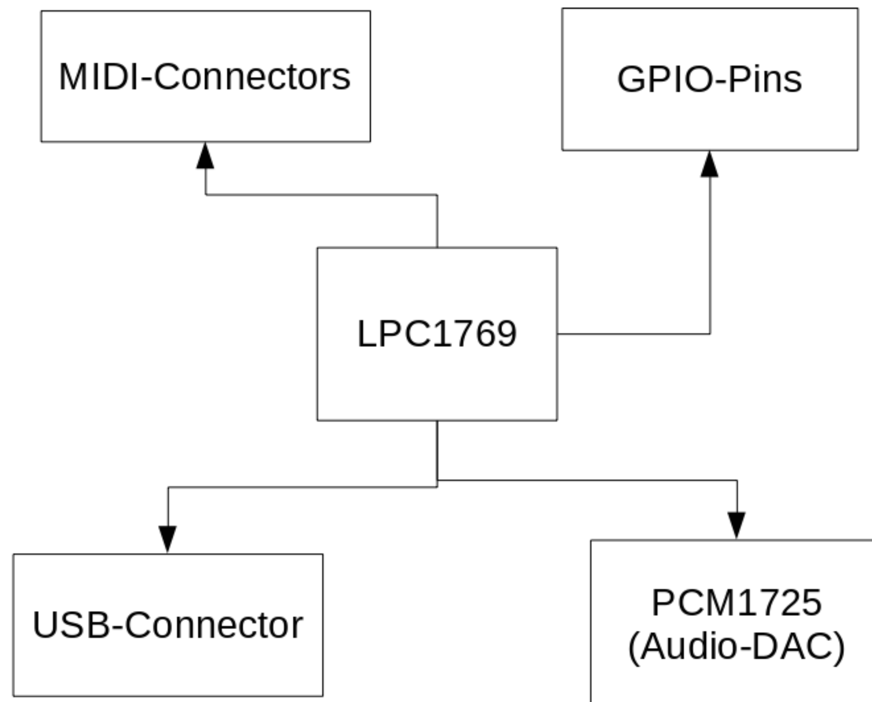

Abbildung 2.14.: Blockschaltbild der Hardware

2.3.1. MIOS

MIOS ist ein Betriebssystem für die MBHP und stellt ein Hardware-abstraction-layer bereit. Damit wird der Zugriff auf MIDI-Schnittstellen und Audioausgabe erleichtert. Der MIOS-bootloader erlaubt auch das Hochladen von Applikationen über MIDI-sysex-Nachrichten. Über MIDI-sysex-Nachrichten ist es auch möglich in Applikationen Debug-Nachrichten zu verschicken oder Kommandos von einem Terminal entgegenzunehmen.

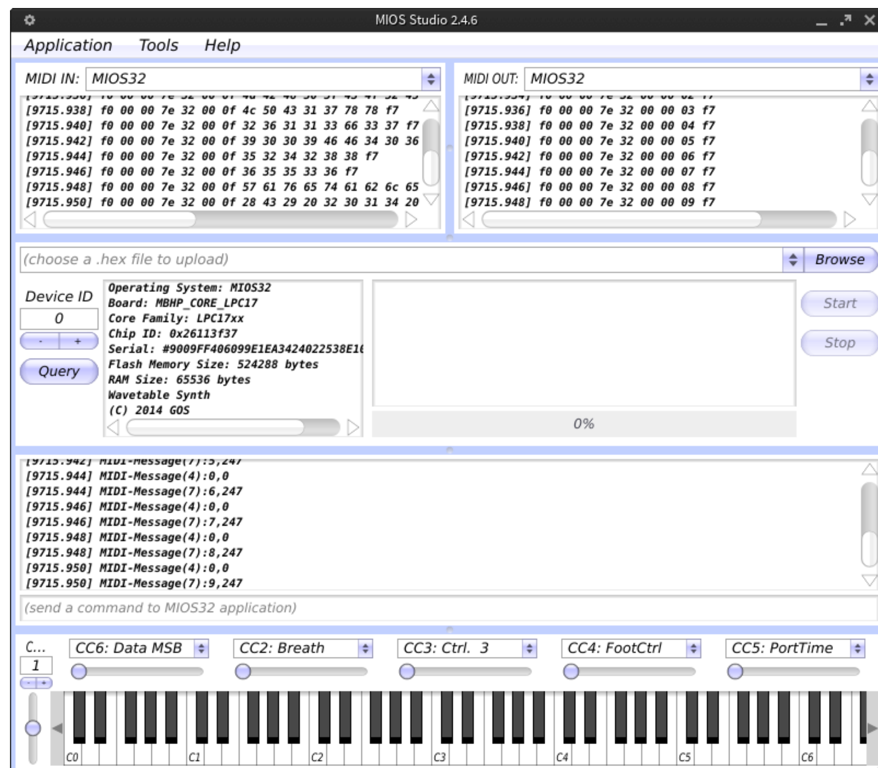

Abbildung 2.15.: MIOS-Studio. GUI zum Hochladen und Testen von Applikationen

Um Applikationen für MIOS zu entwickeln muss zuerst die gcc-toolchain eingerichtet werden. Dazu gehört ein Compiler für den jeweiligen Prozessor sowie die 'newlib'. Die Daten und eine Beschreibung zum Einrichten für verschiedene Betriebssysteme (Linux, Windows, Mac) findet man auf der Webseite [3 (2015)] . In dieser Arbeit werden die MIOS-Treiber für die Audio-Ausgabe und den MIDI-Eingabe verwendet. MIOS verwendet Teile vom Real-time-Betriebssystem freeRTOS. So ist es möglich beispielsweise Tasks und Semaphore von freeRTOS benutzen.

3. Konzept

Es wurde ein Konzept für die Plugin-Schnittstellen erstellt. Dafür wurde ein Framework entwickelt, welches die Details der Audioausgabe und MIDI-Nachrichten verbirgt, und Schnittstellen für grundlegende Techniken für Synthesizer anbietet. In diesem Kapitel wird eine Übersicht über das Framework gegeben und die einzelnen Komponenten erklärt.

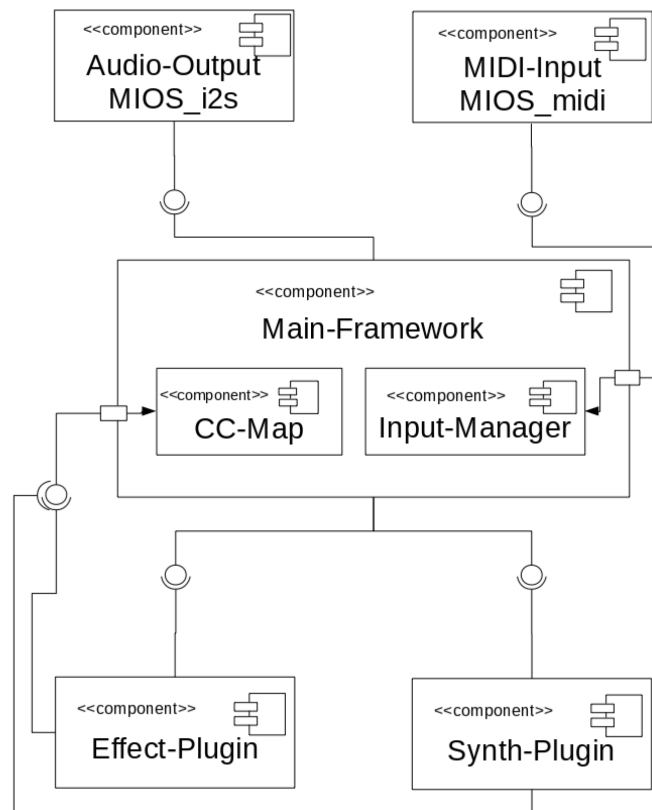

Abbildung 3.1.: Komponentendiagram des Plugin-Systems

3.1. Überblick

Das Framework besteht aus

- den Treibern von MIOS
- Main-Framework
- Input-Manager
- CC-Map
- Plugins

Das Main-Framework ist die zentrale Komponente des Systems, da es mit allen anderen Komponenten kommuniziert. Sie bietet die Schnittstelle für die Plugins an. Der Input-Manager ist für die Verwaltung eingehender MIDI-Nachrichten verantwortlich. Die CC-Map leitet Control-Change MIDI-Nachrichten an registrierte Teilnehmer weiter.

Es werden zwei Module von MIOS verwendet. Zum einen wird das I2S-Modul von MIOS für das schreiben der Samples auf den DAC verwendet. Des Weiteren wird das Programmiermodel von MIOS verwendet. Dabei gibt es ein Interface mit Funktionen, welche implementiert werden müssen um eine Applikation zu erstellen. Diese Funktionen werden aufgerufen wenn Ereignisse aufgetreten sind. Darunter befindet sich auch die Funktion die bei Empfang von MIDI-Nachrichten aufgerufen wird.

