

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

Autor	Matrikelnummer
Dominik Scharnagl	30 54 54 1
Florian Boemmel	00 00 00 0
Ngoc Luu Tran	30 32 96 3

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
2.2	Plattformen	8
2.2.1	DevkitPro	8
2.2.2	Belogic	8
2.2.3	„Sappy“	9
2.3	Softwareentwicklung	18
2.3.1	Erstellung der Beispielprogramme	18
2.3.2	Untersuchung des Assemblercodes mit IDA-Pro	21
2.3.3	Sappy - „Audio-Reflektor“	23
3	Emulation mittels mGBA	24
3.1	Was ist der mGBA?	24
3.2	Emulation des Game Boy Advance	24
3.2.1	Abgrenzung der Untersuchung	24
3.2.2	Start des Emulators	25
3.2.3	Initialisierung des „mCore“	27
3.2.4	Laden des ROM	30
3.2.5	Starten des ROM	31
3.2.6	Ausführung des ROM	32
3.3	Emulation des Soundsystems	33
3.3.1	Audioverarbeitung im Assembler	33
3.3.2	Weiterverarbeitung im Emulator	34
3.4	Interaktion mit dem Betriebssystem	35
3.4.1	Start des Emulators	35
3.4.2	Einstellungen über die mGBA GUI	36
3.4.3	Starten des ROM	37
3.4.4	Transferierung der Audiodaten	38
4	Zusammenfassung	39
4.1	Inhalt des Dokumentes	39
4.2	Fazit zur Studienarbeit	39

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 Schubladen der damaligen 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.



Abb. 1: Game Boy Advance - 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 2.2 und Abschnitt 2.3)
- Untersuchung der Fragestellung mit Hilfe eines Beispielprogrammes (siehe Abschnitt 2.3.2)
- 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, Sappy
- **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
<code>0x060</code>	1	DMG Sweep control	<code>SOUND1CNT_L</code>
<code>0x062</code>	1	DMG Length, wave and envelope control	<code>SOUND1CNT_H</code>
<code>0x064</code>	1	DMG Frequency, reset and loop control	<code>SOUND1CNT_X</code>
<code>0x068</code>	2	DMG Length, wave and envelope control	<code>SOUND2CNT_L</code>
<code>0x06C</code>	2	DMG Frequency, reset and loop control	<code>SOUND2CNT_H</code>

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 diene 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% 01: 50% 10: 100% 11: forbidden	SDS_DMG25 SDS_DMG50 SDS_DMG100
2	AV	Direct Sound A Volume Ratio 50% if clear 100% if set	SDSA50 SDSA100
3	BV	Direct Sound B Volume Ratio 50% if clear 100% if set	SDSB50 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 Use timer 1 (if set) for Direct Sound A	SDS_ATMR0 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 Use timer 1 (if set) for Direct Sound B	SDS_BTMR0 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 Videospielekonsolen spezialisiert hat. Das Ziel dabei ist es Hobby und Amateur Videospieleentwicklern 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 Videospieleentwicklung übertragen.

Angefangen hat alles im Jahre 2003 mit dem Cross-Compiler namens devkitARM, welcher es ermöglichte Spiele für den Game Boy Advance zu entwickeln. Mittlerweile umfasst das Angebot nicht nur den Game 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 Videospielekonsolen, sondern sie ist bis dato auch die beliebteste unter den Toolchains. So kamen 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

Belogic ist eine Webseite welche einen Audio Development für Game Boy Advance Spiele näher bringt. Die Seite ist insofern Hilfreich, da einen nicht nur die ganzen Sound Register erklärt werden, sondern auch jeder der 6 Channels. Zusätzlich wird zu jedem der 6 Channels auch ein Beispiel ROM mit entsprechendem Sourcecode zu Verfügung gestellt.

Im Sourcecode gibt es auch ein entsprechenden Header, bei welchem schon alle Register und Sound Channels vordefiniert sind.

```
1  ...
2  #define SOUND1PLAYONCE      0x4000    // play sound once
3  #define SOUND1PLAYLOOP      0x0000    // play sound looped
4  #define SOUND1INIT          0x8000    // makes the sound restart
5  #define SOUND1SWEEPSHIFTS(n)  n        // number of sweep shifts (0-7)
6  #define SOUND1SWEEPINC       0x0000    // sweep add (freq increase)
7  #define SOUND1SWEEPDEC       0x0008    // sweep dec (freq decrease)
8  #define SOUND1SWEEPTIME(n)   (n<<4)   // time of sweep (0-7)
9  #define SOUND1ENVSTEPS(n)    (n<<8)   // envelope steps (0-7)
10 #define SOUND1ENVINC         0x0800    // envelope increase
11 #define SOUND1ENVDEC         0x0000    // envelope decrease
12 #define SOUND1ENVINIT(n)     (n<<12)   // initial envelope volume (0-15)
13  ...
```

