Computer Architektur
Studienarbeit

Emulation des Soundsystems
Game Boy Advance Reverse Engineering

Dominik Scharnagl - Florian Boemmel - Ngoc Luu Tran

bei Nils Weis / Prof. Dr. Hackenberg

16. Mai 2018

Inhaltsverzeichnis

1 Einleitung 1
1.1 Untersuchungsgegenstand . . . 2
1.2 Verwendete Software . . . 2

2 Game Boy Advance 3
2.1 Hardwareumgebung . . . 3
2.1.1 Übersicht der Audio Register . . . 3
2.1.2 Übersicht der Sound Master Register . . . 5
2.1.3 DevkitPro . . . 8
2.1.4 BeLogic . . . 8
2.1.5 MaxMod . . . 8
2.1.6 „Sappy“ . . . 9
2.2 Softwareentwicklung . . . 17
2.2.1 API zur Audioverarbeitung . . . 17
2.2.2 Audiodaten im Speicher . . . 17
2.2.3 Untersuchung des Assemblercodes . . . 18
2.2.4 Sappy - „Audio-Reflektor“ . . . 19

3 Emulation mittels mGBA 20
3.1 Was ist der mGBA? . . . 20
3.2 Emulation des Game Boy Advance . . . 20
3.2.1 Abgrenzung der Untersuchung . . . 20
3.2.2 Start des Emulators . . . 20
3.2.3 Initialisierung des „mCore“ . . . 20
3.2.4 Laden des ROM . . . 20
3.2.5 Starten des ROM . . . 20
3.2.6 Ausführung des ROM . . . 20
3.3 Emulation des Soundsystems . . . 29
3.3.1 Audioverarbeitung im Assembler . . . 29
3.3.2 Weiterverarbeitung im Emulator . . . 29
3.4 Interaktion mit dem Betriebssystem . . . 30
3.4.1 Start des Emulators . . . 30
3.4.2 Einstellungen über die mGBA GUI . . . 31
3.4.3 Starten des ROM . . . 32
3.4.4 Transferirrung der Audiodaten . . . 33

4 Zusammenfassung 34
4.1 Inhalt des Dokumentes . . . 34
4.2 Fazit zur Studienarbeit . . . 34

1 Einleitung

Autor: Florian Boemmel

Der Game Boy Advance zählt zu einer der erfolgreichsten Spielekonsolen der Welt. Der 2001 von Nintendo [1] veröffentlichte Nachfolger des Game Boy Classic findet sich heute noch in den Schubläden der damalilgen Jugend. Deshalb überrascht es auch nicht, dass die Fans der Konsole den Erinnerungen aus ihrer Kindheit neues Leben einhauchen und sogar Emulatoren für diverse Spiele-Klassiker der Plattform entwickeln.

Abbildung 1: Game Boy Advanced - Blue Edition

Der zentrale Inhalt der Studienarbeit, ist das Reverse Engineering eines solchen Game Boy Advance Emulators. Der genaue Inhalt dieser wird in den nächsten Kapiteln zunächst eingeschränkt und später weiter konkretisiert.

Emulatoren gehören zu einem beliebten Werkzeug der Informatik. Sie bilden ein System oder ein Teilsystem ab. Dabei ist zu beachten, dass diese bekanntes Verhalten nur „nachahmen“. Genauer ausgeführt bedeutet dies, dass zum Beispiel bei einem Game Boy Advance Emulator die Software intern anders als auf dem originalen Gerät arbeitet. Jedoch kommt es beim Emulieren nicht auf die gleiche Arbeitsweise an, sondern auf das Ergebnis. In diesem konkreten Fall, einen voll funktionsfähigen Nachbau des Game Boys in Software. Mit dem es möglich ist digitalisierte Versionen eines Spieles spielen zu können.

CPU	16,77 MHz 32 Bit RISC (ARM7TDMI) 8 Bit CISC CPU (Z80/8080-Derivat)
Arbeitsspeicher	32 KB IRAM (1 cycle/32 bit) + 96 KB VRAM (1-2 cycles) + 256 KB ERAM (6 cycles/32 bit)
Lautsprecher	Lautsprecher (Mono), Kopfhörer (Stereo)

Tabelle 1: Technische Daten des Game Boy Advance [3]

1.1 Untersuchungsgegenstand

Autor: Florian Boemmel

In dieser Studienarbeit wird die Fragestellung, wie wird das Soundsystem des Game Boy Advance in einem beliebigen Emulator emuliert, thematisiert. Ein konkreter Emulator wurde nicht vorgegeben. Wir einigten uns demnach auf den Game Boy Advance Emulator „mGBA“. Dieser stellt im Folgenden unseren zentralen Untersuchungsgegenstand dar.

Die Untersuchung wird in vier Unterthemen gegliedert:

- Erstellung eines Beispielprogramms (siehe Abschnitt ?? und Abschnitt ??)
- Untersuchung der Fragestellung mit Hilfe eines Beispielprogrammes (siehe Abschnitt ??)
- Untersuchung der Interaktion des Beispielprogrammes mit dem Emulator (siehe Abschnitt 3.2)
- Untersuchung der Interaktion von Emulator und Betriebssystem (siehe Abschnitt 3.3)

1.2 Verwendete Software

Autor: Florian Boemmel

- **Betriebssysteme**: Ubuntu 16.0 x64, Windows 10 x64, macOS 10.13.4
- **Disassembler**: IDA Pro
- **Emualtor**: mGBA
- **SDK**: devkitPro
- **IDE's**: Programmer's Notepad, Visual Studio Code, Eclipse, Qt Creator

2 Game Boy Advance

Autoren: Florian Boemmel, Ngoc Luu Tran

2.1 Hardwareumgebung

Autor: Florian Boemmel

Der Game Boy Advance verfügt über sechs Soundkanäle. Vier davon wurden, vor allem aus Gründen der Abwärtskompatibilität, aus dem Vorgänger „Game Boy Classic“ übernommen.

Kanal	**Art**
1	Rechteckwellengenerator (square wave generator)
2	Rechteckwellengenerator (square wave generator)
3	Klangerzeuger (Sample-Player)
4	Rauschgenerator (Noise-Generator)
A	Direct Sound
B	Direct Sound

Tabelle 2: Übersicht der Soundkanäle des Game Boy Advance

2.1.1 Übersicht der Audio Register

Autor: Florian Boemmel

Intern besitzt der Game Boy Advance drei Sound-Master-Register. Dort müssen, je nach Einstellungswunsch, ein paar Bits gesetzt werden. Erst dann ist eine Soundwiedergabe oder die generelle Funktionsfähigkeit des Soundsystems möglich. [4]

Der Offset im Folgenden bezieht sich auf die Basisadresse $0x04000000$ und wird in hexadezimaler Schreibweise angegeben. An dieser Stelle muss darauf hingewiesen werden, dass die Bezeichnungen der Register nicht eindeutig sind und sich je nach verwendeter Quelle unterscheiden.

Offset	**Kanal**	**Funktion**	**Bezeichnung**
$0x060$	1	DMG Sweep control	`SOUND1CNT_L`
$0x062$	1	DMG Length, wave and evelope control	`SOUND1CNT_H`
$0x064$	1	DMG Frequency, reset and loop control	`SOUND1CNT_X`
$0x068$	2	DMG Length, wave and evelope control	`SOUND2CNT_L`
$0x06C$	2	DMG Frequency, reset and loop control	`SOUND2CNT_H`

Tabelle 3: Übersicht der Sound-Register - Teil 1

Offset	**Kanal**	**Funktion**	**Bezeichnung**
$0x070$	3	DMG Enable and wave ram bank control	`SOUND3CNT_L`
$0x072$	3	DMG Sound length and output level control	`SOUND3CNT_H`
$0x074$	4	DMG Frequency, reset and loop control	`SOUND3CNT_X`
$0x078$	4	DMG Length, output level and evelope control	`SOUND4CNT_L`
$0x07C$	4	DMG Noise parameters, reset and loop control	`SOUND4CNT_H`
$0x080$		DMG Master Control	`SOUNDCNT_L`
$0x082$		Direct Sound Master Control	`SOUNDCNT_H`
$0x084$		Master Sound Output Control / Status	`SOUNDCNT_X`
$0x088$		Sound Bias	`SOUNDBIAS`
$0x090$	3	DMG Wave RAM Register	`WAVE_RAM0_L`
$0x092$	3	DMG Wave RAM Register	`WAVE_RAM0_H`
$0x094$	3	DMG Wave RAM Register	`WAVE_RAM1_L`
$0x096$	3	DMG Wave RAM Register	`WAVE_RAM1_H`
$0x098$	3	DMG Wave RAM Register	`WAVE_RAM2_L`
$0x09A$	3	DMG Wave RAM Register	`WAVE_RAM2_H`
$0x09C$	3	DMG Wave RAM Register	`WAVE_RAM3_L`
$0x09E$	3	DMG Wave RAM Register	`WAVE_RAM3_H`
$0x0A0$	A	Direct Sound FIFO	`FIFO_A_L`
$0x0A2$	A	Direct Sound FIFO	`FIFO_A_H`
$0x0A4$	B	Direct Sound FIFO	`FIFO_B_L`
$0x0A6$	B	Direct Sound FIFO	`FIFO_B_H`

Tabelle 4: Übersicht der Sound-Register - Teil 2

Die in Tabelle 3 und in Tabelle 4 gelisteten Register sind im mGBA als Felder der Enumeration *GBAIORegisters* (*$/include/mgba/internal/gba/io.h*) gelistet und entsprechend ihrer Registeradressen belegt. Sie werden unter anderen zur Adressierung des emulierten Speichers verwendet. Als Quelle für die beiden Tabellen diente neben der *io.h* auch die Webseite `http://belogic.com/gba/`, Stand Juni 2018.

2.1.2 Übersicht der Sound Master Register

Autor: Florian Boemmel

Die Register DMG Master Control, Direct Sound Master Control und Master Sound Output Control / Status bilden die Sound Master Register.

DMG Master Control

Hier müssen zunächst einige Bits gesetzt werden, bevor eine generelle Verwendung des Soundsystems möglich ist.