3.2. Main-Framework

In dem Main-Framework werden folgende Aufgaben durchgeführt:

- Initialisierung der weiteren Module: CC-Map, Input-Manager und MIOS-I2S.
- Laden der angegebenen Plugins.
- Es wird auf MIDI-Nachrichten reagiert.
  - Channel-Nachrichten werden an den Input-Manager weitergeleitet.
  - CC-Channel-Nachrichen werden an die CC-Map weitergeleitet.
  - Es wird auf Terminal-Kommandos reagiert, welche als MIDI-Nachrichten eintreffen.

  – Ein generischer Command-Callback-Mechanismus ist noch nicht implementiert, ist aber möglich und sinnvoll für Debug-Zwecke.

- Zudem wird zyklisch der Buffer für die Audioausgabe gefüllt, indem die Füll-Funktion des Synthesizer-Plugins und die Verarbeitungs-Funktionen der Effekt-Plugins aufgerufen werden. Es wird Blockweise- und Sample-by-Sample-Verarbeitung berücksichtigt.

Das folgende Listing zeigt Pseudocode für den Ablauf des zyklischen Füllen des Buffers.

Zeile 2: Hier werden die aktiven Noten vom Input-Manager geholt.

Zeile 4-6: Füll-Funktion des Synthesizers bei Blockverarbeitung. Der Synthesizer füllt das Array 'values' mit seinen Samples. Es werden SAMPLE_BUFFER_SIZE Samples auf einmal synthetisiert.

Zeile 8: Für jedes einzelne Sample ...

  Zeile 9-13: Füll-Funktion des Synthesizers bei Sample-für-Sample-Verarbeitung. Der Synthesizer füllt das Array 'values' an der aktuellen Stelle mit einem Sample.

  Zeile 15: Tick mit der Sample-Frequenz. Der Input-Manager berechnet damit die Hüllkurven für die aktiven Noten.

  Zeile 17-21: Füll-Funktion der Effekte bei Sample-für-Sample-Verarbeitung. Die Effekte bearbeiten das Sample an der aktuellen Stelle im Array 'values'.

Zeile 25-29: Füll-Funktion des Synthesizers bei Block-Verarbeitung. Die Effekte bearbeiten das gesamte Array 'values'.

Zeile 31-33: Die verarbeiteten Samples werden geclippt und in den Buffer für den DAC-Treiber geschrieben.

Zeile 35: Input-Manager wird aufgefordert die Zustandsübergänge für die aktiven Noten zu prüfen.

```
void reloadSampleBuffer(int *buffer) {
  Note active_notes[] = get_active_notes();

  if(synth->block_processing) {
    synth->synthesize_samples(active_notes, values, synth->data);
  }

  for i in 0...SAMPLE_BUFFER_SIZE-1 {
    if(!synth->block_processing) {
      synth->synthesize_sample(active_notes,
                               &values[i*CHANNELS],
                               synth->data);
    }

    move_adsr_index(active_notes);

    for j in 0...NUM_EFFEKTS-1 {
      if(!effekts[j]->block_processing)
        effekts[j]->process_sample(&values[i*CHANNELS],
                                   effekts[j]->data);
    }

  }

  for j in 0...NUM_EFFEKTS-1 {
    if(effekts[j]->block_processing)
      effekts[j]->process_samples(values,
                                  effekts[j]->data);
  }

  for i in 0...NUM_CHANNELS*SAMPLE_BUFFER_SIZE - 1 {
    buffer[i] = clip(values[i]);
  }

  check_adsr_state_change(active_notes);
}
```

3.3. Input-Manager

Der Input-Manager verwaltet die eingehenden MIDI-Nachrichten. Es werden Noten-Informationen unter Berücksichtigung von Polyfonie und ADSR-Hüllkurven-Informationen verwaltet. Zusätzlich werden hier MIDI-Nachrichten wie Pitchbend und Aftertouch gespeichert. Die maximale Polyfonie wird zur Compile-Zeit angegeben. Damit kann der Grad der Polyfonie zur Compile-Zeit auf die verfügbare Rechenleistung angepasst werden, um unvorhersehbare Effekte zur Laufzeit zu vermeiden. Der Input-Manager verfügt über Speicherressourcen für die Noten-Informationen. Bevor neue Noten hinzugefügt werden können, muss eine Speicherressource vom Input-Manager angefordert werden. Es ist möglich den Input Manager im Monophonic-Modus zu betreiben, dann wird immer nur die zuletzt gedrückte Taste als aktive Note erkannt. Die Noten-Informationen werden in folgender Struktur abgelegt.

```
1 struct Note {
2     u32 seqnum;
3     int number;
4     int state;
5     u8 velocity;
6     u32 adsr_vol;
7 };
```

seqnum: Logische Zeit des Startzeitpunkts der Note, zur Identifizierung der Reihenfolge in der Noten gespielt wurden.

number: Notennummer, die zugehörige Frequenz kann mit dem frqtab-Modul berechnet werden.

state: Zustand der ADSR-Statemachine.

velocity: Anschlagstärke der Note.

adsr_vol: Pegel der Hüllkurve. Lautstärke der Note setzt sich zusammen aus $velocity \cdot adsr_vol$.

Der Input-Manager bietet folgende Funktionen zur Verwaltung der Noten an:

- int im_get_active_notes(struct Note ***active_notes): Call by referece. Nach Aufruf zeigt der Pointer auf die Adresse des Arrays der Note-Pointer.
- int im_get_note_index(int num): Gibt die Position der Note (mit der Notennummer num) in der Liste zurück.

- struct Note* im_get_note(int index)
- void im_put_note(struct Note *note): Noten-Ressource muss vorher durch 'im_get_free_note' angefordert werden.
- void im_remove_note(int index)
- int im_get_free_note(struct Note **free_note): Call by reference. Nach Aufruf zeigt der Pointer auf die Adresse einer freien Speicherressource.

Voice-stealing

Der Input-Manager nutzt eine Technik, die sich Voice-stealing nennt. Diese wird genutzt wenn alle Oszillatoren bereits besetzt sind und eine neue Note gespielt wird. Dabei wird einer alten Note der Oszillator entzogen und der neuen Note bereitgestellt. Es gibt verschiedene Möglichkeiten die zu verwerfende Note auszuwählen. Eine Möglichkeit ist es die älteste Note die sich im Release Segment der Hüllkurve befindet auszuwählen. Wenn sich keine Note im Release Segment befindet, wird die neuste Note ausgewählt.

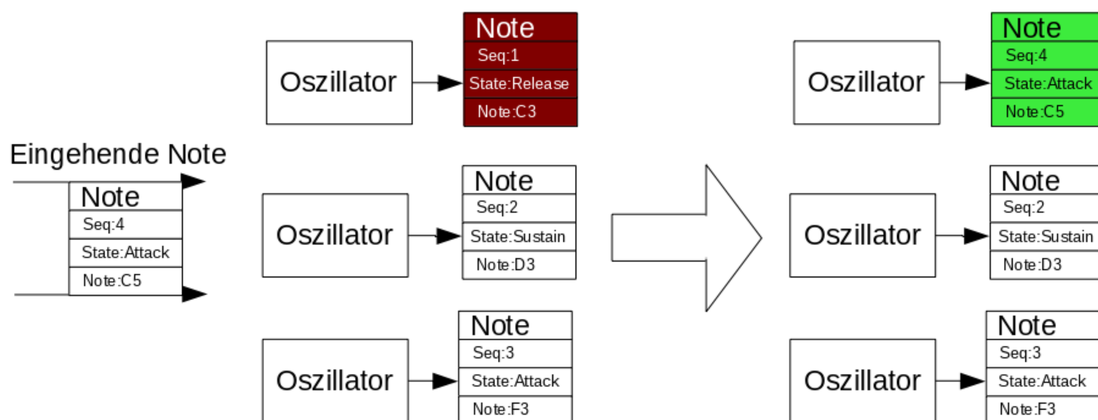

Abbildung 3.2.: Voice-stealing

Auf Abbildung 3.3 ist die ADSR-Hüllkurven-Statemachine zu sehen, welche die Zustände einer Note im Input-Manager zeigt. Aus dem Zustand einer Note wird ihre Hüllkurve berechnet. Siehe Abbildung 2.7 im Grundlagenkapitel. Der Zustand einer Note wird auch anderweitig verwendet, wie beispielsweise beim Voice-stealing.