Snippet 1: 1. Ausschnitt aus Belogic gba.h

So erleichtert es einem das Programmieren auf Dauer doch sehr Erheblich, wenn man anstelle von „*(vu16*)0x4000064 = 0x8000“, um den Sound Neustarten zulassen, „REG_SOUND1CNT_X = SOUND1INIT“ schreiben kann.

```

1  ...
2  #define REG_SOUND1CNT      *(vu32*)0x4000060    //sound 1
3  #define REG_SOUND1CNT_L   *(vu16*)0x4000060    //
4  #define REG_SOUND1CNT_H   *(vu16*)0x4000062    //
5  #define REG_SOUND1CNT_X   *(vu16*)0x4000064    //
6
7  #define REG_SOUND2CNT_L   *(vu16*)0x4000068    //sound 2 lenght & wave duty
8  #define REG_SOUND2CNT_H   *(vu16*)0x400006C    //sound 2 frequency+loop+reset
9  ...

```

Snippet 2: 2. Ausschnitt aus Belogic gba.h

2.2.3 „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

Die Daten müssen nicht Zwangsläufig in dieser Reihenfolge in einer ROM gespeichert sein und es ist ebenso möglich, dass andere Daten dazwischen verschachtelt sind.

Voice Tabelle

Eine Voice Tabelle ist eine Ansammlung Zeigern zu Instrumenten, welches der Song benutzen kann. Diese besteht für gewöhnlich aus 127 Instrumenten, obwohl manche davon ungenutzt sein können.

So beinhaltet dieser unter anderen Sampled Instrument, PSG Instrument, Sub-Instrument, Key-Split Instrumente und Every-Key-Split Instrumente.

Definition der Generischen Instrumenten Daten

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.

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

Sampled Instrument / Sub-Instrument Format

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

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 Lautstärke)
1	Release-Wert (8-bit : 0x00 = sofortiger Release, 0xFF = längster Release)

Tabelle 9: Sampled Instrument / Sub-Instrument Format

PSG Instrument / Sub-Instrument Format

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

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 Lautstärke)
1	Release-Wert (8-bit : 0x00 = sofortiger Release, 0xFF = längster Release)

Tabelle 10: PSG Instrument / Sub-instrument Format

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 wäre 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, gehört 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.

Key-Split/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: Key-Split/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:

$12 * \text{KeySplitTable}[\text{MidiKey}] + \text{Zeiger auf ersten Sub-Instrument}$

Every Key-Split

Die Position des benutzten Sub-Instrumentes ist:

$12 * \text{MidiKey} + \text{Zeiger auf Percussion Tabelle.}$

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, 0x0F, 0x00

Aus irgendeinem Grund werden nicht genutzte Instrumente in der Sappy Sound Engine immer gleich definiert. Würde diese mutwillig Abgespielt werden, würde es sich um ein PSG Square 1 ohne richtig aktivierten Sweep handeln (tiefe Oktaven werden nicht abgespielt).

16-Byte programmierbare 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

Die konstante Erhöhung bis 0xFE und dann konstanter Fall des Wertes, wird eine Triangle Wave produzieren.

Der Game Boy Advance besitzt auch ein 64 Sample-Modus, jedoch keine bekannte Möglichkeit diesen Modus auszulösen.

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.

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 nicht 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.

Allen Anschein nach weiß das Spiel beim Start eines Songs nicht in welcher Gruppe es Anfängt, dies wird von der Song Zeiger Struktur entschieden. Danach kann aber Volume, Panning und andere Einstellungen durch Referenzen zur Gruppe, in welcher die Songs spielen, verändert werden.

Jeder Track in jeder Gruppe hat sein eigenen Satz an Zeigern und Zähler welche von der Sound Engine genutzt werden und sich im RAM befinden. Je mehr Tracks und Track Gruppen für ein Spiel benötigt werden, desto mehr RAM wird beansprucht.

Die Track Gruppe beansprucht dabei im IWRAM, welches im Game Boy Advance auf 0x3000000 abgebildet ist, 12 Bytes an Zeiger.

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.

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

Tabelle 13: Song Zeiger Format

Sample Format

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

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

Die 8-Bit signed PCM Sample Datei folgt umgehend.