Bit	**Kanal**	**Funktion**	**Bezeichnung**
0	1-4	Left Volume	
1	1-4	Left Volume	
2	1-4	Left Volume	
3			
4	1-4	Right Volume	
5	1-4	Right Volume	
6	1-4	Right Volume	
7			
8	1	Channel 1 Left	`SDMG_LSQR1`
9	2	Channel 2 Left	`SDMG_LSQR2`
A	3	Channel 3 Left	`SDMG_LWAVE`
B	4	Channel 4 Left	`SDMG_LNOISE`
C	1	Channel 1 Right	`SDMG_RSQR1`
D	2	Channel 2 Right	`SDMG_RSQR2`
E	3	Channel 3 Right	`SDMG_RWAVE`
F	4	Channel 4 Right	`SDMG_RNOISE`

Tabelle 5: DMG Master Control Register

Direct Sound Master Control

Dieses Register kontrolliert die Lautstärke der DMG Kanäle und aktiviert diese. Die Einstellungen können separiert voneinander für den linken und rechten Lautsprecher vorgenommen werden.

Bits	**Name**	**Funktion**	**Bezeichnung**
0-1	DMGV	DMG Volume Ratio	
		00: 25%	SDS_DMG25
		01: 50%	SDS_DMG50
		10: 100%	SDS_DMG100
		11: forbidden	
2	AV	Direct Sound A Volume Ratio	
		50% if clear	SDSA50
		100% if set	SDSA100
3	BV	Direct Sound B Volume Ratio	
		50% if clear	SDSB50
		100% if set	SDSB100
4-7			
8	AR	Direct Sound A enable Direct Sound on Right speaker	SDS_AR
9	AL	Direct Sound A enable Direct Sound on Left speaker	SDS_AL
A	AT	Direct Sound A Timer.	
		Use timer 0 (if clear) for Direct Sound A	SDS_ATMR0
		Use timer 1 (if set) for Direct Sound A	SDS_ATMR1
B	AF	FIFO reset for Direct Sound A	SDS_ARESET
C	BR	Direct Sound B enable Direct Sound on Right speaker	SDS_BR
D	BL	Direct Sound B enable Direct Sound on Left speaker	SDS_BL
E	BT	Direct Sound B Timer.	
		Use timer 0 (if clear) for Direct Sound B	SDS_BTMR0
		Use timer 1 (if set) for Direct Sound B	SDS_BTMR1
F	BF	FIFO reset for Direct Sound B	SDS_BRESET

Tabelle 6: Direct Sound Master Control Register

Wenn DMA für Direct Sound verwendet wird, dann wird DMA den FIFO-Puffer zurücksetzen, nachdem er verwendet wurde.

Master Sound Output Control / Status

Aus diesem Register kann zu einem der Status der einzelnen DMG Kanäle ausgelesen werden und zum Anderen die generelle Soundausgabe aktiviert werden. Dazu muss das Bit 7 gesetzt werden.

Bits	**Name**	**Funktion**	**Bezeichnung**
0	1A	Channel 1 is active and currently playing.	`SSTAT_SQR1`
1	2A	Channel 2 is active and currently playing.	`SSTAT_SQR2`
2	3A	Channel 3 is active and currently playing.	`SSTAT_WAVE`
3	4A	Channel 4 is active and currently playing.	`SSTAT_NOISE`
4-6			
7	MSE	Master Sound Enable Must be set if any sound is to be heard at all. Set this before you do anything; otherwise other sound registers can't be accessed (see GBATek for more details).	`SSTAT_DISABLE` `SSTAT_ENABLE`
8-F			

Tabelle 7: Master Sound Output / Status Register

Die Bits 0-3 geben ausschließlich darüber Auskunft, welcher Kanal aktuelle bespielt wird und nicht ob dieser eingeschaltet ist. Zum Ein- und Ausschalten eines Kanals dient das DMG Master Control Register (siehe Abschnitt 2.1.2).

2.2 Plattformen

Autor: Ngoc Luu Tran

2.2.1 DevkitPro

Autor: Ngoc Luu Tran

DevkitPro ist eine Organisation welche sich auf Cross-Compiler für beliebte Videospielkonsolen spezialisiert hat.

Das Ziel dabei ist es Hobby und Amateur Videospielentwicklern eine Plattform zu bieten, in welcher sie wertvolle Erfahrungen im Programmieren für Ressourcen limitierte Geräte sammeln können. Im Idealfall können sie die gesammelten Erfahrungen dann auf eine Karriere in der Videospielentwicklung übertragen.

Angefangen hat alles im Jahre 2003 mit dem Cross-Compiler namens devkitARM, welcher es ermöglichte Spiele für den Gambe Boy Advance zu entwickeln. Mittlerweile umfasst das Angebot nicht nur den Gambe Boy Advance, sondern auch den Nintendo GameCube, Nintendo Wii, GP32, Nintendo DS, GP2X und die Nintendo Switch.

DevkitARM ist dabei nicht nur die erste Toolchain für Videospielkonsolen, sondern sie ist bis dato auch die beliebteste unter den Toolchains. So kammen über die Jahre immer weitere ARM basierende Konsolen hinzu und entwickelte sich somit allgemein zu einen der besten Windows basierenden Entwicklertools für ARM-Geräte.

2.2.2 BeLogic

Autor: Ngoc Luu Tran

2.2.3 MaxMod

Autor: Ngoc Luu Tran

2.2.4 „Sappy“

Autor: Ngoc Luu Tran

Die von Nintendo bereitgestellte Sound-Engine, welche von der Mehrheit der kommerziellen Game Boy Advance Spielen genutzt wird, wird in der Rom-Hacking-Szene Sappy genannt. Der Name der Engine stammt von einem Programm namens Sappy, welches Musik aus den Game Boy Advance Spielen extrahiert und diese in MIDI-Dateien konvertiert.

Videospielentwickler komponieren ihre Musik als Standard MIDI-Datei oder als Trackermodul und konvertieren diese dann mithilfe eines Programms in das „Sappy-Format“. Dieses Format ist sehr ähnlich zu der MIDI-Datei und besteht unter anderem aus Key-On, Key-Off und Delta-T Werten. Jedoch wird das Sappy-Format effizienter gespeichert und wurde speziell für Videospiele entwickelt um zusätzlich noch sowas wie Soundeffekte abzuspielen.

Die Daten werden normalerweise in folgender Reihenfolge auf dem GBA-ROM gespeichert:

- Definitionen der Instrument Banks
  - Sampled Instrument, Programmable-Sound-Generator (PSG) Instrument
- Key-split Instrument Daten
  - Key Split, Every Key-Split
- Gameboy Channel 3 Waveform Daten
- Track-Group RAM Zeiger
- Musik/SFX Zeiger
- Samples
- Daten für jeden Track
- Zeiger auf jeden Track

Definition der Generischen Instrumenten Daten

HEX-Zahl	Instrumenten-Typ
0x00	Sample (GBA Direct Sound channel)
0x01	Rechteckwellengenerator (Game Boy channel 1)
0x02	Rechteckwellengenerator (Game Boy channel 2)
0x03	PSG Programmable Waveform (Game Boy channel 3)
0x04	Rauschgenerator (Game Boy channel 4)
0x08	Sample (GBA Direct Sound channel) that is never resampled (always playing at the engine's rate)
0x09-0x0C	selbige wie 0x01-0x04
0x40	Key split instrument : Points to different sub-instruments depending on MIDI key
0x80	Every Key split instrument / percussion : Each MIDI key points to its own sub-instrument anything else = invalid (the engine crashes)

Tabelle 8: Übersicht des Instrumententypes

Jede Definition von einem Instrument oder Sub-Instruments besteht aus 12 Bytes, dabei gibt das erste Byte an um welchen Typen von Instrument es sich handelt.

Sampled Instrument / Sub-Instrument Format

Byte-Größe	Definition
1	0x00 (normal) oder 0x08 (nicht resampled)
1	MIDI key (only used as percussion sub-instrument)
1	nicht genutzt (immer 0)
1	panning (only used as percussion sub-instrument). If bit 7 is set, the lower 7 bits forces the panning value for this key. Otherwise, the channel's panning is used
4	Pointer zu Sample Dateien
1	Attack-Wert (8-bit : 0x01 = längster Attack, 0xFF = kein Attack)
1	Decay-Wert (8-bit : 0x00 = kein Decay, 0xFF = längster Decay)
1	Sustain-Level (8-bit : 0x00 = Sustain zu Stumm, 0xFF = Sustain zur vollen Lautstlärke)
1	Release-Wert (8-bit : 0x00 = sofortiger Release, 0xFF = längster Release)

Tabelle 9: Sampled Instrument / Sub-Instrument Format

Um ein Sampled Instrument unverändert, also ohne Anwendung von Attack, Decay, Sustain und Release abzuspielen, werden die Werte0xFF, 0x00, 0xFF, 0x00 verwendet.

PSG Instrument / Sub-Instrument Format

Byte-Größe	Definition
1	PSG Channel (0x01 = square 1, 0x02 = square 2, 0x03 = programmierbare waveform, 0x04 = noise)
1	MIDI Key (wird nur als Percussion Sub-Instrument verwendet)
1	Hardware Time length Control (0x00 um Time length zu deaktivieren)
1	Sweep Control (nur Square 1 Channel, 0x08 um Sweep zu deaktivieren)
4	Square Channel: 1 Byte = Duty Cycle (0=12,5%, 1=25%, 2=50%, 3=75%) 3 Bytes = 0x0000 Noise Channel: 1 Byte = Steuert Noise's Periode (0 = Normal (32767 Samples), 1 = Metallic (127 Samples)) 3 bytes = 0x0000 Programmierbarer Channel: 4 bytes = Zeiger auf 16-Byte Waveform-Datei
1	Attack-Wert (8-bit : 0x01 = längster Attack, 0xFF = kein Attack)
1	Decay-Wert (8-bit : 0x00 = kein Decay, 0xFF = längster Decay)
1	Sustain-Level (8-bit : 0x00 = Sustain zu Stumm, 0xFF = Sustain zur vollen Lautstlärke)
1	Release-Wert (8-bit : 0x00 = sofortiger Release, 0xFF = längster Release)

Tabelle 10: PSG Instrument / Sub-instrument Format

Um ein PSG Instrument unverändert, also ohne Anwendung von Attack, Decay, Sustain und Release abzuspielen, werden die Werte0xFF, 0x00, 0xFF, 0x00 verwendet.