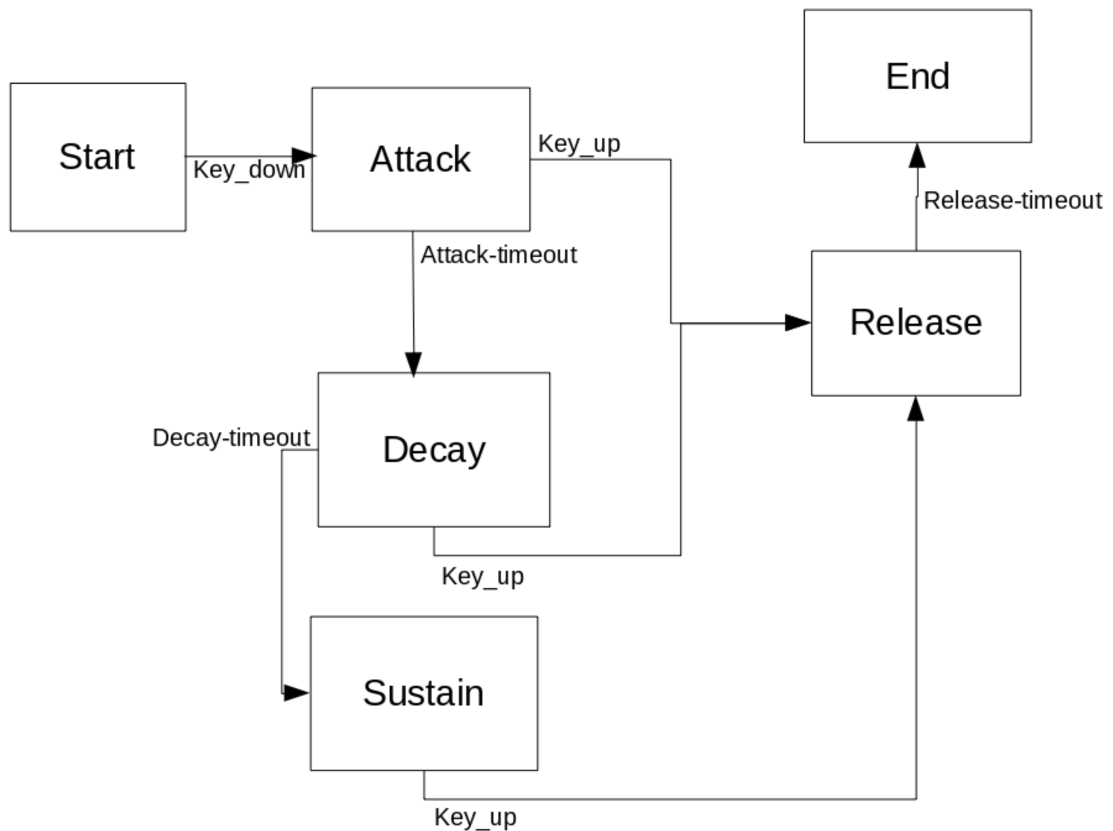

Abbildung 3.3.: ADSR-Hüllkurven-Statemachine

Sperren für Input-Manager

Eine Notwedigkeit für Sperren kann entstehen, wenn 2 oder mehr Threads auf eine Datenstruktur zugreifen. Hier sind es folgende Ereignisse:

- Das zyklische Füllen des Ausgabebuffers. Hier wird die Liste der aktiven Noten abgefragt.
- Die Interrupts, welche durch die MIDI-Nachrichten Note-on/Note-off ausgelöst werden.

Es werden Sperren für das Hinzufügen, Entfernen und Abfragen von Noten benötigt, da hier parallel auf der selben Datenstruktur gearbeitet wird.

3.4. CC-Map

Die CC-Map soll es den verschiedenen Komponenten ermöglichen auf Control-Change-MIDI-Nachrichten zu reagieren. Für die Klangerzeuger- und Effektplugins lassen sich somit Parameter über Regler von einem MIDI-Keyboard steuern. Es existieren 127 verschiedene Control-Change-Nachrichten. Es existieren feste CC-Nachrichten wie z.B. für das Modulation-Wheel, aber auch frei zuweisbare Nachrichten-Nummern. Die CC-Map soll eine Abbildung von den Nachrichten-Nummern auf Funktionspointer realisieren. Eine Komponente wie beispielsweise ein Effekt-plugin kann eine Funktion auf eine bestimmte Control-Change-Nummer registrieren. Diese Funktion wird dann aufgerufen wenn die entsprechende Nachricht angekommen ist. Übergeben wird die Nummer der Nachricht und der aktuelle Wert.

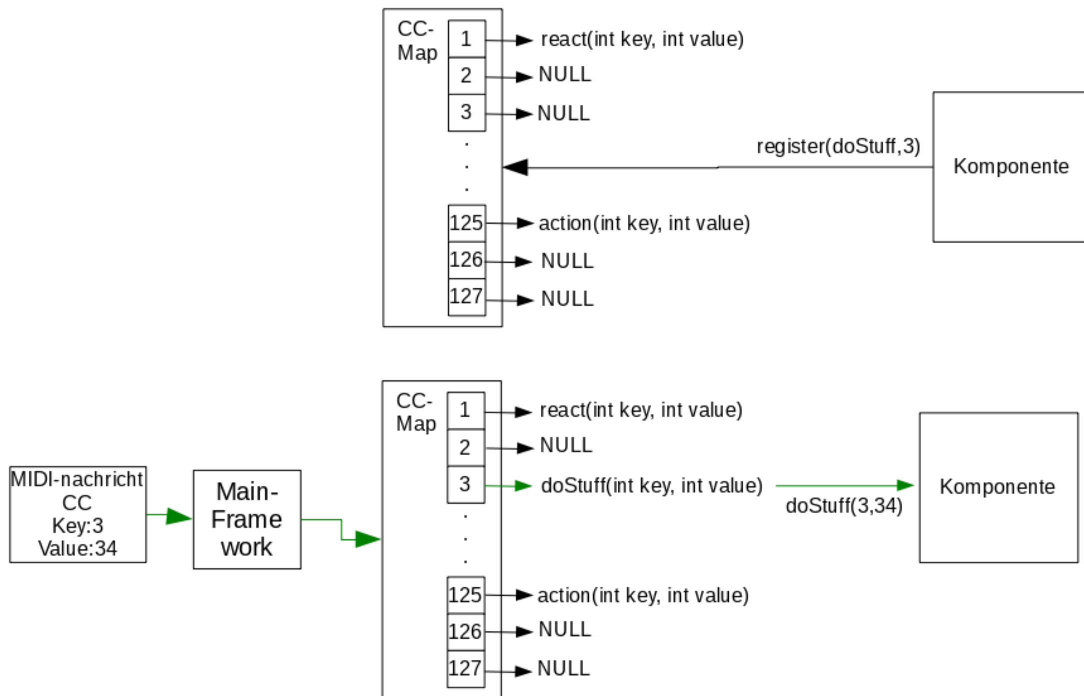

Abbildung 3.4.: Arbeitsweise der CC-map

3.5. Plugin-Interfaces

In diesem Abschnitt werden die Schnittstellen für Klangerzeuger und Effekte erläutert. Es sind jeweils Strukturen mit Funktionspointern definiert, die implementiert werden müssen. Die Plugin Strukturen müssen gefüllt und dem Main-Framework übergeben werden. Mit

```
set_synth_plugin(struct Synth_plugin *synthesizer)
```

kann ein Klangerzeuger für das Framework gesetzt werden. Das Synthesizer-Plugin kann entweder im Block-Verarbeitungs- oder im Sample-by-Sample-Modus genutzt werden. Wenn Block-Verarbeitung aktiviert ist, muss das Plugin ein Block von Samples beim Aufruf der Funktion synthesize_samples erzeugen. Ansosten muss das Plugin ein einziges Sample beim Aufruf der Funktion synthesize_sample erzeugen. In folgendem Listing sind die Elemente der Synthesizer-Plugin Struktur aufgelistet.

```
struct Synth_plugin {
    int block_processing;
    void (*synthesize_samples)(struct Note **active_notes,
                               int num_notes, s32 *values,
                               int block_size, void *data);
    void (*synthesize_sample)(struct Note **active_notes,
                              int num_notes, s32 *values,
                              void *data);
    void  (*change_program)(int num, void *data);
    void  (*note_start)(struct Note *note, void *data);
    void  (*note_end)(struct Note *note, void *data);
    void *data;
};
```

- **block_processing** Gibt an ob das Plugin Block- oder Sample-by-Sample-Verarbeitung nutzt.
- **synthesize_samples** Synthetisiert einen Block von Samples, mitgegeben werden die momentan aktiven Noten, die Blockgröße und ein Buffer für die Ausgabesamples. Die Ausgabesamples setzten sich aus Samples für mehrere Channels zusammen. Die Anzahl lässt sich aus der constants.h entnehmen.
- **synthesize_sample** Synthetisiert ein Sample.
- **change_program** Wechselt das Programm (oder Preset) des Klangerzeugers. Wertebereich Num: [0,127]

- **note_start** Wird aufgerufen wenn eine neue Note gespielt wurde.
- **note_end** Wird aufgerufen wenn eine Note ausgeklungen ist.
- **data** Pointer auf Daten des Plugins.

Mit

```
add_effekt_plugin(struct Effekt_plugin *effekt)
```

kann ein Effekt hinzugefügt werden. Das Effekt-Plugin kann analog zum Synthesizer-Plugin entweder im Block-Verarbeitungs oder im sample-by-sample-Modus genutzt werden. Es können bis zu MAX_EFFECT Effekte hinzugefügt werden. Die Anzahl der gleichzeitig möglichen Effekte hängt von dem Rechenaufwand der Effekte ab.

```
struct Effekt_plugin {
    u8 effect_id;
    int block_processing;
    void (*process_samples)(s32 *values, int block_size, void *data);
    void (*process_sample)(s32 *values, void *data);
    void *data;
};
```

- **effect_id** Macht dieses Effekt-Plugin eindeutig.
- **block_processing** Gibt an ob das Plugin Block- oder Sample-by-Sample-Verarbeitung nutzt.
- **process_samples** Verarbeitet die Samples blockweise. In dem Buffer 'values' ist ein Block von rohen Samples für alle Channels enthalten. Ausgabewerte überschreiben die rohen Samples im selben Array.
- **process_sample** Verarbeitet ein Sample.
- **data** Pointer auf Daten des Plugins.