Der Pitch Adjustment Wert wird mit folgender Formel berechnet:

$\text{Pitch Adjustment} = 1024 * \text{Sample-rate für Mid-C (Mid-C entspricht dem MIDI-Key \#60)}$

Das äquivalent von Pitch Adjustment zu verfügbare Engine Sampling Rate (für mid-C Note) sind wie folgt:

0x599800	(5734 Hz)	0x1488000	(21024 Hz)
0x7B3000	(7884 Hz)	0x1A21800	(26758 Hz)
0xA44000	(10512 Hz)	0x1ECC000	(31536 Hz)
0xD10c00	(13379 Hz)	0x2376800	(36314 Hz)
0xF66000	(15768 Hz)	0x2732400	(40137 Hz)
0x11BB400	(18157 Hz)	0x2910000	(42048 Hz)

Diese Werte werden oft für Percussion Sounds und Soundeffekte genutzt, welche normalerweise bei Engine Sample Rate abgespielt werden.

Wenn der Loop genau die Größe von einer Schwingung einer geschleiften Waveform hat, kann der richtige Pitch auch wie folgt berechnet werden:

$\text{Pitch} = 267905 * (\text{Loop_Ende} - \text{Loop_Start})$

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 Original-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

Die Priorität wird wie Folgt gehandhabt:

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

- 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.

Sound Mixer

Die gesampelte Sounds werden in ein oder zwei Direct Sound Channels, je nach Mono oder Stereo Operation, gemixt. Dabei benutzten verschiedene Spiele verschiedene Mixer, so ist es Möglich sein eigenen Mixer mit gewünschte Features in ARM Assembly zuschreiben. Der Mixer simuliert dabei ein Digital Signal Processor (DSP) mit Sound Channels und produziert dabei eine Serie an Nummern im RAM Buffer für die dazugehörigen Output Waveforms.

Der Code für die meisten Sound Engine verwandten Routinen sind im ROM vor den Instrumenten und Sequenzen. So wird der Sound Mixer normalerweise von dort aus in den IWRAM (0x3000000-0x3007FFF) kopiert und dort Frame für Frame aufgerufen. Dies wird in den IWRAM kopiert, damit die CPU vom Game Boy Advance die Befehle schneller aufgerufen kann. Da die Sound Verarbeitung sehr komplex ist und zu jeden Frame passieren muss, beansprucht dieser die CPU stark.

Es gibt mehrere Variationen des Sound Mixers, darunter sind die drei üblichsten Mixer die folgende Versionen:

- Die Stereo Version ist 256 Words lang und nutzt 2 Buffers für Stereo Output
- Die Mono Version ist 224 Words lang und nutzt 1 Buffer für Mono Output (ist auch schneller)
- Die ältere Version ist 256 Words lang und ist ähnlich zur Stereo Version, aber ist langsamer. Nur alte Game Boy Advance Spiele (Veröffentlicht im Jahr 2001) nutzen diese.

Die Features des normalen Sound Mixer sind folgende:

- Relativ langsames Mischen
- Lineare Interpolation wird benutzt
- Kein Anti-Aliasing
- Jeder Sample wird direkt auf 8-Bit herunter skaliert und in den Output Buffer gemischt, welches in einen lauten Output resultiert
- Kein Saturation Output Control: Wenn der Output über den maximalen Wert geht, produziert es ein unangenehmes „Klick“ Geräusch

Verschiedene Features können dem Mixer hinzugefügt werden. Zum Beispiel:

- Komprimierte Samples (Pokemon)
- Synth Instruments (Camelot)
- Schnelleres mischen mit selbst-modifizierbarer Code (Camelot, Final Fantasy Advance Sound Restoration Hacks)
- Mischen von Instruments in 16-Bit, bevor auf 8-Bit herunter skaliert wird, welches bessere Soundqualität (Camelot, Final Fantasy Advance Sound Restoration Hacks)
- Besserer Nachhall (Camelot, Final Fantasy Advance Sound Restoration Hacks)
- Schutz gegen Sättigung vom Output (Camelot, Final Fantasy Advance Sound Restoration Hacks)
- Bessere Attack, Decay, Sustain und Release Envelope mit 16-Bit anstelle von 8-Bit (Camelot, Final Fantasy Advance Sound Restoration Hacks)
- Samples umgekehrt abspielen
- Ping-Pong und umgekehrtes Sample looping
- Bessere Interpolation / Anti-Aliasing

Mit Camelot sind die Spieleentwickler gemeint, welche unter anderem die Golden Sun Serie für den Game Boy Advance entwickelt haben.

Final Fantasy Advance Sound Restoration Hacks steht für die ROM Hacks, welche ROM Hacker für die Final Fantasy Portierungen auf dem Game Boy Advance entwickelt haben. Denn im Vergleich zu den Originalen auf dem Super Nintendo Entertainment System ist die Soundqualität der Game Boy Advance Ports sehr schlecht. Dieser Hack restauriert die Musik und Soundeffekte von den Game Boy Advance Spielen wieder zu den Originalen vom Super Nintendo Entertainment System.