Square & Noise Channels:
Da die Hüllkurve durch das ansteigen bzw. verringern des Hardware Volume Registers bestimmt wird, unterscheidet die sich grundsätzlich von den Software emulierten Attack, Decay, Sustain und Release des Sampled Instrument.

Ein Beispiel hierfür währe wenn Volume auf 6 ist, wird ein Attack von „3“ doppelt so schnell ausgeführt wie der Attack von „3“ wenn Volume 12 ist. Dies geschieht, weil Volume unabhängig vom Volume Wert, in der gleichen Rate ansteigt.

Zusätzlich kommt zu diesem Problem hinzu, dass bei jeder Änderung von Volume in den Tracks sich Attack, Decay, Sustain und Release zurücksetzen. Dies wiederum setzt den Sound zurück, welches in ein „klick“ Sound resultiert. Das kann auch in Spiele, welche Musik Volume ein und aus blenden, gehöhrt werden. Aus diesem Grund sollten Volume-Änderungen vermieden werden und stattdessen falls möglich Attack, Decay, Sustain und Release verwendet werden.

Programmierbarer Waveform Channel:
Die Hüllkurve wird mit den vier Level welche für Volume zuverfügung steht (25%, 50%, 75% and 100%) von der Software simuliert. Deswegen kann die Hüllkurve sehr abgehackt und grob klingen, da nur sowenig Level zuverfügung stehen. Jedoch leidet dieser Channel nicht unter dem Problem des Klickens bei Volume-Änderung, wie bei den Square & Noise Channels.

Every/Key-Split Instrumente

Byte-Größe	Key Split Definition	Every Key-Split Definition
1	0x40	0x80
3	0x00	0x00
4	Zeiger auf ersten Sub-Instrument	Zeiger auf Percussion Tabelle
4	Zeiger auf Key Split Tabelle	0x00

Tabelle 11: Every/Key-Split Instrumente

Key-Split:
Die Key-Split Tabelle hat eine Länge von 128 Bytes und gibt Informationen darüber welches Sub-Instrument bei welchem MIDI-Key (1 Byte) genutzt wird. Die Position des benutzten Sub-Instrumentes ist:
12 * KeySplitTable[MidiKey] + Zeiger auf ersten Sub-Instrument

Every Key-Split:
Die Position des benutzten Sub-Instrumentes ist:
12 * MidiKey + Zeiger auf Percussion Tabelle.
Der MIDI-Key und das Panning des spezifizierten Instruments wird nur hier benutzt.

Ungenutztes Instrumente

HEX-Folge von einem ungenutzten Instrument

0x01, 0x3c, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0x00

16-Byte programmaierbare Waveform format

Die Waveform besteht aus 32 4-Bit Puls-Code-Modulation (PCM) Samples, Big Endian, Unsigned (0 = niedriges Level, F = Höheres Level) für Insgesamt 16-bytes pro Waveform.

Zum Beispiel:

0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0x10

Produziert ein Triangle Wave.

Der Game Boy Advance hat ein 64 Sample-Modus, jedoch keine möglichkeit diesen auszulösen.

Track-Gruppen RAM Zeiger Format

Byte-Größe	Definition
4	Zeiger auf Track-Gruppen Variablen, besteht aus 0x40 Bytes
4	Zeiger auf Track Variablen, jeder Track besteht aus 0x50 Bytes
1	Maximale Anzahl von Tracks für diese Gruppe. Falls versucht wird ein Song mit mehr Tracks abzuspielen, werden die überschrittenen Tracks ignoriert und nciht abgespielt
3	0x00, 0x00, 0x00

Tabelle 12: Track-Gruppen RAM Zeiger Format

Die Sappy Sound Engine kann mehrere "Gruppen von Tracks"gleichzeitig wiedergeben, so können Sound-Effekte abgespielt werden während Hintergrundmusik läuft. Verschiedene Spiele handhaben die Gruppen auf verschiedene Wegen.

Apparently when the game starts a song it doesn't know in which groups it is started (it's the song pointers structure who decides it), but afterward he can alter the volume, panning and other settings by refering to the group in which the song plays. It's a bit strangle but apparently this is how the sappy engine works.

Each track in each group have it's own set of pointers and counters used by the sound engine, which should sit somewhere in RAM. The more tracks and track groups are neccesarly for the game, the more RAM it takes up. Each track group takes 12 bytes of pointers at this locations. Remember that IRAM is mapped in the GBA at 0x3000000.

Song Zeiger Format

Byte-Größe	Definition
4	Zeiger auf Song-Header
1	Track Gruppe
1	0x00
1	Track Gruppe
1	0x00

Tabelle 13: Song Zeiger Format

Obwohl die Track Gruppe zweimal vorkommt scheint diese immer Gleich zu sein. Ändert man diese so ab, dass sie sich von einander Unterscheiden, scheint nur die erste Track Gruppe einen Effekt zuhaben.

Sample Format

Byte	Definition
3	0x00, 0x00, 0x00
1	0x00 für nicht geloopte Sample, 0x40 für geloopte Sample
4	Pitch anpassung
4	Loop relativ zum Startpunkt
4	Größe der Sample

Tabelle 14: Sample Format

Jedes Sample kommt mit einem 16-Byte Header gefolgt von einer variablen Datenlänge.

Die 8-Bit signed PCM Sample Datei folgt umgehend.

Der Pitch Adjustement Wert wird mit folgender Formel berechnet:
Pitch Adjustement = 1024 * Sample-rate für Mid-C (Mid-C entspricht dem MIDI-Key #60)

The pitch adjustment equivalent to available engine sampling rates (for mid-C note) are as following :

0x599800 (5734 Hz)
0x7b3000 (7884 Hz)
0xa44000 (10512 Hz)
0xd10c00 (13379 Hz)
0xf66000 (15768 Hz)
0x11bb400 (18157 Hz)
0x1488000 (21024 Hz)
0x1a21800 (26758 Hz)
0x1ecc000 (31536 Hz)
0x2376800 (36314 Hz)
0x2732400 (40137 Hz)
0x2910000 (42048 Hz)

Those values are often used for percussion sounds and sound effects, which are typically played at the engine's sample rate.

If the loop is exactly the size of a single oscillation of a looped waveform, there is an useful shorcut to compute the right pitch: pitch = 267905 (loop_end loop_start)

Track Format

Das Track Format ist das einzige Format welches nicht in 4-Byte Gruppiert wird.

Im Gegensatz zu den meisten anderen Sound-Formaten in Spielen ist Polyphonie, also mehrere Noten gleichzeitig, im selben Track möglich.

Bytes zwischen 0x81-0xB0 sind Delta-T Werte, Bytes zwischen 0xB1-0xFF sind Befehle und Bytes zwischen 0x00-0x7F sind Argumente für Befehle. „Negative“ Argumente 0x80-0xFF sind möglich, jedoch für Gewöhnlich „verboten“.

HEX-Zahl	Definition	Bytes	Wiederholbar
0x80	Wait Tero Time	1	nein
0x81-0xB0	Delta-T, je Höher desto Länger wird gewartet. Siehe Tabelle 16 unten	1	nein
0xB1	Ende des Tracks	1	nein
0xB2	Sprungbefehl auf eine Adresse	4	nein
0xB3	Ruft Subsection auf	4	nein
0xB4	Ende der Subsection	1	nein
0xB5	Ruft und Wiederholt Subsection	5	nein
0xB9	Speicherinhalt bedingter Breakpoint	3	nein
0xBA	Bestimmt Track-Priorität. Höherer Wert entspricht höhere Priorität	1	nein
0xBB	Tempo entspricht Beats per Minute geteilt durch zwei	1	nein
0xBC	Transponiert. Einziger Befehl, welches ein negatives Argument zulässt	1	nein
0xBD	Stellt Instrument ein	1	ja
0xBE	Stellt Lautstärke ein	1	ja
0xBF	Stellt Panning ein	1	ja
0xC0	Tonhöhenänderungs-Wert. 0x41-0x7F und höher ändert es nach oben, 0x00-0x3F ändert es nach unten, 0x40 ist der Original-Ton	1	ja
0xC1	Tonhöhenänderungs-Reichweite. Argument ist die Anzahl von Halbtönen entfernt vom Oringal-Ton bei Änderung zum Maximum bzw. Minimum	1	ja
0xC2	Low Frequency Oscillator (LFO) Geschwindigkeit. Je Höher desto Schneller	1	nein
0xC3	LFO-Verzögerung. Anzahl der Ticks welche runter gezählt werden bevor LFO startet	1	nein
0xC4	LFO-Tiefe. Beeinflusst wie tief der LFO-Effekt ist	1	ja
0xC5	LFO-Typ. Wählt betroffene Variable, welche LFO modifiziert aus. 0 = Pitch (default), 1 = Lautstärke, 2 = Panning	1	nein
0xC8	Verstimmt. 0x40 ist normaler Pitch, 0x00 ist ein Halbton tiefer und 0x7F ist ein Halbton höher	1	ja
0xCE	Note aus	1-2	ja
0xCF	Note an	1-2	ja
0xD0-0xFF	Note an mit automatischem Time-out. Siehe Tabelle 16 unten	1-3	ja

Tabelle 15: Track Format

Die Längen-Tabelle für Note-On und Delta-T Befehle sind wie folgt:

Note	**D0**	**D1**	**D2**	**...**	**E6**	**E7**	**E8**	**E9**	**EA**	**EB**	**EC**	**ED**	**EE**	**EF**	**F0**
Delta-t	81	82	83	...	97	98	99	9A	9B	9C	9D	9E	9F	A0	A1
Länge	1	2	3	...	23	24	28	30	32	36	40	42	44	48	52
Note	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**FA**	**FB**	**FC**	**FD**	**FE**	**FF**
Delta-t	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	B0
Länge	56	56	60	64	66	68	72	76	78	80	84	80	90	92	96

Tabelle 16: Note an mit automatischem Time-out

Für gewöhnlich wird der Song so erstellt, dass die Länge von 96 eine ganze Note ergibt. Demnach ist ein Tick ein drittel der Note und ist die feinere mögliche Lösung.