Durch den generischen Pointer 'data' ist es dem Plugin-Entwickler möglich eine eigene Daten-Struktur für das Plugin zu verwenden. Dieser Pointer wird bei jeder Funktion der Plugin-Schnittstellen mit übergeben.

3.5.1. Wavetable-Synthesizer

Als Beispiel für einen Klangerzeuger-Plugin wurde ein Wavetable-Synthesizer entwickelt. Es wird nur ein Wavetable-Oszillator verwendet. Die Länge der Wavetable wird nur vom Speicher der Hardware beschränkt

Auf Abbildung 3.5 ist eine Wavetable im WAV-Format aufgeführt. Diese enthält 33 Wellenfor-

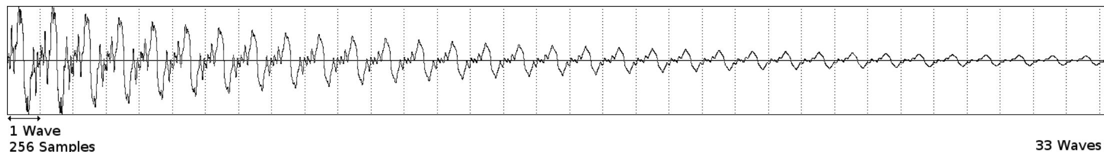

Abbildung 3.5.: Wavetable-Datei

men mit einer Länge von je 256 Samples. Somit ergibt das eine WAV-Datei mit $33 \cdot 256 = 2100$ Samples. Neben der WAV-Datei muss noch die Anzahl der Samples pro Wellenlänge bekannt sein um eine Wavetable zu nutzen. Die Wavetables werden von dem Wavetable-Synthesizer aus Header-Dateien geladen, da auf ein externes Speichermedium verzichtet wurde. Es wurde dazu ein Tool geschrieben welches Wavetables im WAV-Format in einen c-Header mit Array umwandelt.

Die Wavetable kann in x- und in y-Richtung navigiert werden. Um durch die Wavetable innerhalb der einzelnen Waves (x-Richtung) zu navigieren wird eine Schrittweite genutzt um welcher der Index mit der Frequenz der SAMPLE_RATE erhöht wird. WAVE_SIZE ist die Größe (in Samples) einer Wellenform. Die Schrittweite für eine Note mit der Frequenz f_n wird dann wie folgt berechnet:

$$step_x = \frac{WAVE_SIZE \cdot f}{SAMPLE_RATE}$$

Jede $SAMPLE_RATE^{-1}$ Sekunden wird der index_x inkrementiert:

$$index_x = index_x + step_x$$

Das Sample für die Wellenform 'W' an der Stelle index_x wird dann so berechnet:

$$W(\lfloor step_x \rfloor) \cdot (\lceil step_x \rceil - step_x) + W(\lceil step_x \rceil) \cdot (step_x - \lfloor step_x \rfloor)$$

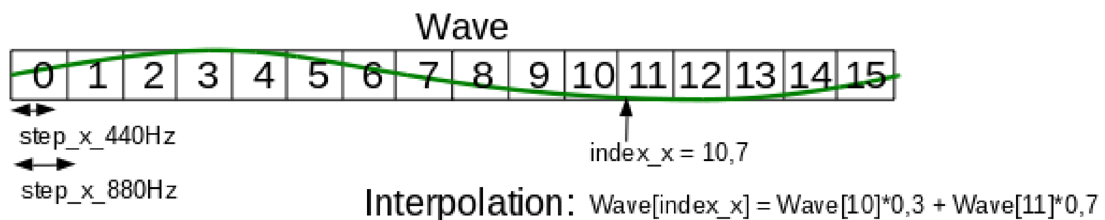

Abbildung 3.6.: Beispiel für die Schrittweiten und Interpolation bei einer Wavetable in x-Richtung

Für die Bewegung zwischen den verschiedenen Wellenformen (y-Richtung) kann eine feste Zeit t_w festgelegt werden, so dass ein Durchlauf der Wavetable bei allen Frequenzen mit der selben Geschwindigkeit passiert. Die Zeit t_w gibt die Dauer für einen Durchlauf durch die gesamte Wavetable an. WAVETABLE_SIZE ist die Anzahl der Wellenformen in der Wavetable. Die Schrittweite für y ergibt sich somit aus:

$$step_y = \frac{WAVETABLE_SIZE}{t_w \cdot SAMPLE_RATE}$$

Abbildung 3.7.: Beispiel für die Schrittweiten und Interpolation bei einer Wavetable in y-Richtung

Es sind 2 Modi vorhanden: Mit Interpolation zwischen den verschiedenen Wellenformen und ohne. Die Interpolation zwischen Wellenformen ermöglicht es mit geringem Speicherverbrauch

eine Wellenform zur nächsten zu transformieren. Allerdings erhöht das den Rechenaufwand erheblich. Deshalb ist die Variante ohne Interpolation zwischen 2 Wellenformen auch interessant, denn Sie hat den Vorteil dass komplexe Klänge mit wenig Rechenaufwand erzeugt werden können.

Es ist möglich die Wavetable mit verschiedenen Strategien zu durchlaufen. Es werden drei Mögliche Strategien vorgestellt:

Anlauf-Sustain Bei dieser Strategie gibt es ein Anlauf-Segment und eines Sustain-Segment. Im Sustain-Segment wird dann vom Anfang bis Ende, und wieder zurück gelaufen.

Loop Eine weitere Möglichkeit ist das durchlaufen der Wellenformen mit einer Schleife.

Zufall Hierbei wird die aktuelle Wellenform durch einen Zufallsgenerator bestimmt. Dabei sollte zwischen den Wellenformen interpoliert werden, da der Übergang sonst nicht glatt ist.

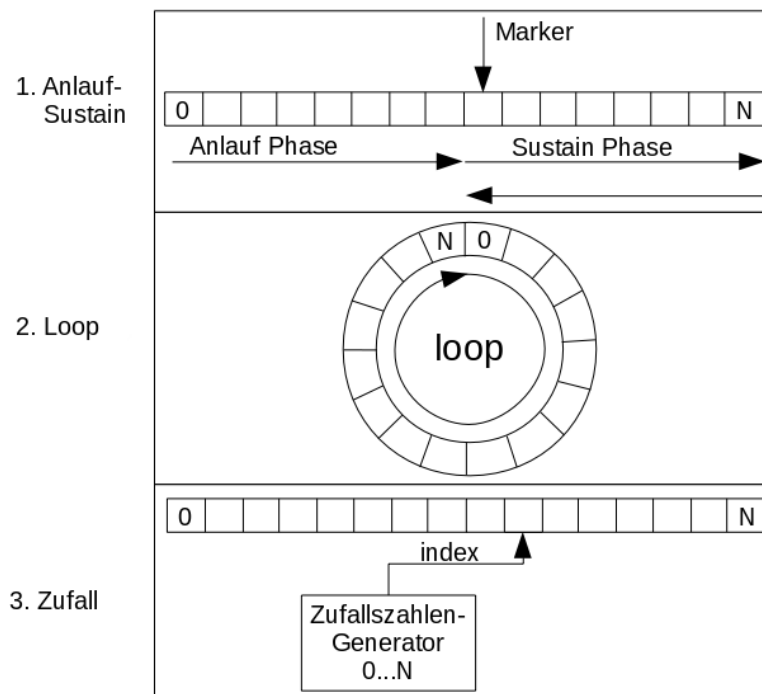

Abbildung 3.8.: Wavetable-strategie

Die Wavetables können entweder durch das Wavegenerator-Modul erzeugt werden, mit welchem die Möglichkeit besteht verschiedene einfache Wellenformen zu erzeugen und diese zu mischen. Oder die Wavetables können von externen Tools erzeugt werden. Zwei Beispiele sind blofeld_wavetable_creator und Audio-Term. Der blofeld_wavetable_creator ist für den Wavetable-Synthesizer Blofeld von Waldorf konzipiert. Es ist möglich Wavetables mit verschiedenen Bitbreiten und Größen zu exportieren. Es lassen sich durch Techniken zum manipulieren von Grundwellenformen synthetisch klingende Wavetables erstellen. Audio-Term erlaubt das laden von Samples, wie z.B. einen Pianoton. Diese Samples lassen sich bearbeiten und können als Wavetable mit einstellbarer Größe und Bitbreite exportiert werden. Das Ergebnis ist oft ein Klang der Charakteristiken von dem Original-Sample enthält, mit einer leicht veränderten Klangfabe.