2.3 Softwareentwicklung

Autor: Ngoc Luu Tran

2.3.1 Erstellung der Beispielprogramme

Autor: Ngoc Luu Tran

Erstellung der Beispielprogramme anhand der Belogic Channel 1 Demo und Belogic Direct Sound Demo:

Belogic Channel 1 Demo:

Die Demo zeigt alle Features von Sound Channel 1. So erlaubt die Demo die Bitweise-änderung von den REG_SOUND1CNT_L, REG_SOUND1CNT_H und REG_SOUND1CNT_X. Der Hörbare Unterschied ist bei Channel 1 jedoch sehr Subtil oder nicht bemerkbar.

```
1    ...
2    u16 note;
3    u16 delta,u,sweepshifts=2,sweepdir=1,sweeptime=7,cur=6;
4    u16 envinit=0xf, envdir=0, envsteptime=7,waveduty=2,soundlength=1;
5    u16 loopmode=0,sfreq=0x400,resamp=1;
6
7    ...
8    if(~REG_KEYPAD&BUTTON_A) //play the sound
9    {
10    REG_SOUND1CNT_L=(sweeptime<<4)+(sweepdir<<3)+sweepshifts;
11    REG_SOUND1CNT_H=(envinit<<12)+(envdir<<11)+(envsteptime<<8)+(waveduty<<6)+soundlength;
12    REG_SOUND1CNT_X=SOUND1INIT+(loopmode<<14)+sfreq;
13    }
14    ...
```

Snippet 3: Belogic Channel 1 Demo

Der Sound Channel 1 produziert eine Square Wave mit Envelope und Frequenzdurchlaufs-Funktion.

Im Belogic Channel 1 Demo ROM übergeben wir am Anfang dem **REG_SOUND1CNT_L**, welcher für den Frequenzdurchlauf zuständig ist, die 0x7A. Das Bedeutet für den REG_SOUND1CNT_L ein Sweep Shift von 2, Sweep decrease und Sweep Time von 54.7 ms.

REG_SOUND1CNT_H, welches die 0xF781 zugewiesen wird, ist für den Duty Cycle, die Länge und den Envelope zuständig. Bei unserem Beispiel bedeutet das eine Sound Länge von 1, Wave Duty Cycle von 50%,Envelope Step Time von 3, Envelope decrement und Initial Envelope Value von 0xF, welches maximale Lautstärke bedeutet.

REG_SOUND1CNT_X beinhaltet die Frequenz und die Kontrolle für Looped oder Timed Modus. Im Beispiel von 0x8400 bedeutet das eine Frequenz von 32,8 Khz, Loop und Sound Reset an.

Belogic Direct Sound Demo:

Das Programm gibt bei Knopfdruck (A,B,L,R,Start und Select) ein Sample über den Direct Memory Access Sound Channel wieder.

```
1  ...
2  int AgbMain(void){
3  // play a sound using timer 0 as sampling source
4  // When timer 0 overflows, the interrupt handler loads the FIFO with the next sample
5
6  init(); //set graphic mode 4
7  LoadPic((u16*)_binary_logo_raw_start,(u8*)logopal); //load logo
8
9  Print(5,2,"Direct Sound Mono Playback Demo");
10 Print(6,19,"(c)2001 www.Belogic.com / Uze");
11 TimerPlaySound(); //play in interrupt mode
12
13 while(1){
14     if(~REG_KEYPAD&0x3ff){
15         REG_TMOCNT_H=0; //disable timer 0
16         DmaPlaySound(); //play in dma mode
17     }
18 };
19 }
20 ...
```

Snippet 4: Belogic Direct Sound Demo

Das Spiel startet und landet in einer Schleife, welche auf einen Tastendruck wartet. Wird eine Taste gedrückt, ruft diese dann die DmaPlaySound() Funktion auf.

```
1  ...
2  void DmaPlaySound (void){
3  //Play a mono sound at 16khz in DMA mode Direct Sound
4  //uses timer 0 as sampling rate source
5  //uses timer 1 to count the samples played in order to stop the sound
6  REG_SOUND CNT_L=0;
7  REG_SOUND CNT_H=0x0b0F; //DS A&B + fifo reset + timer0 + max volume to L and R
8  REG_SOUND CNT_X=0x0080; //turn sound chip on
9
10 REG_DMA1SAD=(unsigned long)_binary_lo1234_pcm_start; //dma1 source
11 REG_DMA1DAD=0x040000a0; //write to FIFO A address
12 REG_DMA1CNT_H=0xb600; //dma control: DMA enabled+ start on FIFO+32bit+repeat
13
14 REG_TM1CNT_L=0x7098; //0xffff-the number of samples to play
15 REG_TM1CNT_H=0xC4; //enable timer1 + irq and cascade from timer 0
16
17 REG_IE=0x10; //enable irq for timer 1 overflow
18 REG_IME=1; //enable interrupt
19
20 //Formula for playback frequency is: 0xFFFF-round(cpuFreq/playbackFreq)
21 REG_TMOCNT_L=0xFBE8; //16khz playback freq
22 REG_TMOCNT_H=0x0080; //enable timer at CPU freq
23 }
24 ...
```