Die Anzahl von Argumenten bei Befehlen kann zwischen 0 und 3 Variieren. Ein „negativ“ Byte bedeutet, dass die Argumenten-Liste beendet wird und das deshalb der Key-On Befehl komplett dekodiert wurde.

Die „volle“ Version von einem Note-On Befehls benötigt zwei Argumente, Key und Geschwindigkeit. Im Falle vom auto Time-out Note-On Befehl, kann ein optionales drittes Argument vorkommen.

Da die Längen-Tabelle fehlende längen Werte hat, kann ein extra drittes Argument genutzt werden um eine feinere Präzision zu erreichen.

Ein Noten Befehl gefolgt von einem einzigen Argument spielt eine neue Note mit einem neuem Key ab, verwendet jedoch die zuletzt genutzte Geschwindigkeit.

Folgt nach dem Note-On Befehl kein Argument, verwendet die neue Note den zuletzt angewendeten Key und die zuletzt genutzte Geschwindigkeit

Song-Header Format

Byte	**Definition**
1	Anzahl von Tracks
1	Unbekannt
1	Song Priorität
1	Echo feedback
4	Zeiger auf Definition der Instrumente
n*4	Zeiger auf Track-Daten

Tabelle 17: Song-Header Format

Die Priorität wird wie Folgt gehandhabt:

- Bei PSG Channel wird die Note mit der höchsten Priorität wiedergegeben
- Bei Direct Sound Channel wird falls kein freier Channel mehr zu Verfügung steht, die Noten mit den höchsten Prioritäten wiedergegeben und die mit niedrigeren Prioritäten werden ignoriert oder zum Schweigen gebracht um Platz für höher priorisierte Noten zu schaffen.

Im Falle von gleicher Priorität wird nach der Track Nummer entschieden. Die niedrigere Track Nummer hat höhere Priorität als die hohen Track Nummern.

2.3 Softwareentwicklung

Autor: Ngoc Luu Tran

2.3.1 API zur Audioverarbeitung

Autor: Ngoc Luu Tran

2.3.2 Audiodaten im Speicher

Autor: Ngoc Luu Tran

3 Emulation mittels mGBA

Autoren: Dominik Scharnagl, Florian Boemmel

3.1 Was ist der mGBA?

Autor: Dominik Scharnagl

3.2 Emulation des Game Boy Advance

Autoren: Dominik Scharnagl, Florian Boemmel

Die Anwendung „mGBA“ wurde von den Entwicklern mit dem GUI-Toolkit Qt realisiert. Qt ermöglicht die plattformunabhängige Entwicklung von Anwendungen mit grafischer Benutzeroberfläche und basiert auf der Sprache C++. Damit ist es Entwicklern auch möglich, bereits realisierte Basis-Software problemlos zu integrieren.

3.2.1 Abgrenzung der Untersuchung

Autor: Florian Boemmel

Für die Untersuchung, wie der Emulator mit dem Betriebssystem interagiert, wird im Folgenden nur auf die dafür benötigten Klassen, Methoden und Konzepte eingegangen. Dabei liegt der Fokus ausschließlich auf Abläufe die zur Emulation des Soundsystems notwendig sind.

3.2.2 Start des Emulators

Autor: Dominik Scharnagl

3.2.3 Initialisierung des „mCore“

Autor: Dominik Scharnagl

3.2.4 Laden des ROM

Autor: Dominik Scharnagl

3.2.5 Starten des ROM

Autor: Dominik Scharnagl

3.2.6 Ausführung des ROM

Autor: Dominik Scharnagl

Wie üblich beginnt auch beim mGBA die Anwendung in der globalen `main`-Methode (*$/src/platform/qt/main.cpp*). Diese initialisiert den **ConfigController** mittels `argc` und `argv`. Anschließend wird eine neue Instanz der Klasse **GBAApp** ebenfalls mit `argc` und `argv`, sowie dem vorinitialisierten `configController` initialisiert. Die weitere

Logik der `main`-Methode dient der Initialisierung und Lokalisierung einer **Window**-Instanz zur Anzeige der mGBA GUI. Die dabei erzeugte **Window**-Instanz wird währenddessen dazu aufgefordert die Einstellungen aus dem bereits initialisierten `configController` zu laden. Hierzu wird die Methode `loadConfig()` der **Window**-Klasse verwendet.

ConfigController (*$/src/platform/qt/ConfigController.h & .cpp*)

Im Konstruktor der **ConfigController**-Klasse werden eventuell vorhandene Einstellungen aus einer „qt.ini“ oder „config.ini“ geladen und Standard-Werte der Membervariable `m_opts` vom Typen der **mCoreOptions**-Struktur (*$/include/mgba/core/config.h*) festgelegt, siehe Snippet 1.

```
1    ...
2    m_opts.audioSync = GameController::AUDIO_SYNC;
3    m_opts.audioBuffers = 1536;
4    m_opts.sampleRate = 44100;
5    m_opts.volume = 0x100;
6    ...
```

Snippet 1: Ausschnitt aus dem Konstruktor der ConfigController-Klasse

Alle im **ConfigController** enthaltenen Einstellungen werden im Laufe der Anwendung je nach Bedarf entweder über die `options()`-Methode oder über die `config()`-Methode abgerufen. Dabei wird bei der ersten Methode eine **mCoreOptions**-Struktur (*$/include/mgba/core/config.h*) und bei der zweiten Methode eine **mCoreConfig**-Struktur (*$/include/mgba/core/config.h*) bereitgestellt. Während die **mCoreConfig**-Struktur ausschließlich eine Abstraktion der konfigurierten Werte, der Standardwerte und der überschriebenen Werte bietet, stellt die **mCoreOptions**-Struktur alle verfügbaren Einstellungen direkt als typisierte Felder bereit.

GBAApp (*$/src/platform/qt/GBAApp.h & .cpp*)

Im Konstruktor der **GBAApp**-Klasse wird der lokale `m_configController` mit dem übergebenen initialisiert und der Treiber der **AudioProcessor**-Klasse mittels `AudioProcessor.setDriver(...)` festgelegt. Der **AudioProcessor.Driver** (eine Enumeration) legt dabei fest, ob entweder die **AudioProcessor**-Spezialisierung **AudioProcessorQt** oder **AudioProcessorSDL** mittels `AudioProcessor.create()`-Aufruf erstellt wird. Der zu verwendende **AudioProcessor.Driver** wird dabei durch den **ConfigController** über die Option „audioDriver“ bereitgestellt.

Window (*$/src/platform/qt/Window.h & .cpp*)

Im Konstruktor der **Window**-Klasse wird die lokale `m_config` mit dem übergebenen **ConfigController** (`config`-Parameter) und der lokale `m_inputController` initialisiert. Daraufhin wird eine neue Instanz der **GameController**-Klasse erzeugt, in der Membervariablen `m_controller` gespeichert und der `m_inputController` an die **GameController**-Instanz mittels `m_controller.setInputController(...)` übergeben. Weiter stellt der Konstruktor der **Window**-Klasse Verbindungen mittels Qt Signals & Slots zwischen den folgenden Methoden her:

- `Window.audioBufferSamplesChanged` → `m_controller::setAudioBufferSamples`
- `Window.sampleRateChanged` → `m_controller.setAudioSampleRate`

Als letzte Anweisung des Konstruktors wird die lokale `setupMenu()`-Methode der **Window**-Klasse aufgerufen. Neben diversen Menüeinträgen erzeugt diese Methode auch Menüpunkte zur Interaktion mit dem emulierten Soundsystem. Besonders interessant ist dabei auch der Menüpunkt „Record output...“, welcher mittels Qt Signals & Slots mit der Methode `openVideoWindow()` der **Window**-Klasse verbunden wird. Bei Ausführung der `openVideoWindow()`-Methode wird eine neue Instanz der **VideoView**-Klasse erzeugt (falls nicht bereits geschehen) und die folgenden Methoden mittels Qt Signals & Slots mit Methoden der **GameController**-Klasse verbunden. Zum Ende der Methode wird das *QWidget* **VideoView** noch zur Anzeige gebracht.

- `VideoView.recordingStarted` → `m_controller.setAVStream`
- `VideoView.recordingStopped` → `m_controller.clearAVStream`

Durch den Aufruf der `loadConfig()`-Methode wird wiederum die Methode `reloadConfig()` der **Window**-Klasse aufgerufen. Diese vermittelt unter anderen die aktuelle **mCoreConfig**-Struktur der `m_config` (vom Typen **ConfigController**) an den `m_controller` (vom Typen **GameController**) mittels `setConfig()`-Methode der **GameController**-Klasse.

VideoView (*$/src/platform/qt/VideoView.h & .cpp*)

Bei der Instanziierung der **VideoView**-Klasse verwendet der Konstruktor die globale Methode **FFmpegEncoderInit** ($/src/feature/ffmpeg/ffmpeg-encoder.c) zur Initialisierung der Membervariablen `m_encoder`. Die für die Audio-/Videoausgabe verwendete Struktur vom Typen **FFmpegEncoder** ($/src/feature/ffmpeg/ffmpeg-encoder.c) wird beim Aufruf der Instanzmethode `startRecording()` der **VideoView**-Klasse mttels globaler **FFmpegEncoderOpen** Methode so final konfiguriert, dass der Encoder die bei der Emulation anfallenden Audio-/Videodaten aufzeichnet. Zum Abschluss der **startRecording()**-Methode wird das Qt Signal `recordingStarted` mit dem Feld `d` vom Typen der Struktur **mAVStream** der `m_encoder` Membervariablen als Parameter gesendet. Dieses Signal endet schließlich in einen Aufruf der `setAVStream`-Methode der **GameController**-Instanz `m_controller` der **Window**-Klasse.