3.5.2. Modulations-effekt

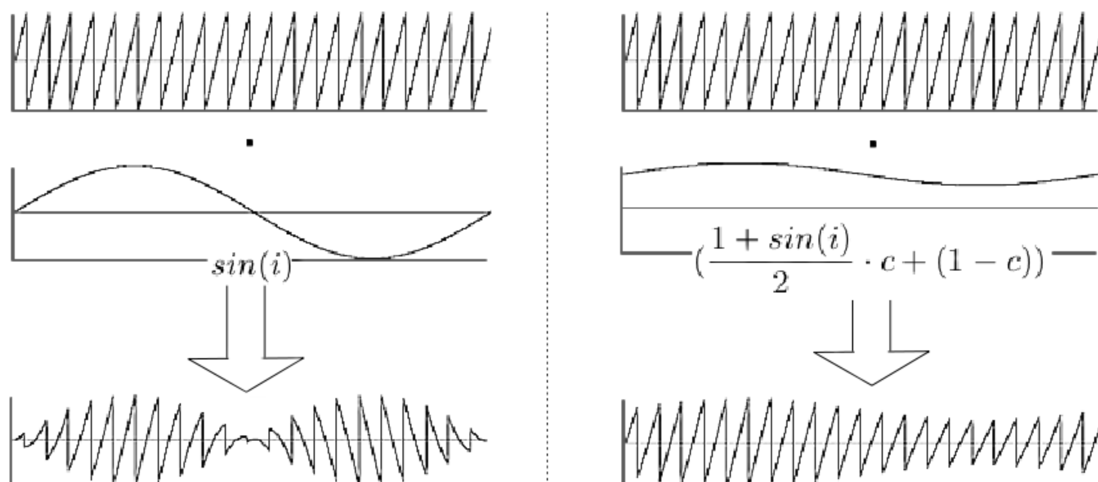

Abbildung 3.9.: Ringmodulator und Tremolo

Als Beispiel für einen Effekt-Plugin wurde ein generischer Modulator-Effekt implementiert. Mit dieser Technik lassen sich verschiedene Effekte realisieren.

1. Ein möglicher Effekt ist der Ringmodulator. Er besitzt einen Modulator beliebiger Länge, der mit einem Index in beliebiger Schrittweite im Kreis durchlaufen wird. Die eingehenden Samples werden mit den Modulator-Sample am aktuellen Index multipliziert.

2. Ein weiterer möglicher Effekt ist das Tremolo. Dazu wird als Modulator ein Sinus oder Dreieckssignal verwendet, welches eine Amplitude von 0 bis 1 hat. Dadurch wird ein

glattes periodisches Schwanken zwischen Maximaler Amplitude und Stille erreicht. Um ein weniger extremes Resultat zu erhalten existiert ein Intensity-Parameter, mit welchem das Originalsignal zu einem bestimmten Verhältnis zugemischt werden kann. Die Schrittweite ist ein weiterer Parameter der dynamisch angepasst werden kann. Dieser bestimmt die Geschwindigkeit des Tremolos.

3.5.3. Faltungshall

Die Faltung ist eine rechenintensive Operation. Eine Faltung von einem 1 Sekunde langem Signal mit einer 1 Sekunden langen Impulsantwort bei einer Samplefrequenz von 48kHz braucht $(48k)^2$ = 2304M Multiplikationen. Der ARM cortex m3 besitzt laut Datenblatt eine Rechenleistung von 1.25 DMIPS/MHz und der LPC1769 taktet mit 120 MHz, damit sind bis zu 150M Instr/s möglich. Somit würde eine Bearbeitung eines 1 Sekunden Samples ca 15 Sekunden auf dem ARM cortex m3 dauern. Mit einem Fast-convolution-Algorithmus lässt sich die Rechenzeit signifikant verbessern, allerdings entsteht dadurch auch eine Latenz, wie im Grundlagenkapitel beschrieben.

Ein weiteres Problem stellt die Speichernutzung dar. Eine 1 Sekunde langes unkomprimiertes Audiosignal mit einer Samplefrequenz von 48kHz ergeben 48000 Samples. Bei einer Auflösung von 16 Bit (2 Byte) ergibt das 96 kByte Daten. Der LPC1769 besitzt jedoch nur 64 kByte RAM-Speicher. Es ist somit nicht möglich eine Faltung mit einer Impulsantwort von 1 Sekunde oder länger durchzuführen, da für fast-convolution-Algorithmen mindestens die Impulsantwort im RAM-Speicher sein muss. Weiterhin sind Impulsantworten oft länger als 2 Sekunden, wenn interessante Halleffekte erzielt werden sollen.

Somit kommt eine Implementierung auf dieser Plattform nicht in Frage. Es wurde dennoch eine Version des Faltungshall entwickelt, welche auf der linux-Version läuft. Die Techniken die dafür eingesetzt wurden, können unter Umständen für eine spätere Version des Frameworks genutzt werden.

Der Ablauf der Faltung wird durch folgende Schritte beschrieben. Die Latenzdauer lässt sich zur Compile-Zeit einstellen. Die Größe der Latenz muss kleiner als die Hälfte der Impulsantwort sein.

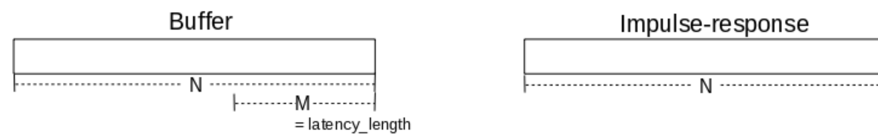

Abbildung 3.10.: Arrays-längen des Faltungsalgorithmus

1. init: f_impulse_respone = fft(impulse_response)
2. Rechtes Buffer-Ende mit M Audio-Samples füllen
3. f_buffer = fft(buffer) und f_erg = f_buffer · f_impulse_response
4. erg = ifft(f_erg). Die letzten M Samples sind das Ergebnis
5. verschieben der samples im Buffer um M nach links. Wiederhole Schritt 2

4. Implementierung

In diesem Kapitel gibt es einen Einblick in die Umsetzung der im Rahmen dieser Bachelorarbeit implementierten Software-Komponenten.

4.1. Fremde Bibliotheken

Wie im Grundlagen-Kapitel beschrieben werden Audio-Ausgabe und MIDI-Eingabe von MIOS genutzt. Der MIDI-Treiber stellt eine Funktion zur Verfügung, welche bei Empfang einer MIDI-Nachricht aufgerufen wird. Als Parameter werden der MIDI-Port und die Nachricht in speziellen Strukturen übergeben. Der Parameter 'port' gibt die Quelle der Nachricht an (UART1, UART2, USB, ...). Der Parameter 'midipackage' enthält die Daten der MIDI-Nachricht.

```
void APP_MIDI_NotifyPackage(mios32_midi_port_t port,
                            mios32_midi_package_t midi_package)
```

Der DAC-Treiber bietet ein Callback-Mechanismus an. Hier kann ein Buffer beliebiger Länge und eine Callback-Funktion angegeben werden. Diese Callback-Funktion wird periodisch aufgerufen wenn neue Samples benötigt werden. Der Buffer wird in 2 Hälften unterteilt, und die Hälften müssen abwechselnd gefüllt werden. Die Samples werden dann im i2s-Format auf den DAC geschrieben.

```
s32 MIOS32_I2S_Start (u32 *buffer, u16 len, void *_callback)
```

Die Test- und Entwicklungsplattform für das Entwickelte Plugin-Interface war ein Linux-System. Deshalb wurde das Framework dafür portiert, wobei die Bibliotheken 'RtMidi', 'ALSA' und 'libsndfile' genutzt wurden um die MIOS-Treiber zu ersetzen. Der MIDI-Treiber wurde hier durch 'RtMidi', und der DAC-Treiber durch 'ALSA' und 'libsndfile' ersetzt.

ALSA (Advanced Linux Sound Architecture) stellt Treiber für die Audio Ein- und Ausgabe für Linux bereit. Damit kann das Audiosignal auf dem Linux-system ausgegeben werden.

libsndfile ist eine c-Bibliothek für das schreiben und lesen von Audio-Dateien. Als Plattformen werden Linux, Mac OS und Windows unterstützt. Sie konvertiert automatisch zwischen

verschiedenen Audio-Formaten und verbirgt auch andere Details vor dem Anwender der Bibliothek. Sie ermöglicht ein Abspeichern in eine Datei für nachfolgende Analysen. Somit lassen sich die Funktionalitäten von Oszillatoren und Interpolation gut testen, da dadurch die Audiodaten gut visuell dargestellt werden können.

RtMidi ist eine Bibliothek für Realtime MIDI-Input/Output und unterstützt Linux, Mac OS und Windows. Sie wird dafür genutzt MIDI-Nachrichten über USB von einem Keyboard zu empfangen.