Snippet 5: Belogic DmaPlaySound()

Das Register **REG_SOUND_CNT_L** kontrolliert nur den Dot Matrix Game (DMG) Output Verstärker und hat sonst keinen Einfluss auf die Verarbeitung der Sound Channels oder der Direct Sound Lautstärke. So beinhaltet dieses Register auch Bits für Vin Rechts und Links, welches Spielkassetten vom Game Boy Classic berechtigten ihre eigenen Sound Quellen zu verwenden. Jedoch ist es nicht Bekannt, ob diese Funktion noch vom Game Boy Advance unterstützt wird oder noch funktioniert.

REG_SOUND_CNT_H bekommt im Sourcecode den Wert 0x0B0F. Dies bedeutet ein Output Sound Verhältnis für Channel 1 bis 4 von 100%, ein Direct Sound A&B Verhältnis von 100% und für Direct Sound A eine Ausgabe für Links und Rechts mit FIFO Reset und Timer 0.

Dabei kontrolliert Output Ratio die ausgegebene Lautstärke, wenn die DMG oder Direct Sound Channels im Verhältnis zueinander zu Laut sind.

Der Direct Sound ist ein dual 8-bit Digital-To-Analog Converter (DAC), welche die Daten von zwei FIFO's bekommt. Die FIFO's können manuell oder automatisch in DMA Modus geladen werden, dabei verwendet der DMA Modus die Timer als Abtastfrequenzbezug.

REG_SOUND_CNT_X schaltet den Sound Chip (DMG und Direct Sound) an oder aus. Dieser sollte so oft wie nur möglich aus sein um Batterieleistung zu sparen, dies ermöglicht es Batterien bis zu 10% länger zuhalten.

Danach wird **REG_DMA1SAD**, welches die Source Adresse angibt, die Sound Datei zugewiesen und dann man mit dem **REG_DMA1DAD** Register die Ziel Adresse (In unserem Fall Channel A FIFO Register) bestimmt. Dann muss noch durch **REG_DMA1CNT_H** der DMA aktiviert und das Start Timing auf Sound FIFO gesetzt werden. Zuallerletzt müssen noch die benötigten Timer gesetzt und aktiviert werden.

2.3.2 Untersuchung des Assemblercodes mit IDA-Pro

Autor: Ngoc Luu Tran

Untersuchung des Assemblercodes der Sappy Sound Engine anhand von Pokémon Blattgrün mit Hilfe des Disassembler IDA-Pro:

Um in IDA Pro die einzelnen Songs zu finden, wird zunächst nach der Songtabelle gesucht.

Diese findet man, indem man nach der HEX-Reihenfolge von ungenutzten Instrumenten sucht. Dabei handelt es sich um folgende Reihenfolge:

0x01, 0x3c, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00,
0x00, 0x00, 0x0f, 0x00 (1.)

Wird die Sappy Sound Engine, wie bei Pokémon Blattgrün, vom Spiel genutzt, sollten mehrere dieser Hex-Strings gefunden werden. Scrollt man jetzt von der letzten übereinstimmung der HEX-Reihenfolge aus weiter nach unten, sollten wir Spalten mit 0x00 und 0x08 auffinden. (2.)

Bei den ersten 4 Bytes welche mit 0x08 enden, handelt es sich um ein Zeiger auf den ersten Song im Spiel. (3.) Dieser befindet sich auf der ersten Position der Songtabelle. In der Abbildung 2 handelt es sich dabei um den Zeiger auf die Adresse „0x086B4F1C“.

Wird jetzt von der HEX-Ansicht auf die IDA-Ansicht gewechselt, können die einzelnen Zeiger auf die Songs erkannt werden. (4.) Folgt man nun einer dieser Zeiger landet man bei 0xB2, welches für eine Sprung-Anweisung steht, gefolgt von der Adresse des Instruments. (5.)