GameController (*$/src/platform/qt/GameController.h & .cpp*)

Im Konstruktor der **GameController**-Klasse wird die lokale `m_audioProcessor` Membervariable mit dem Ergebnis des `AudioProcessor.create()`-Aufrufs initialisiert. Daraufhin erfolgt das Setup der Membervariable `m_threadContext` vom Typen der **mCoreThread**-Struktur. Hierbei wird unter anderen das `startCallback`, `cleanCallback` und das `userData` Feld der Kontextvariablen entsprechend belegt. Abschließend werden die folgenden Methoden mittels Qt Signals & Slots miteinander verbunden:

- `GameController.gamePaused` → `m_audioProcessor.pause`
- `GameController.gameStarted` → `m_audioProcessor.setInput`

Wählt der mGBA-Anwender im Menü den Punkt „Load ROM...“, wird hierfür die Methode `selectROM()` der **Window**-Klasse ausgeführt. Nach erfolgter Auswahl einer entsprechend unterstützten Datei, wird die Methode `loadGame(path)` der lokalen **GameController**-Instanz (`m_controller`) mit dem Pfad zur ausgewählten ROM-Datei aufgerufen. Diese führt nach einigen Vorabaktionen die Methode `openGame()` der **GameController**-Instanz aus. Mittels globaler **mCoreFind**-Methode (*$/src/core/core.c*) wird der vom Format der ROM-Datei abhängige „Core“ ermittelt und erstellt. Handelt es sich bei der ROM-Datei um ein Game Boy Advance (kurz „GBA“) Speicherabbild, wird die globale **GBACoreCreate**-Methode (*$/src/gba/core.c*) dazu verwendet den Speicher für die Struktur **GBACore** (*$/src/gba/core.c*) zu allokieren. Das dabei implizit allokierte **mCore**-Feld `d` wird daraufhin mit diversen Funktionszeigern zu globalen Methoden mit dem Prefix **_GBA** beziehungsweise **_GBACore** initialisiert. Das auf diese Weise konfigurierte `d`-Feld wird dann von der globalen **GBACoreCreate**-Methode zurückgeliefert und im Feld `mCoreThread.core` der lokalen Membervariable `m_threadContext` der **GameController**-Instanz gespeichert.

_GBACoreInit (*$/src/gba/core.c*)

Der erste der zuvor festgelegten Funktionszeiger der daraufhin verwendet wird ist der der Funktion auf die im Feld `init` verwiesen wird. Nach Durchlaufen der globalen **GBACoreCreate**-Methode ist das die globale Methode **_GBACoreInit**. Die globale Methode initialisiert die Felder `cpu` und `board` des **mCore**. Hierzu wird für das Feld `cpu` die Struktur **ARMCore** (*$/include/mgba/internal/arm/arm.h*) und für das Feld `board` die Struktur **GBA** (*$/include/mgba/internal/gba/gba.h*) verwendet. Nach der Initialisierung einzelner weiterer Felder wird dann die globale Methode **GBACreate** (*$/src/gba/gba.c*) mit den Verweis auf die zuvor initialisierte `board`-Variable vom Typen der **GBA**-Struktur aufgerufen. Diese legt unter anderen als Wert für das `init`-Feld des `d`-Feldes vom Typen

der **mCPUComponent**-Struktur der `board`-Variablen die globale Methode **GBAInit** (*$/src/gba/gba.c*) fest. ~~Im weiteren Verlauf der _**GBACoreInit**-Methode wird schließlich noch die globale Methode **ARMInit** aufgerufen und ihr dabei die zuvor initialisierte cpu-Variable vom Typen der **ARMCore**-Struktur übergeben.~~ Anschließend wird in Folge der Aufrufe der globalen Methoden **ARMSetComponents** (*$/src/arm/arm.c*) und **ARMInit** (*$/src/arm/arm.c*) die zuvor auf dem `init`-Feld des `d`-Feldes der `board`-Variablen die globale Methode **GBAInit** aufgerufen.

GBAInit (*$/src/gba/gba.c*)

In dieser Low-Level Init-Routine werden alle virtuellen Hardwarekomponenten des **mCore** initialisiert und mit weiteren globalen Methoden verlinkt. Dazu gehöhrt unter anderen das Setup des Interrupt-Handlers, welcher über das Feld `irqh` des cpu-Feldes der **GBA**-Instanz an die globale Methode **GBAInterruptHandlerInit** übergeben wird. Nach der Initialisierung des Interrupt-Handlers folgt die Initialisierung des Speichers des **GBA** mittels globaler **GBAMemoryInit**-Methode. Darauf folgt das Setup der „Audio“-Peripherie des **GBA** mit Hilfe der globalen Methode **GBAAudioInit**.

GBAInterruptHandlerInit (*$/src/gba/gba.c*)

Die einzige Aufgabe dieser Methode ist es die **ARMInterruptHandler**-Struktur (*$/include/mgba/internal/arm/arm.h*) des **GBA** zu initialisieren. Hierzu legt die Methode entsprechende Funktionszeiger für die einzelnen Service-Routinen der Interrupt-Handler-Struktur fest.

```
1     irqh->reset = GBAReset;
2     irqh->processEvents = GBAProcessEvents;
3     irqh->swi16 = GBASwi16;
4     irqh->swi32 = GBASwi32;
5     ...
```

Snippet 2: Ausschnitt aus der **GBAInterruptHandlerInit**-Methode

GBAMemoryInit (*$/src/gba/memory.c*)

Neben den diversen Initialisierungsoperationen und Aufrufen weiterer Subroutinen zur Initialisierung des `memory`-Feldes der „CPU“ über das cpu-Feld des **GBA** legt auch diese Methode entsprechende Funktionszeiger für die einzelnen Speicherzugriffe auf der **ARMMemory**-Struktur (*$/include/mgba/internal/arm/arm.h*) fest. Die im folgenden Snippet gezeigten Zeilen sind für die Untersuchtung der Emulation des Soundsystems relevant.

```
1     ...
2     cpu->memory.load32 = GBALoad32;
3     cpu->memory.load16 = GBALoad16;
4     cpu->memory.load8 = GBALoad8;
5     cpu->memory.loadMultiple = GBALoadMultiple;
6     cpu->memory.store32 = GBAStore32;
7     cpu->memory.store16 = GBAStore16;
8     cpu->memory.store8 = GBAStore8;
9     cpu->memory.storeMultiple = GBAStoreMultiple;
10    cpu->memory.stall = GBAMemoryStall;
11    ...
```

Snippet 3: Ausschnitt aus der **GBAMemoryInit**-Methode

GBAAudioInit (*$/src/gba/audio.c*)

Die globale **GBAAudioInit**-Methode ist für den vollen Setup der **GBAAudio**-Struktur (*$/include/mgba/inter-*

nal/gba/audio.h) der **GBA**-Instanz verantwortlich. Neben diversen Audio-Parametern werden auch benötigte **mTimingEvent**-Strukturen initialisiert. Diese Event-Strukturen dienen dem Scheduler später bei der quasi-parallelen Verarbeitung der Audiodaten. Die dafür eigens definierten Events werden mit entsprechenden Callback-Routinen verlinkt, welche die verzögerte / parallele Verarbeitung der Audiodaten durchführen. Zusammen mit der ebenfalls globalen Methode **GBAudioInit** werden während der Ausführung der Methode die folgenden Events konfiguriert.

Event	**Priorität**	**Kanal**	**Callback**
GB(A) Audio Sample	$0x18$		`_sample`
GB Audio Frame Sequencer	$0x10$		`_updateFrame`
GB Audio Channel 1	$0x11 \rightarrow 0x18$	1	`_updateChannel1`
GB Audio Channel 2	$0x12$	2	`_updateChannel2`
GB Audio Channel 3	$0x13$	3	`_updateChannel3`
GB Audio Channel 3 Memory	$0x14$	3	`_fadeChannel3`
GB Audio Channel 4	$0x15$	4	`_updateChannel4`

Tabelle 18: Übersicht der Events der Soundkanäle des Game Boy Advance

_GBACoreSetAudioBufferSize (*$/src/gba/core.c*)

Anschließend wird mit Hilfe der globalen Methode **mCoreLoadForeignConfig** (*$/src/core/core.c*) die Konfiguration der **ConfigController**-Instanz, die durch die **Window**-Klasse an den **GameController** übertragen wurde, auf den **mCore** des `core`-Feldes der Membervariablen `m_threadContext` angewendet. Hierbei wird unter anderen die Funktion auf die im Feld `setAudioBufferSize` verwiesen wird aufgerufen. Nach Durchlaufen der globalen **GBACoreCreate**-Methode ist das die globale Methode **_GBACoreSetAudioBufferSize**. Sie leitet den Aufruf direkt weiter an die globale Methode **GBAAudioResizeBuffer** unter Verwendung des `audio`-Feldes der **GBAAudio**-Struktur des `board`-Felds der **mCore**-Struktur.

_GBACoreLoadConfig (*$/src/gba/core.c*)

Nachdem die Funktion auf die im Feld `setAudioBufferSize` verwiesen wird aufgerufen wurde, wird von der globalen Methode **mCoreLoadForeignConfig** die allgemeine Funktion auf die im Feld `loadConfig` verwiesen wird aufgerufen. Nach Durchlaufen der globalen **GBACoreCreate**-Methode ist das die globale Methode **_GBACoreLoadConfig**. Sie übernimmt im Wesentlichen die Konfiguration für das Mastervolume des `audio`-Feldes der **GBAAudio**-Struktur des `board`-Felds der **mCore**-Struktur.

_GBACoreLoadROM (*$/src/gba/core.c*)

Auf die vorangegangene Konfiguration des **mCore** wird schließlich der ROM in den „Core“ geladen. Hierzu verwendet die **GameController**-Instanz die Funktion auf die im Feld `loadROM` verwiesen wird. Nach Durchlaufen der globalen **GBACoreCreate**-Methode ist das die globale Methode **_GBACoreLoadROM**. Sie dient dem finalen Setup der virtuellen Hardwarekonfiguration des **mCore** sowie der Initialisierung des virtuellen Prozessspeichers im `memory`-Feld des `board`-Feldes der **mCore**-Instanz.