4.2. Platformabhängige Entscheidungen

Folgend werden Entscheidungen aufgeführt, die Aufgrund begrenzter Rechenleistung und Speicherkapazität der Hardware getroffen wurden. Da keine Floatingpoint-Unit vorhanden ist, und das berechnen von float-Operationen durch Bibliotheken zu rechenaufwändig ist, wurde mit Fixedpoint-Zahlen gerechnet. Da Fixedpoint-Berechnungen als Integer-Operationen ausgeführt werden können ist die Geschwindigkeit der Berechnung deutlich schneller als für Floatingpoint-Berechnungen. Ein Nachteil ist komplexerer Quellcode, da nicht immer sofort ersichtlich ist mit wie vielen Nachkommastellen gerechnet wird, und dies mit Kommentaren gekennzeichnet werden muss. Ein weiterer Nachteil von Fixedpoint ist die geringere Wertebereich, dafür ist aber die Zahl der Nachkommastellen über den gesamten Wertebereich gleich, was bei Floatingpoint nicht der Fall ist.
Die Fixedpoint-Operationen werden hier am Beispiel der linearen Interpolation erläutert. Die Folgende Zeile zeigt eine lineare Interpolation mit Floatingpoint-Zahlen. 'ratio' hat einen Wertebereich von [0,1].

```
1 interpolated_value = value1*(1-ratio) + value2*ratio.
```

Die Nächste Zeile zeigt die gleiche lineare Interpolation, dieses mal aber mit Fixedpoint-Zahlen. 'ratio' hat einen Wertebereich von [0,U16_MAX]. Da aus der Multiplikation des 'value1'(16 bit) mit dem 'ratio'(16 bit) eine 32-bit-Zahl entsteht, muss um 16 Stellen nach rechts geschoben werden, damit wieder eine 16-bit-Zahl vorliegt.

```
1 interpolated_value =
2 (((value1 * (U16_MAX - ratio)) >> 16) + ((value2 * (ratio)) >> 16))
```

Eine Eigenschaft welche von der Rechenleistung beschränkt wird ist die maximal mögliche Polyfonie. Diese unterscheidet sich je nach implementiertem Klangerzeuger und Effekten. Für die Wavetable und Modulation ergab sich durch Messen mit dem Oszilloskop eine maximale

Polyphonie von 8.
Als Buffergröße für die Audioausgabe wurde 64(Stereo-Samples) gewählt. Somit muss bei einer Sample-Frequenz von 48000Hz jede 1333 μs die Buffer-Füll-Funktion aufgerufen werden.

$$\frac{buffer_size}{sample_rate} = \frac{64}{48000Hz} = 1333\mu$$

Es wurden mit einem Oszilloskop Messungen durchgeführt um die aktive Rechenzeit innerhalb dieser 1333 μs zu bestimmen. Dafür wurde ein GPIO-Pin bei jedem Eintritt in die Füll-Funktion auf 1 und vor jedem return auf 0 gesetzt.

Polyphonie	Framework ohne Plugins	Wavetable + Modulations-effekt
0	34 μs (2.5 %)	108 μs (8.1 %)
1	72 μs (5.4 %)	270 μs (20.3 %)
2	96 μs (7.2 %)	420 μs (31.5 %)
4	136 μs (10.2 %)	720 μs (54.0 %)
6	176 μs (13.2 %)	1000 μs (75.0 %)
8	216 μs (16.2 %)	1298 μs (97.4 %)

Aus den Messergebnissen kann geschlossen werden, dass bei den Plugins Wavetable-Synthesizer und Modulations-Effekt eine Maximale Polyphonie von 8 möglich ist. Diese muss in den Header 'constants.h' eingetragen werden. Die Maximal mögliche Polyphonie ist abhängig von den verwendeten Plugins und muss individuell angepasst werden, um unvorhersehbare Effekte zu vermeiden.

4.3. Input-manager

Um Speicher- und Recheneffizienz zu erhöhen, wird die Maximale Polyfonie zur Compilezeit festgelegt. Damit kann dann ein Array über Noten-Informationen von der Größe der maximalen Polyfonie erstellt werden. Weiterhin gibt es ein Array über Notenpointer um die Anordnung der Noten-Informationen dynamisch zu halten. Das Array über die Noten-Informationen fungiert als Speicher-Ressource, und das Pointerarray als dynamische Liste. In Folgender Abbildung wird der Aufbau der Arrays und deren Zusammenhang gezeigt.

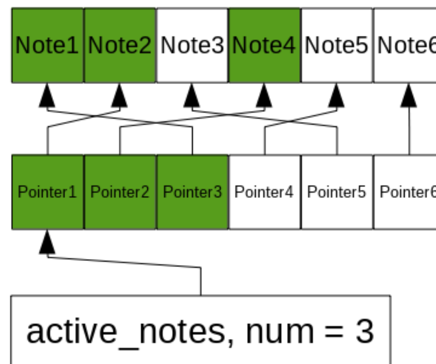

Abbildung 4.1.: Aufbau der Notenliste im Input-Manager

Monophonic-Modus

Für den Monophonic-Modus muss eine Note schon entfernt werden, wenn sie in den Release Zustand geht und es noch andere Noten gibt die nicht im Release Zustand sind. Bei der Funktion get_active_notes wird im Monophonic-Modus nur die aktuellste aktive Note zurückgegeben. Die Hüllkurve wird auch angepasst, und zwar wird sie nicht zurückgesetzt wenn eine neue Note gespielt wird, sondern fängt dort an wo die letzte aktive Note aufgehört hat.

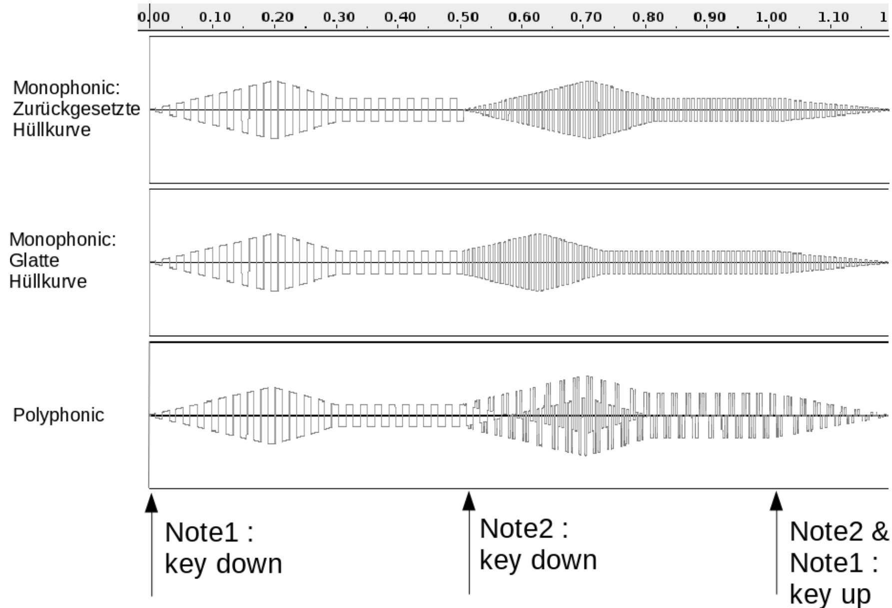

Abbildung 4.2.: Anpassung der Hüllkurve bei monophonen Synthesizern

4.4. CC-map

Wie im letzten Kapitel beschrieben ermöglicht es die CC-map Klangerzeugern und Effekten auf Controll-Change MIDI-Nachrichten zu reagieren. Dazu muss bei der Initialisierung ein Funktionspointer mit einer Controller-Nummer registriert werden. Bei Eintreffen einer Controll-Change MIDI-Nachricht wird die registrierte Funktion aufgerufen. Da es nur 127 verschiedene Controller nummern gibt, kann in der Implementierung direkt ein Array über Funktionspointer genutzt werden, in welcher der Index die Controller-Nummer ist.

```
static void (*map[NUM_CC_KEYS])(u8, u8);
```

4.5. Wavetable-Synthesizer und Modulationseffekt

Wie im letzen Kapitel erwähnt, wird nur ein Oszillator genutzt. Der Wavetable-Oszillator wird mit dem Modul 'wavetable' implementiert. In dem Wavetable-Modul wird je nach Einstellung eine lineare Interpolation zwischen 2 Samples einer Wave oder eine bilineare Interpolation zwischen 4 Samples aus 2 Waves berechnet.
Das Wavetable-Modul besitzt eine Struktur für den Oszillator. In diesem gibt es ein zweidimensionales Array 'table' welches die Wavetable repräsentiert, wobei die erste Dimension (Anzahl der Wellenformen) dynamisch ist und über die Variable 'size' angegeben wird. Weiterhin gibt es Variablen für die aktuelle Position in der Wavetable für jede Note ('pos_x', 'pos_y'). Zum Schluss gibt es noch die Variablen für die Strategie zum Durchlaufen der Wavetable.

```
struct WaveTable {
        s32 (*table)[WAVE_SIZE];
        int   size;
        u32 pos_x[FREQTABLEN];
        u32 pos_y[FREQTABLEN];
        s8  pos_y_direction[FREQTABLEN];
        u32 pos_y_step;
        u16 pos_y_turning_point;
};
```

Folgend ist die Füll-Funktion des Wavetable-Synthesizers aufgeführt:

Zeile 3: Daten für den Synthesizer werden auf die passende Struktur gecastet.

Zeile 8-10: Lautstärke der Note zu diesem Zeitpunkt wird berechnet aus Velocity und Hüllkurve.