Nach der Adresse des Instruments folgt der Song-Header. (6.) Dieser führt zum Trackformat, welcher immer mit der „Tempo“-Anweisung 0xBB Anfängt und mit der „Ende des Songs“-Anweisung 0xB1 endet.

So wird in Abbildung 3 zunächst das Tempo (0xBB) auf 33 gesetzt und das 91. Instrument (0xBD) ausgewählt. Danach wird die Lautstärke (0xBE) auf 100 gesetzt und die Tonhöhe (0xC0) auf 58 variiert. Anschließend folgt eine Note mit automatischem Time-Out mit 60 112 132, dann ein Instrumenten Wechsel und endet mit der Ende des Songs Anweisung (0xB1).

Für genauere Bedeutung der Anweisungen, siehe „Sappy Track Format“-Tabelle 15.

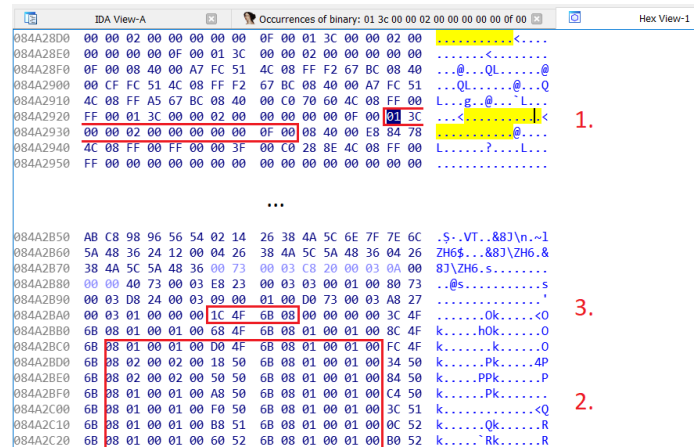


Abb. 2: Anfang der Songtabelle in Hex-Ansicht

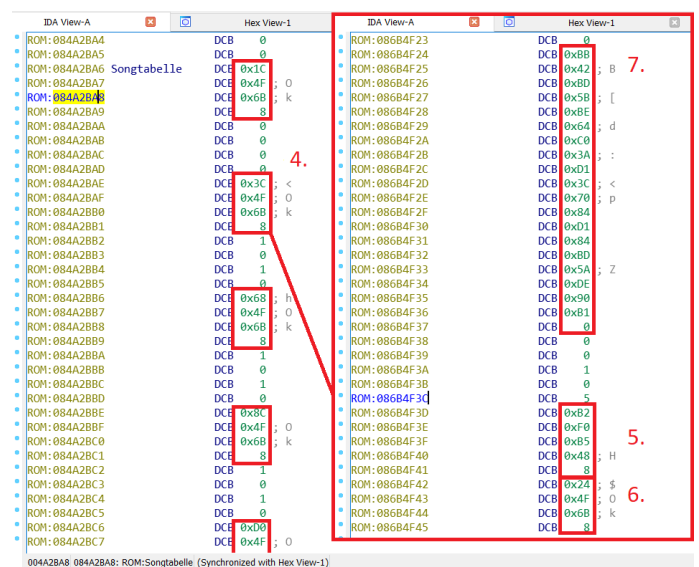


Abb. 3: IDA-View der Songtabelle und erstem Song

Folgt man der Adresse des Instrumentes, landet man bei der Voice-Tabelle. (8.) Dieser besteht typischerweise aus 127 Instrumenten, welche jeweils 12 Bytes pro Eintrag beanspruchen.

In der Abbildung 4 erkennt man dass es sich in diesem Beispiel um ein Sample handelt, dieser hat einen Zeiger auf die Sample Datei (0x084ED548). Weiterhin hat es einen Attack-Wert von 255 (0xFF), einen Decay-Wert von 249 (0xF9), einen Sustain-Level von 103 (0x67) und Release-Wert von 165 (0xA5) hat. Daraufhin folgen zwei nicht genutzte Instrumente, welche man an der bereits Bekannten HEX-Reihenfolge erkennen kann. Für genauere Informationen, siehe „Sappy Sample Instrument“-Tabelle 9.

Die Sample Datei kommt mit einem 16-Byte Header (9.), gefolgt von einer variablen Datenlänge. So handelt es sich in der Abbildung 4 um ein geschleiftes Sample mit Pitch Adjustment von 0xAC4400 (11025 Hz).

Für genauere Informationen, siehe „Sappy Sample Instrument“-Tabelle 14.

Die Song-Auswahl-Funktion für Sappy-Sound-Engine schaut in IDA aus wie in der Abbildung 5.

In der Proximity-Browser-View wird klarer, dass es sich um die Song-Auswahl-Funktion handelt, als in der Graph-View.