_GBACoreSetAVStream (*$/src/gba/core.c*)

Bevor mit der eigentlichen Emulation begonnen wird, wird nun noch der Audio-/Videostream in Form der **mAVStream**-Struktur als `m_stream`-Membervariable der **GameController**-Instanz an den **mCore** übergeben. Diese geschieht durch Aufruf der Funktion auf die im Feld `setAVStream` verwiesen wird. Nach Durchlaufen der

globalen **GBACoreCreate**-Methode ist das die globale Methode _**GBACoreSetAVStream**. Diese Methode geht hierbei lediglich dazu über den **mAVStream**-Verweis im `stream`-Feld des `board`-Feldes der **mCore**-Instanz zu speichern.

_GBACoreEnableAudioChannel (*$/src/gba/core.c*)

Aufgabe dieser globalen Methode ist es dem `board`-Feld des **mCore** mittels gegebener Parameter zu konfigurieren. Das hierbei vorgenommene Setup bezieht sich auschließlich auf das `audio`-Feld des `board`-Felds vom Typen der **GBA**-Struktur. Die dabei vorgenommenen Änderungen beziehen sich somit nur auf Felder der **GBAAudio**-Struktur.

mCoreThreadStart (*$/src/core/thread.c*)

Nach Abschluss des vollständigen Setups des **mCore** wird die im `m_threadContext.core` gespeicherte Instanz samt `m_threadContext` an die globale Methode **mCoreThreadStart** übergeben. Bevor aber die Methode den eigentlichen Thread erzeugt initialisiert sie diverse Mutex- sowie Condition-Instanzen zur Synchronisation der Thread-übergreifenden Operationen. Von besonderer Bedeutung sind hierbei der Mutex `audioBufferMutex` und die Condition `audioRequiredCond`. Beide Felder sind Teil der **mCoreSync**-Struktur des `theadContext`-Parameters vom Typen **mCoreThread**.

Sind alle Bedingungen für das Multithreading erfüllt, legt die Methode mittels globaler **ThreadCreate**-Methode (*$/include/mgba-util/platform/{os}/threading.h*) den Emulations-Thread an. Als **ThreadEntry** wird dabei die globale Methode _**mCoreThreadRun** und als `context`-Parameter ein Verweis auf den **mCoreThread** alias `threadContext` verwendet. Der Verweis auf den so erzeugten Thread wird schließlich noch im `thread`-Feld des `threadContext`-Parameters gespeichert.

_mCoreThreadRun (*$/src/core/thread.c*)

Bevor nun mit der eigentlichen Ausführung des Prozesses begonnen wird, nimmt die globale **_mCoreThreadRun** noch ein paar Vorkehrungen für die Threadinteraktion mittels Callback-Routinen vor. Darauf folgt der Aufruf der im Feld `startCallback` des `threadContext`-Parameters hinterlegten Methode. Dabei wird die durch die **GameController**-Klasse definierte anonyme Methode mit `threadContext`-Parameter aufgerufen. Während der Ausführung des Start-Callbacks stellt der **GameController** sicher, dass im **mCore** (im `core`-Feld des `threadContext`-Parameters) die korrekten Audio-Kanäle ein- beziehungsweise ausgeschaltet sind. Hierzu verwendet der **GameController** die Funktion auf die im Feld `enableAudioChannel` verwiesen wird. Nach Durchlaufen der globalen **GBACoreCreate**-Methode ist das die globale Methode **_GBACoreEnableAudioChannel**.

Abgeschlossen wird der Code der Callback-Routine mit dem dynamischen Auslösen der Signale **gameStarted** und **startAudio** der **GameController**-Instanz die für den übergebenen `threadContext` zuständig ist. Während **gameStarted** auf die `setInput()`-Methode der `m_audioProcessor`-Instanz im **GameController** weiterleitet, um den aktuellen **mCoreThread** der **AudioProcessor**-Instanz mitzuteilen, führt der Aufruf des **startAudio**-Signals zum Aufruf der `start()`-Methode der `m_audioProcessor`-Instanz im **GameController**.

Wurde auch alle weiteren Callback-Routinen durchlaufen, beginnt die Ausführung des Prozesses durch stetigen Aufruf der Funktion auf die im Feld `runLoop` verwiesen wird. Nach Durchlaufen der globalen **GBACoreCreate**-Methode ist das die globale Methode **_GBACoreRunLoop**. Dies geschieht solange, wie sich der Thread im Zustand kleiner/gleich `THREAD_MAX_RUNNING` befindet.

Nach Durchlaufen der Setup-Phase bestehend aus dem Einrichten der notwendigen Strukturen und dem Laden des Prozessspeichers, kann der Inhalt des ROMs gemäß dem bekannten Instruction-Set eines ARM-Prozessors abgerarbeitet werden. Hierbei wird jede Anweisung im ROM sequentiell eine nach der anderen ausgewertet und ausgeführt. Die dabei im ROM beschriebenen Assembler Befehle für die ARM-Architektur, werden durch entsprechende Methoden abgearbeitet, welche das Verhalten der Plattform so emulieren, als ob der Prozess auf einem physikalichen ARM ausgeführt werden würde.

_GBACoreRunLoop (*$/src/gba/core.c*)

Die bereits im vorangegangenen Abschnitt erwähnte globale Methode ist für die Ausführung der einzelnen Assembler Anweisungen im geladenen ROM zuständig. Hierzu bedient sie sich der ebenfalls globalen Methode **ARMRunLoop** und übergibt dieser dabei die Kontrolle über die „CPU“.

ARMRunLoop (*$/src/arm/arm.c*)

Mit Hilfe der übergebenen „CPU“ in Form der **ARMCore**-Struktur führt die Methode die Assembler-Anweisungen Schritt für Schritt aus. Dabei berücksichtigt sie die Anzahl der auszuführenden Anweisungen in Abhängigkeit zur Ausführung des nächsten Events. Bis es zur einer Abarbeitung von Events kommt wird je Zyklus die globale Methode **ARMStep** ausgeführt. Entspricht die Anzahl der vollzogenen Zyklen dem Zyklus eines anstehenden Events, wird die weitere Verarbeitung unterbrochen und dem Interrupt-Service-Routinen-Handler Zeit gegeben die anstehenden Events abzuarbeiten.

ARMStep (*$/src/arm/arm.c*)

Entsprechend der Natur von Software welche auf Hardware-Level in Form von Assembler-Befehlen ausgeführt wird, holt auch diese Methode stets den **OpCode** des als nächstes auszuführenden Befehls aus dem Prefech-Speicher der **MMU**. Basierend auf den Wert des **OpCodes** wird aus einem global definierten und mittels Makros gefülltem Array der Zeiger zur Funktion ermittelt, welche für die Emulation des Assembler-Befehls verantwortlich ist.

Die zur Ausführung per Makro definierten Routinen vollziehen dabei nicht ausschließlich einfache Delegationsarbeit zu global definierten Methoden deren Funktionszeiger in diversen Feldern des **mCore** gespeichert sind. Sie führen zusätzliche Prüfungen, Vorabbedingungen und Nachbedingungen sowie weitere Operationen aus die für die korrekte Interaktion mit dem Prozess und dem emulierten Speicher notwendig sind. Die Operationen stellen dabei ein

Mindestmaß an Korrektheit der ausgeführten Assembler-Befehle vor und nach Ausführung der Callback-Routinen sicher - falls für den Befehl eine solche vorliegt.

```
1     uint32_t opcode = cpu->prefetch[0];
2     cpu->prefetch[0] = cpu->prefetch[1];
3
4     cpu->gprs[ARM_PC] += WORD_SIZE_ARM;
5
6     LOAD_32(
7         cpu->prefetch[1],
8         cpu->gprs[ARM_PC] & cpu->memory.activeMask,
9         cpu->memory.activeRegion);
10
11    ...
12
13    uint32_t instructionIndex = ((opcode >> 16) & 0xFF0) | ((opcode >> 4) & 0x00F);
14
15    ARMInstruction instruction = _armTable[instructionIndex];
16    instruction(cpu, opcode);
```

Snippet 4: Ausschnitt aus der **ARMStep**-Methode

Die Signatur einer **ARMInstruction** ist dabei so einfach wie möglich gehalten. So erwartet jede Funktion des Instruction-Sets einen Verweis auf die „CPU“, auf der die Anweisung ausgeführt werden soll, sowie den zur **ARMInstruction** geführten **OpCode**.

Ein Beispiel für so eine Makrodefinition kann im folgenden betrachtet werden. Die eigentliche Verarbeitung mittels globaler Callback-Routine findet in Zeile 5 des Snippets 5 statt.

```
1     DEFINE_LOAD_STORE_T_INSTRUCTION_ARM(STRT,
2         enum PrivilegeMode priv = cpu->privilegeMode;
3         int32_t r = cpu->gprs[rd];
4         ARMSetPrivilegeMode(cpu, MODE_USER);
5         cpu->memory.store32(cpu, address, r, &currentCycles);
6         ARMSetPrivilegeMode(cpu, priv);
7         ARM_STORE_POST_BODY;)
```

Snippet 5: ARM Instruction Makro für **STRT**

Gemäß vorangegangenem Snippet 3 sah man in Zeile 6 der Methode **GBAMemoryInit**, dass das Feld `store32` mit der globalen Methode **GBAStore32** belegt wurde, welche an dieser Stelle bei der Ausführung des Assembler-Befehls **STRT** (unter anderen) ausgeführt wird.

Der Aufruf der gloablen **GBAStore32** (*$/src/gba/memory.c*) Methode führt dann zum Beispiel zum Aufruf der ebenfalls globalen Methode **GBAIOWrite32** (*$/src/gba/memory.c*) welche wiederum zum Beispiel in eine der für das Soundsystem folgenden relevanten Methoden münden kann:

- **GBAAudioWriteWaveRAM** (*$/src/gba/audio.c*)
- **GBAAudioWriteFIFO** (*$/src/gba/audio.c*)

Damit die vom ROM beziehungsweise vom Prozess generierten Audiodaten auch mittels **mAVStream** in der **VideoView** sowie durch das **AudioDevice** verarbeitet werden bedient sich mGBA verschiedener Methoden.