Zeile 11-13: Sample für eine Note wird von Wavetable-Oszillator berechnet.

Zeile 14: Noten-Samples werden mit ihrer Lautstärke gewichtet und zusammengemischt.

Zeile 16-19: Schrittweite für Note wird berechnet. Dazu wird zuerst die Aktuelle Frequenz (mit Pitchbend) der Note berechnet.

Zeile 20-21: Der index für die Wavetable wird in x- und y-Richtung bewegt.

Zeile 24-26: Auf alle Channels wird das gleiche Sample geschrieben

```
1  void synthesize_sample(struct Note **active_notes, int num_notes,
2                        s32 *values, void *data) {
3      struct wavetable_data *wdata = (struct wavetable_data *)(data);
4  
5      s32 mixed_sample = 0;
6      int k;
7      for(k=0;k<num_notes;k++) {
8          u16 multiplyer = ((im_note_get_adsr_vol(
9                                  active_notes[k])/127)
10                         * active_notes[k]->velocity);
11         s32 raw_sample = wavetable_get_value(
12                         active_notes[k]->number,
13                         &(wdata->wavetable));
14         mixed_sample +=  (multiplyer * raw_sample)>>16;
15 
16         u32 step = ((WAVE_SIZE*get_freq_with_pitchbend(
17                     active_notes[k]->number,
18                     wdata->pitch_bend_range)) /
19                     (SAMPLE_RATE>>4)) << 2;
20         wavetable_move_pos(active_notes[k]->number,
21                             step, &(wdata->wavetable));
22     }
23     int chn;
24     for(chn=0;chn<CHANNELS;chn++) {
25         values[chn] = mixed_sample;
26     }
27 }
```

Folgend ist die Verarbeitung-Funktion des Modulations-Effektes aufgeführt:

Zeile 4-9: Interpolation des Modulator-Wertes am aktuellen Index.

Zeile 13: Ausgangssample wird Moduliert.

Zeile 14-15: Ausgangssample und Moduliertes Sample werden im Verhältnis des Intensity-Parameters zusammengemischt.

Zeile 18-21: Index für Modulation wird abhängig vom Geschwindigkeits-Parameter erhöht und bei überschreiten der Array-Grenze zurückgesetzt.

```
1  static void modulation_effekt(s32 *values, void *data) {
2      struct mod_data *mdata = (struct mod_data*) data;
3  
4      s32 m1 = mdata->modulation[mdata->index_mod>>16];
5      s32 m2 = mdata->modulation[((mdata->index_mod>>16) + 1)
6               >= mdata->modulation_size ? 0
7               : ((mdata->index_mod>>16) + 1)];
8      u16 prop   = INTPOL_PROP(mdata->index_mod);
9      s32 mod_val = INTPOL_LINEAR(m1, m2, prop);
10 
11     int chn;
12     for(chn=0;chn<CHANNELS;chn++) {
13         s32 modded_val = (values[chn]*mod_val)>>16;
14         values[chn] = INTPOL_LINEAR(values[chn], modded_val,
15                                     mdata->mod_intensity);
16     }
17 
18     mdata->index_mod += mdata->mod_speed;
19     if(mdata->index_mod>=(mdata->modulation_size<<16)) {
20         mdata->index_mod -= (mdata->modulation_size<<16);
21     }
22 }
```

5. Zusammenfassung und Ausblick

Es wurde ein Plugin-Interface entwickelt womit Synthesizer und Effekte für einen arm-cortex-Microcontroller entwickelt werden können. Es wurden Beispiele implementiert welche die Nutzung der Schnittstellen demonstrieren. Als Synthesizer wurde ein simpler Wavetable-Synthesizer entwickelt und als Effekt ein Modulations-Effekt.

Für eine nächste Version dieses Frameworks wäre es Sinnvoll auf einen Microprozessor umzusteigen der eine Floatingpoint-Unit besitzt (bspw. ARM cortex-m4), um den Code zu vereinfachen und mit einem größerem Wertebereich rechnen zu können. Es könnten noch einige weitere Features hinzugefügt werden. Möglich wären:

- Verwendung von Analog/Digital-Input-MIDIbox-Module um die Parameter der Plugins über ein dediziertes Hardwareinterface zu steuern.
- Schnittstelle für ein grafisches Kontrollinterface für die Plugins, welches auf dem Display-Modul der MIDIbox angezeigt werden könnte.
- Portamento-Support.

Es besteht noch ein Problem mit dem genutzen DAC PCM1725. Es wird ein Rauschsignal ausgegeben, selbst wenn am Eingang nur Stille anliegt. Die Ursache könnte eine zu kleine Spannung am PCM1725 sein.

A. Anleitung zur Entwicklung von Plugins

In diesem Kapitel wird eine Anleitung zum Entwickeln von Plugins vorgestellt. Es werden Webseiten verlinkt, welche zur Zusammenstellung der Hardware und Installation der benötigten Software benötigt werden. Weiterhin wird beschrieben wie ein Plugin für das im Rahmen dieser Bachelorarbeit entwickelte Framework entwickelt werden kann.

Hardware

Zum Ausführen der Software wird ein MIDIBox-Core-Modul mit zusätzlichem DAC-Modul benötigt. Ein Schaltplan für den Aufbau des MIDIBox-Core-Moduls findet sich hier [6 (2011)].
Für die Audio-Ausgabe gibt es die Möglichkeit folgendes DAC-Modul zu nutzen [7 (2015)].
Für die linux-Version wird ein x86-Rechner mit einer linux-Installation benötigt.

Toolchain und benötigte Bibliotheken

Um das Framework und die Plugins für die MIDIBox kompilieren zu können, muss die MIOS-Toolchain eingerichtet werden. Anleitungen zum installieren der Toolchain befinden sich auf dieser Webseite [3 (2015)].
Zum Kompilieren auf einem Linux-System werden die Header der Bibliotheken 'alsa', 'rtmidi' und 'sndfile' benötigt. Diese sollten sich in der Paketverwaltung der entsprechenden Distributionen befinden.
Zudem wird das im Rahmen dieser Arbeit entwickelte Framework benötigt.

Entwicklung eines Plugins

Zunächst muss ein Modul entwickelt werden, welches alle Funktionen der Plugin-Schnittstelle enthält. Als Beispiel für ein Plugin wird hier ein einfacher Sinus-Oszillator vorgestellt.

Für ein Funktionstüchtiges Plugin muss mindestens die Funktion 'synthesize_sample' für Sample-für-Sample-Verarbeitung oder 'synthesize_samples' für Block-Verarbeitung implementiert werden. Alle anderen Funktionen der Schnittstelle sind optional.

Als ersten Schritt für die Entwicklung von Plugins wird eine Struktur entwickelt, welche die Daten für eine Instanz des Plugins enthält. Dies ist optional, ermöglicht aber mehrere Instanzen des Plugins. Für den monophonen Sinus-Oszillator werden lediglich ein Array für die Sinuskurve und ein Index benötigt. Dieser wird im Header 'sinosc_synth.h' des Moduls abgelegt.

```
struct sinosc_data {
    s32 wave_buffer[WAVE_SIZE];
    s32 index;
};
```

Weiterhin wird eine Initialisierungsfunktion im Header deklariert, welche eine Plugin-Struktur mit den Funktionen und Daten des Plugins füllt. Somit kann der Speicher für die Strukturen außerhalb dieses Moduls reserviert werden, und es können mehrere Instanzen des Plugins gleichzeitig laufen.

```
void init_sinosc_synthesizer(struct Synth_plugin *plugin_s,
                            struct sinosc_data *data);
```

Danach werden die Funktionen der Plugin-Schnittstelle implementiert, welche auf dieser Struktur arbeiten. Zusätzlich gibt es noch die Möglichkeit Callback-Funktionen für Controll-Change-Nachrichten für die Anpassung von Parametern zu entwickeln. Dafür muss die Callback-Funktionen mit 'ccmap_register_function' bei der CC-Map registriert werden. In diesem Beispiel wird die Funktion synthesize_sample implementiert. Das bedeutet, dass Sample-by-Sample-Verarbeitung eingesetzt wird.