Diese kann man leicht finden, indem man nach Folgen der HEX-Reihenfolge sucht:

0x00, 0xB5, 0x00, 0x04, 0x07, 0x4A, 0x08, 0x49, 0x40, 0x0B, 0x40, 0x18, 0x83, 0x88, 0x59, 0x00, 0xC9, 0x18, 0x89, 0x00, 0x89, 0x18, 0x0A, 0x68, 0x01, 0x68, 0x10, 0x1C, 0x00, 0xF0

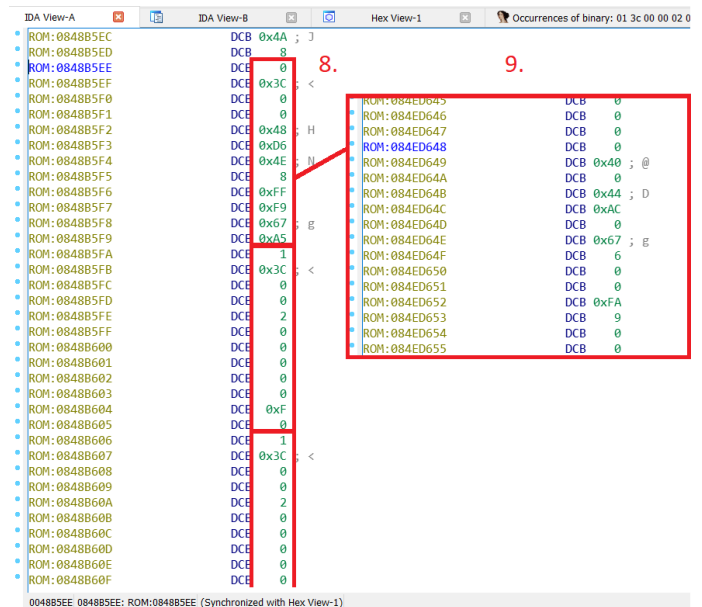


Abb. 4: IDA-View von Voice-Table und Sample-Format

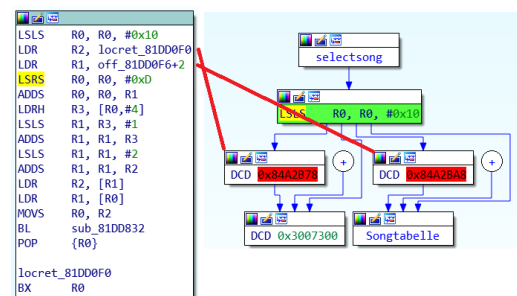


Abb. 5: IDA-Graph-View und Proximity-Browser-View für SelectSong Funktion

2.3.3 Sappy - „Audio-Reflektor“

Autor: Ngoc Luu Tran

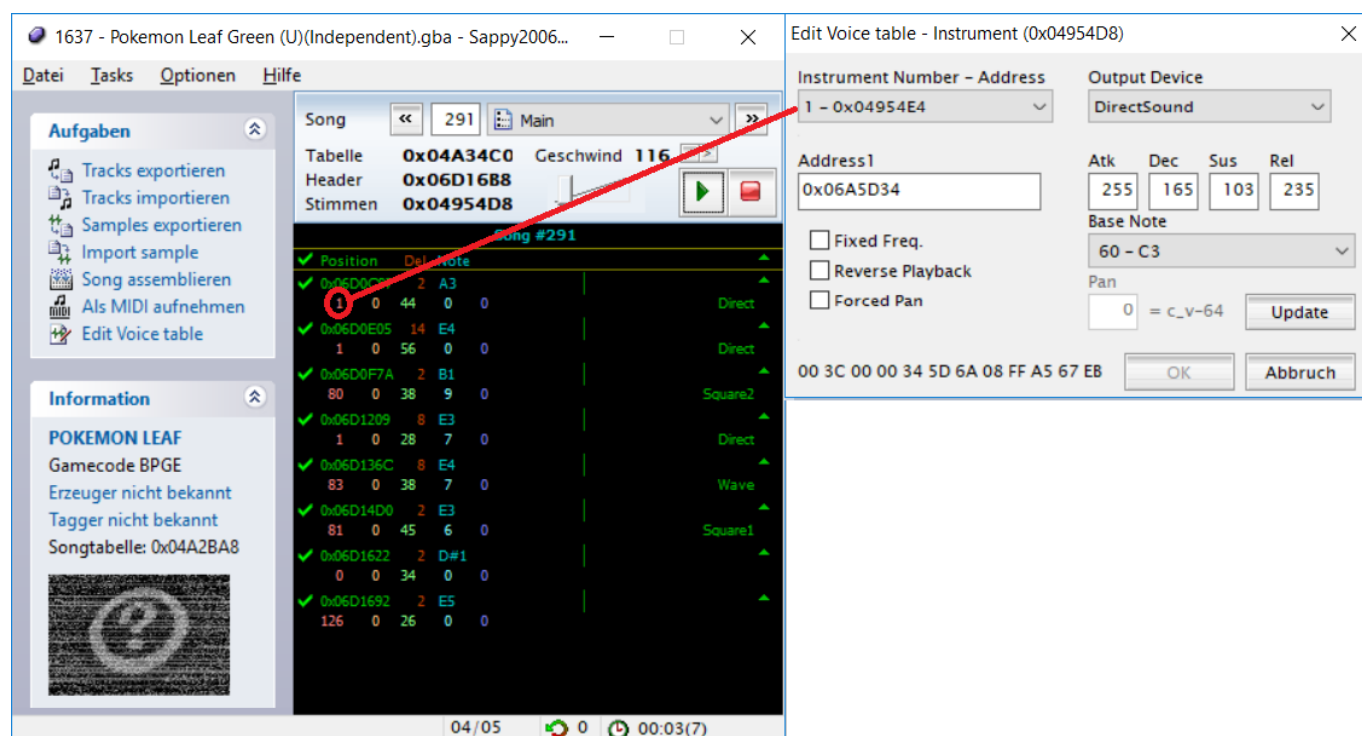


Abb. 6: Sappy Audio-Reflektor

Sappy ist ein Programm um Musik von Game Boy Advance Spielen, falls sie die Sappy Sound Engine nutzen, zu extrahieren. Nach auswählen eines Spieles werden die Songs und deren Instrumente automatisch gefunden.

Zu anderen Anwendungen gehören das Abspielen von den gefundenen Songs, Bearbeitung, Entfernen und Hinzufügen von Tracks und Instrumente oder das Konvertieren in MIDI-Dateien.

So sieht man in der Abbildung 6, dass man im Vergleich zu IDA Pro viel einfacher die Gesuchten Daten findet. Diese werden unter anderem auch kompakter Angezeigt, welches man zum Beispiel an der Voice-Table merkt. So sieht man nicht nur alles auf einem Blick, sondern kann auch durch ein Drop-Down-Liste leicht zwischen Instrumente wechseln, oder die Werte ändern.

3 Emulation mittels mGBA

Autoren: Dominik Scharnagl, Florian Boemmel

3.1 Was ist der mGBA?

Autor: Dominik Scharnagl

„mGBA ist ein Open-Source Game Boy Advance Emulator, der von endrft entwickelt wurde. Von Grund auf neu geschrieben, zielt die Anwendung auf Geschwindigkeit, Genauigkeit und Portabilität ab. Bis jetzt ist mGBA der umfassendste GBA-Emulator, der das ältere Projekt VBA (Visual Boy Advance) und dessen Forks überstanden hat [...]“ [6] Das Kürzel „GBA“ steht dabei für Game Boy Advance.