Zur Ausgabe über den **mAVStream** greift der Callback des „GB(A) Audio Sample"-Events (die globale **_sample** Methode) direkt auf das `stream`-Feld über den **GBA**-Verweis des Feldes `p` in der **GBAAudio**-Struktur zu. Hierbei bedient sich die **_sample** Methode des dort eingetragenen Callbacks im Feld `postAudioBuffer` und ruft somit eine Methode der vom mGBA verwendeten **FFmpeg**-Library auf um die Audiodaten im Stream abzulegen.

Betrachtet man den von der **VideoView** unabhängigen Ablauf der Audiodatenverarbeitung stellt man fest, dass die Verarbeitung direkt über den Speicher des **mCore** stattfindet. Da aber der Speicher vom Emulations-Thread verwendet wird, kann der Main-Thread nicht ohne weiteres auf diesen zugreifen. An dieser Stelle kommen die in der **mCoreThreadStart** initialisierte Condition `audioRequiredCond` sowie der Mutex `audioBufferMutex` ins Spiel. Während der Callback des „GB(A) Audio Sample"-Events (die globale **_sample** Methode) die globale Methode **mCoreSyncProduceAudio** verwendet, nutzt im Main-Thread die **AudioDevice**-Instanz des verwendeten **AudioProcessor**'s die globale Methode **mCoreSyncConsumeAudio**.

Letztere verwendet die **mCoreSyncConsumeAudio** Methode nach dem Zugriff auf die Audiodaten im Speicher, während vor dem Zugriff weitere Zugriffe durch den Prozess mittels Aufruf der globalen **mCoreSyncLockAudio** Methode blockiert werden. Erst der Aufruf der **mCoreSyncConsumeAudio** Methode gibt den Zugriff auf den Audiodaten-Speicher wieder frei.

Ebenso wie das **AudioDevice** den Zugriff auf die Audiodaten blockiert, während diese gelesen werden, so blockiert auch die **_sample**-Methode den Zugriff auf diese mit einem ebenfalls vorgeschalteten Aufruf der **mCoreSyncLockAudio** Methode. Nach der Bearbeitung der Audiodaten werden diese schließlich mit Aufruf der globalen **mCoreSyncConsumeAudio** Methode für den Zugriff wieder freigegeben.

3.3 Emulation des Soundsystems

Autor: Dominik Scharnagl

3.3.1 Audioverarbeitung im Assembler

Autor: Dominik Scharnagl

3.3.2 Weiterverarbeitung im Emulator

Autor: Dominik Scharnagl

3.4 Interaktion mit dem Betriebssystem

3.4.1 Start des Emulators

Autor: Florian Boemmel

In Kapitel TODO: REF wurde bereits der Ablauf bis zur Erstellung der AudioProcessor-Klasse, sowie das Setzten des Treibers beschrieben. Auf diesen Ablauf wird im Folgenden nun aufgebaut. In dieser Arbeit wird der „Qt Multimedia“ Treiber untersucht und schließt somit die Betrachtung des SDL-Treibers aus.

Startet der Benutzer den mGBA, wird unter Verwendung des zuvor gesetzten Treibers, eine neue Instanz von AudioProcessor erstellt. In diesem Fall wird die AudioProcessor::create() Methode aufgerufen und wählt über ein Switch Statement den Konstruktor von AudioProcessorQt aus und liefert eine neue Instanz zurück. Dieser verfügt über keine Logik und ist demnach leer. Für die weitere Untersuchung folgt ein Klassendiagramm der wesentlichen Klassen auf Seitens der Qt-Anwendung.

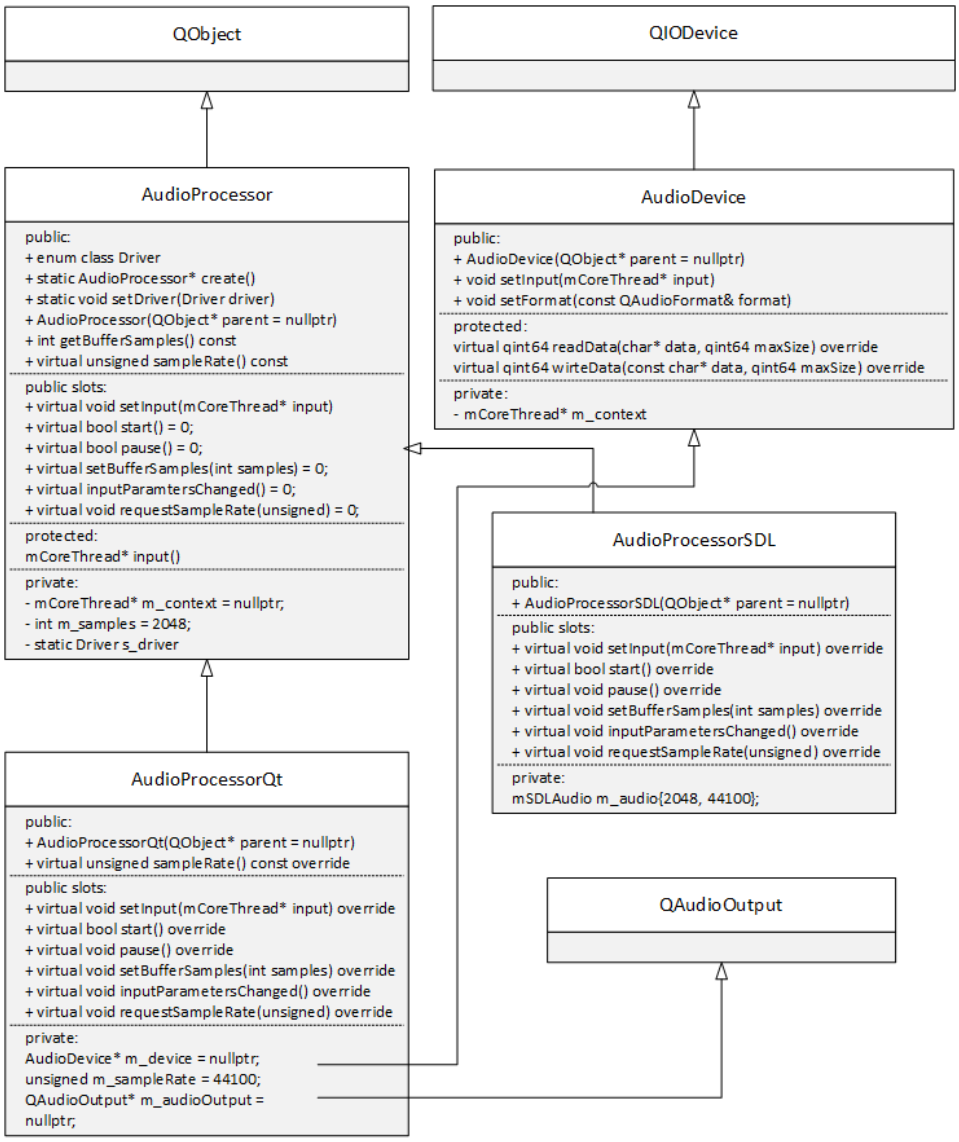

Abbildung 2: Audioklassen in der QT-Anwendung

AudioProcessor-Klasse (*$/src/platform/qt/AudioProcessor.cpp*)

Die Klasse AudioProcessor erbt von QObject, ist eine abstrakte Klasse und definiert ein Interface für die Klassen AudioProcessorQt und AudioProcessorSDL. QObject ist die Basisklasse aller Qt Objekte und im Bezug auf die Objektmodellierung somit das Herzstück von Qt.

AudiProcessor::getBufferSamples liefert die Anzahl der, intern in der Variablen m_samples gespeicherten, Samples zurück. Diese wird initial auf den Wert 2048 gesetzt. m_samples kann mit dem Aufruf von **AudiProcessor::setBufferSamples** geändert werden. Es folgen abstrakte Methoden und Slots. Diese werden in der Klasse **AudioProcessorQt** überschrieben.

Weiterhin beinhaltet die Klasse eine Variable m_context. Diese stellt einen Zeiger auf den mCoreThread dar und kann mit der Methode **AudioProcessor::setInput(mCoreThread* input)** gesetzt werden.

Unter Verwendung einer selbst kompilierten Version von mGBA, mit Log-Ausgaben, konnte nach dem Erstellen der AudioProcessorQt-Klasse festgestellt werden, dass die **AudioProcessorQt::inputParameterChanged** Methode fünfmal aufgerufen wird. Gefolgt von einem Aufruf der **AudioProcessorQt::requestSampleRate** und dreimal der **AudioProcessorQt::inputParameterChanged** Methode. Ein weiterer Aufruf der **AudioProcessorQt::requestSampleRate** Methode folgt. Interessant an dieser Stelle ist, dass jeder Aufruf der zuvor aufgeführten Methoden bereits bei der Überprüfung ob ein **AudioDevice** vorhanden ist, scheitert. Daraus folgt, dass an dieser Stelle keine Änderungen vorgenommen werden.

3.4.2 Einstellungen über die mGBA GUI

Autor: Florian Boemmel

Der mGBA ist gestartet und die**AudioProcessorQt** wurde erstellt. Der Benutzer kann nun über die Menüleiste Werkzeuge→Einstellungen→Audio/Video auf die Audio und Video Einstellungen zugreifen.