Um die Plugin-Daten zu nutzen muss der void-Pointer zuerst gecastet werden. Im nächsten Schritt wird der Wert der Sinuskurve am aktuellen Index in das Ergebnisarray geschrieben, und der Index erhöht. Der Index wird zurückgesetzt wenn das Ende des Arrays erreicht wurde. Die Schrittweite wird vom Modul 'frqtab' aus der Notennummer und der länge des Arrays berechnet. Bei 'index' und 'step' handelt es sich um Fixedpoint(16.16)-Zahlen. Um daraus Ganzzahlen zu machen(Nachkommastellen abschneiden) muss um 16 Stellen nach rechts geschoben werden. Der folgende Code befindet sich in der Datei 'sinosc_synth.c'.

```
static void synthesize_sample(struct Note **active_notes,
                              int num_notes, s32 *values, void *data) {
    struct sinosc_data *d = (struct sinosc_data*)(data);
    if(num_notes>0) {
        s32 val = d->wave_buffer[d->index>>16];
```

```
        int i;
        for(i=0;i<CHANNELS;i++) {
            values[i] = val;
        }
    }
    u32 step = get_step_width(active_notes[0]->number, WAVE_SIZE);
    d->index += step;
    if((d->index>>16)>=WAVE_SIZE) {
        d->index -= (WAVE_SIZE<<16);
    }
}
```

Alle anderen Funktionen müssen auch implementiert werden. Wenn auf diese Funktionen nicht reagiert werden soll werden Sie einfach leer gelassen.

```
static void note_start(struct Note *note, void *data) {
}

static void note_end(struct Note *note, void *data) {
}

static void change_program(int num, void *data) {
}
```

Die Implementierung der Initialisierungsfunktion beschreibt die Daten-Struktur mit einer Sinuskurve, welche vom Modul 'Wavegenerator' berechnet wird. Weiterhin werden die implementierten Funktionen und die Daten-Struktur in die Plugin-Struktur geschrieben. Damit ist das Sinus-Oszillator-Plugin fertig.

```
void init_sinosc_synthesizer(struct Synth_plugin *plugin_s,
                            struct sinosc_data *data) {
    data->index = 0;
    wavegenerator_generate(data->wave_buffer, WAVE_SIZE,
                           WAVEFORM_SINE);
    plugin_s->data = (void*)(data);
    plugin_s->block_processing = 0;
    plugin_s->change_program = &change_program;
    plugin_s->note_end = &note_end;
    plugin_s->note_start = &note_start;
    plugin_s->synthesize_sample = &synthesize_sample;
}
```

Zuletzt muss das Plugin an das Framework übergeben werden. Dies passiert in der 'custom_init.c'. Hier wird Speicher für die Strukturen reserviert. Dann wird die Initialisierungsfunktion des Sinus-Oszillators aufgerufen und das Plugin dem Framework übergeben.

```
static struct Synth_plugin sinosc;
static struct sinosc_data data;

void custom_init() {
    init_sinosc_synthesizer(&sinosc, &data);
    audio_set_synth_plugin(&sinosc);
}
```

Das Makro 'DEBUG_MSG(...)', welches im Header 'audio_framework.h' definiert ist, kann genutzt werden um Debug-Ausgaben zu machen. Für die Linux-version wird ein printf verwendet, bei der MIDIbox-Version spezielle MIDI-Nachrichten, welche von MIOSstudio ausgegeben werden können.

Kompilieren des Plugins

Damit das neu entwickelte Modul auch mit kompiliert wird, muss es in das Makefile eingetragen werden.

Für die MIDIbox-Version muss das Makefile.m modifiziert werden. Es muss lediglich die .c Datei eingetragen werden. Hier wurde das zuvor entwickelte Modul 'sinosc_synth' ans Ende der Variablen 'THUMB_SOURCE' angehängt.

```
THUMB_SOURCE    = app.c \
                  audio_framework.c \
                  custom_init.c \
                  frqtab.c \
                  utils/utils.c \
                  utils/input_manager.c \
                  utils/cc_map.c \
                  synthelements/wavetable.c \
                  synthelements/wavegenerator.c \
                  synthelements/wavetable_synth.c \
                  synthelements/effekt_modulation.c \
                  synthelements/sinosc_synth.c
```

Für das Kompilieren unter linux müssen zwei Makefiles angepasst werden.

1. **synthelements/Makefile** Es muss eine Zeile zur Erzeugung einer .o Datei erstellt werden.

```
sinosc_synth.o: sinosc_synth.c sinosc_synth.h
        gcc -Wall $(DEBUG) $(CFLAGS) -c sinosc_synth.c
```

   Die Erzeugte .o Datei muss als Ziel angegeben werden.

```
all: ... sinosc_synth.o
```

2. **Makefile.d** Hier muss nochmal die .o Datei für den Linker angegeben werden.

```
SYNTH_O_FILES= ... synthelements/sinosc_synth.o
```

Zum kompilieren wird das Skript ./build_app genutzt.

- **./build_app desktop** kompiliert das Plugin und Framework für linux.
  Es wird eine Ausführbare Datei 'project' erzeugt.

- **./build_app midibox** kompiliert das Plugin für die MIDIbox-Hardware.
  Hier wird die Datei 'project.hex' erzeugt, welche auf die Hardware hochgeladen werden kann.

Hochladen auf MIDIbox

Um das Plugin auf der MIDIbox-Hardware zu nutzen, muss es in den Speicher der Hardware geladen werden. Dies wird mit MIOSstudio getan. Zuvor muss aber noch der MIOS-Bootloader hochgeladen werden. Die Vorgehensweise wird auf dieser Webseite [4 (2015)] erklärt.
Wenn der Bootloader installiert ist, kann das kompilierte Framework hochgeladen werden. Dazu muss bei MIOSstudio die 'project.hex' ausgewählt werden. Dies wird man mit dem 'Browse'-Button erreicht. Danach wird mit dem 'Start'-Button das Framework hochgeladen.

Mit MIOSstudio gibt es auch die Möglichkeit das Plugin zu testen. Es ist möglich mithilfe der Keyboard-Simulation MIDI-Nachrichten an die MIDIbox zu schicken. Weiterhin werden Debug-Nachrichten von MIOSstudio angezeigt.

Unterschiede beim Entwickeln eines Effekt-Plugins

Die Entwicklung eines Effekt-Plugins ist analog zu der eines Synthesizer-Plugins. Allerdings muss anstatt der 'synth_plugin' Struktur die 'effect_plugin' Struktur implementiert werden. In der 'custom_init.h' wird das Plugin dann über 'audio_add_effect_plugin' hinzugefügt.

B. Glossar

Block Verarbeitung Signal-Daten werden in einem Block verarbeitet. Dadruch entsteht weniger Overhead (Rechenaufwand), jedoch entsteht eine Latenz.

Call by referece Funktion bekommt beim Aufruf einen Pointer als Parameter. Somit kann der Parameter auch für die Aufrufende Funktion verändert werden.

Callback Eine Callback-Funktion bezeichnet eine Funktion, die einer anderen Funktion als Parameter übergeben und von dieser unter gewissen Bedingungen aufgerufen wird.

Clipping Für das interne Rechnen mit Signal-Daten wird mit einem höherem Wertebereich gerechnet als ausgegeben wird. Wenn beim Rechnen der Wertebereich überschritten wird, muss das Sample für die Ausgabe auf den Wertebereich beschränken (clipping) werden.

Compile-Zeit Einstellungen die zur Compile-Zeit gemacht werden und nicht zur Laufzeit verändert werden können.

DAC (Digital-Analog Converter) Analoge Schaltung, welche ein digitales Signal in ein Analoges wandelt.

DMIPS(Dhrystone million instructions per second) Maßeinheit für die Rechenleistung von Prozessoren.

I2S (Inter-IC Sound) Schnittstelle für die Kommunikation von seriellen digitalen Audiodaten zwischen ICs

Impulsantwort Die Impulsantwort ist das Ausgangssignal eines Systems, bei dem am Eingang ein Impuls-Signal angelegt wird. Für den Faltungshall kann durch Erzeugen eines Testsignals kann die Impulsantwort eines Raumes als Nachhall mithilfe eines Mikrofons bestimmt werden.

Latenz Zeitraum zwischen einem Ereignis und dem Eintreten einer sichtbaren Reaktion darauf. Bei Synthesizern: Zeitraum zwischen Drücken einer Taste und Eintreten des Klangs.

Laufzeit Einstellungen können auch bei laufendem Programm gemacht werden.

Sample-für-Sample Verarbeitung Jedes Signal-Sample wird einzeln verarbeitet. Dadruch entsteht mehr Overhead (Rechenaufwand), jedoch keine Latenz.

Samplerate Freqenz mit der ein Signal ausgegeben wird.

Sperren Eine Sperre ermöglicht den exklusiven Zugriff auf eine Ressource. So können keine Konflikte entstehen.

WAV Dateiformat für Audio-Rohdaten.

Literaturverzeichnis

[1 2008] Russ, Martin: Sound Synthesys and Sampling, Third Edition. (2008)

[2 2002] Zoelzer, Udo: DAFX - Digital Audio Effects. (2002)

[3 2015] : *MIOS32 Tutorials*. 2015. – URL http://www.ucapps.de/mios32_c.html. – Zugriffsdatum: 03.02.2015

[4 2015] : *Installing the MIOS32 Bootloader*. 2015. – URL http://www.ucapps.de/mbhp_core_lpc17.html. – Zugriffsdatum: 03.02.2015

[5 2014] : *FreeRTOS Semaphores*. 2014. – URL http://www.freertos.org/a00113.html. – Zugriffsdatum: 03.02.2015

[6 2011] : *MBHPcore Scematic LPC1769*. 2011. – URL http://www.ucapps.de/mbhp/mbhp_core_lpc17.pdf. – Zugriffsdatum: 03.02.2015

[7 2015] : *I2S Audio DACs*. 2015. – URL http://www.ucapps.de/mbhp_i2s.html. – Zugriffsdatum: 03.02.2015

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 20. Februar 2014 Alexander Sandau