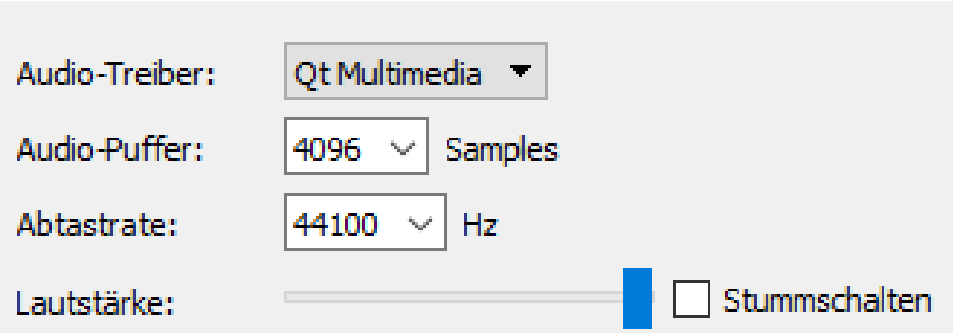

Abbildung 3: Audioeinstellungen im mGBA

In Abbildung 6 sind die für dem Benutzer möglichen Einstellungen im Bezug auf das Audiosystem dargestellt. Der Audio-Treiber kann entweder auf Qt Multimedia oder SDL eingestellt werden. Der Audio-Puffer bietet die gängisten Anzahl an Samples von 512 bis 4096. Kann aber auch benutzerspezifische eingestellt werden. Weiterhin kann die Abtastrate gesetzt werden. Auch hier sind die gängigsten Raten vorgeschlagen, können aber auch frei gesetzt werden. Schliesslich folgt die Lautstärkeneinstellung.

Wie schon zu erahnen ist, wird nach einer Veränderung des Audio-Treibers der Destruktor der zuvor gewählten Klasse aufgerufen und ein erneuter Aufruf der **AudioProcessor::create** Methode erzeugt eine neue AudioProcessor Instanz. Anließend wird die Methode **AudioProcessor::setBufferSamples** aufgerufen. Als Übergabeparameter erhählt diese den Wert aus dem Feld Audio-Puffer und setzt diesen in die interne Variable m_samples.

3.4.3 Starten des ROM

Autor: Florian Boemmel

Wählt der Benutzer eine neue ROM aus und startet diese, wird zunächst die **AudioProcessor::setInput** Methode aufgerufen. Diese setzt den Zeiger auf den m_coreThread zur internen Variablen m_context. Es folgt der Aufruf der **AudioProcessorQt::start** Methode.

AudioProcessorQt::start() (*$/src/platform/qt/AudioProcessorQt.cpp*)

Im ersten Schritt wird überprüft, ob der m_coreThread bereits gesetzt wurde. Im nächsten Schritt wird der internen Variablen **m_device** ein neue Instanz von **AudioDevice** zugewiesen.

Die Klasse **AudioDevice** stellt das Bindeglied zwischen AudioProcessorQt und dem mCoreThread dar. **AudioDevice** erbt von QIODevice. QIODevice ist ein Interface für alle I/O Geräte und kann deshalb nicht direkt instanziiert werden. Wird von dieser Basisklasse abgeleitet, müssen die Methoden **readData** und **writeData** überschrieben werden. Zusätzlich muss die Methode **setOpenMode** mit dem gewünschten Modus im Konstruktor aufgerufen werden. In diesem Fall wird der Parameter „ReadOnly“ übergeben. Daraus resultiert ein nur lesbares QIODevice. Weiterhin wird auch hier der mCoreThread übergeben und gesetzt. Die bereits erwähnte Methode **writeData** spielt hier keine Rolle, da nicht auf das Gerät geschrieben werden darf. Sie muss jedoch überschrieben werden aber beinhaltet nur eine Warnung. Eine genauere Betrachtung der Funktionsweise der Klasse folgt.

Es folgt das Erzeugen einer neuen Instanz von QAudioFormat. Die Klasse **QAudioFormat** speichert Informationen über Audio-Stream Parameter.

```
1     QAudioFormat format;
2         format.setSampleRate(m_sampleRate); // m_sampleRate = 44100
3         format.setChannelCount(2);          // 2 bedeutet Stereo 1 Mono
4         format.setSampleSize(16);           // Typischerweise 8 oder 16
5         format.setCodec("audio/pcm");       // Linear PCM
6         format.setByteOrder(QAudioFormat::Endian(QSysInfo::ByteOrder));
7         // Little- oder Big-Endian
8         format.setSampleType(QAudioFormat::SignedInt);  // Sample Typ
9         m_audioOutput = new QAudioOutput(format, this);
10        m_audioOutput->setCategory("game");
```

Snippet 6: Ausschnitt aus AudioProcessorQt::start()

Wie bereits am Ende von Snippet 6 zu sehen, wird eine neue Instanz der Klasse **QAudioOutput** erzeugt und in die interne Variable **m_audioOutput** geschrieben. Dem Konstruktor der Klasse **QAudioOutput** wird das zuvor erstellte Format übergeben.

QAudioOutput stellt ein Interface zur Verfügung mit dem Audiodaten zu einem Audio-Gerät gesendet werden können(z.B. Lautsprecher, Kopfhörher usw.) Die **setCatecory** Methode setzt den Modus auf „game“. Einige Plattfomren können Audio-Streams in Kategorien gruppieren. Nützlich ist dieser Methodenaufruf, da unter Windows der mGBA nun im Lautstärken Mixer angezeigt wird.

Durch den Methodenaufruf **m_device→setInput** wird der Klasse **AudioDevice** der **mCoreThread** übergeben. Daraufhin wird die **m_device→setFormat** Methode, mit einer Referenz auf das Format von **m_audioOutput** als Übergabeparameter, aufgerufen.

AudioDevice::setFormat(const QAudioFormat& format) (*$/src/platform/qt/AudioDevice.cpp*)

Wie gewohnt wird zunächst überprüft, ob der **mCoreThread** vorhanden ist. Im nächsten Schritt wird ein Multiplikator errechnet. Dieser wird abhängig von der eingestellten FPS berechnet. Bei 60 FPS beträgt dieser 0,995458. Anschließend werden die im **mCoreThread** befindlichen Audio-Speicherbereiche gesperrt. Nun werden die Frequenzen der beiden Audiokanäle im **mCoreThread** mit den Frequenzen, multipliziert mit dem zuvor errechneten Multiplikator, des **m_audioOutput** synchronisiert. Mit dem entsperren der Speicherbereiche endet die Methode.

Zurück in der **AudioProcessorQt::start** Methode wird mit dem Aufruf von **m_audioOutput→start(m_device)** die Audioausgabe gestartet. Schließlich wird der Status des **m_audioOutput** auf aktive gesetzt. Konnte der Status erfolgreich auf aktiv gesetzt werden, liefert **AudioProcessorQt::start** „true“ zurück.

3.4.4 Transferirrung der Audiodaten

Autor: Florian Boemmel

Wie bereits im vorangegangenen Kapitel erwähnt, wird der **m_audioOutput→start** Methode das **AudioDevice** übergeben. Dies bewirkt, dass die Klasse **QAudioOutput** jetzt von der Klasse **AudioDevice** Daten für die Audioausgabe liest und diese an die Systemausgabe weiterleitet. Einmal gestartet, läuft die Audioausgabe kontinuierlich weiter. Nur ein Aufruf der Methode **AudioProcessorQt::pause** stoppt die Ausgabe.

Die Klasse **QAudioOutput** ruft nun selbstständig in gewissen Zeitabständen die **AudioDevice::readData** Methode auf und übergibt einen Zeiger in dem die Daten zur Ausgabe eingelesen werden sollen und dessen Größe.

AudioDevice::readData(char* data, qint64 maxSize) (*$/src/platform/qt/AudioDevice.cpp*)

Wie bereits erwähnt, muss diese Methode überschrieben werden. Zu Beginn wird überprüft ob die maximal zulässige Größe des übergebenen Puffers nicht überschritten ist und der **m_coreThread** vorhanden ist. Nun wird, wie auch in der **AudioDevice::setFormat** Methode, der Speicherbereich im **m_coreThread** in dem die Audiodaten gespeichert sind gesperrt. Mit dem Methodenaufruf **blip_samples_avail** wird die Anzahl der verfügbahren Samples im **m_coreThread** abgefragt. Überschreitet die Anzahl der Samples die Größe des Puffers, geteilt durch die Strukturgröße **GBAStereoSample**, wird die Anzahl der verfügbahren Samples auf die Puffergröße, geteilt durch die Strukturgröße **GBAStereoSample**, reduziert.

Jetzt können die Samples aus dem **m_coreThread** gelesen werden. Dies geschieht durch den Methodenaufruf von **blip_read_samples** für den linken und den rechten Kanal. Dazu wird der übergebene Puffer auf die Struktur **GBAStereoSample** gecastet und übergeben. Ein Entsperren des zuvor gesperrten Speicherbereichs und das Zurückgeben der Anzahl gelesener Daten beendet die Methode **AudioDevice::readData**. Die im Puffer befindlichen Daten werden nun in der Klasse **QAudioOutput** verarbeitet und zum Systemausgang weitergeleitet.

4 Zusammenfassung

4.1 Inhalt des Dokumentes

Autor: Dominik Scharnagl

4.2 Fazit zur Studienarbeit

Dominik Scharnagl

Hello World by Dominik.

Florian Boemmel

Hello World by Florian.

Ngoc Luu Tran

Hello World by Ngoc.

Literatur

[1] Nintendo: *Game Boy Advance*
https://www.nintendo.de/Unternehmen/Unternehmensgeschichte/Game-Boy-Advance/Game-Boy-Advance-627139.html, Mai 2018

[2] Giga Ratgeber: *Was ist der Unterschied zwischen Simulation, Emulation & Virtualisierung?*
https://www.giga.de/extra/ratgeber/specials/was-ist-der-unterschied-zwischen-simulation-emulation-virtualisierung-computertechnik/, Mai 2018

[3] Nintendo: *Game Boy Advance*
http://de.nintendo.wikia.com/wiki/Game_Boy_Advance, Mai 2018

[4] Coranac: *18. Beep! GBA sound introduction*
https://www.coranac.com/tonc/text/sndsqr.htm#sec-intro, Mai 2018

[5] BELOGIC: *The Audio ADVANCE*
http://belogic.com/gba/, Juni 2018

Bilder

- Abbildung 1: *Game Boy Advance - Blue Edition*
https://d3nevzfk7ii3be.cloudfront.net/igi/L3WryntCMswfDks1.large, Mai 2018
- Abbildung 5: *Übersicht der Audioklassen in der Qt Anwendung*