„mGBA ist eine neue Generation des Game Boy Advance Emulators. Das Projekt startete im April 2013 mit dem Ziel, schnell genug auf einer schwächeren Hardware zu laufen als andere Emulatoren, ohne auf Genauigkeit oder Portabilität zu verzichten. Schon in der ersten Version ließen sich Spiele generell ohne Probleme spielen. mGBA ist seither immer besser geworden und rühmt sich nun, der genaueste GBA-Emulator zu sein.

Weitere Ziele sind eine ausreichend genaue Emulation, um eine Entwicklungsumgebung für Homebrew-Software bereitzustellen, ein guter Workflow für Tool-Assist-Runner und ein moderner Funktionsumfang für Emulatoren, den ältere Emulatoren möglicherweise nicht unterstützen.

mGBA ist unter der Mozilla Public License 2.0 lizenziert und [...] kann auf GitHub gefunden werden.“ [7]

Für was steht das „m“?

„[...] mGBA sollte ursprünglich miniGBA heißen, aber als das Projekt wuchs, wurde der Begriff unpassend. Der Name sollte nur temporär sein, aber als die erste veröffentlichte Version näher kam, konnte ich mir keine besseren Namen vorstellen. Andere Projektnamen für mGBA waren GBAC und Gerboa, aber nichts anderes blieb.“ [8]

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 Sprachen C und 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 Abläufen die zur Emulation des Soundsystems notwendig sind.

3.2.2 Start des Emulators

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.

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.

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 6.

```
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 6: 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`

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 mittels 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`

3.2.3 Initialisierung des „mCore“

Autor: Dominik Scharnagl

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.

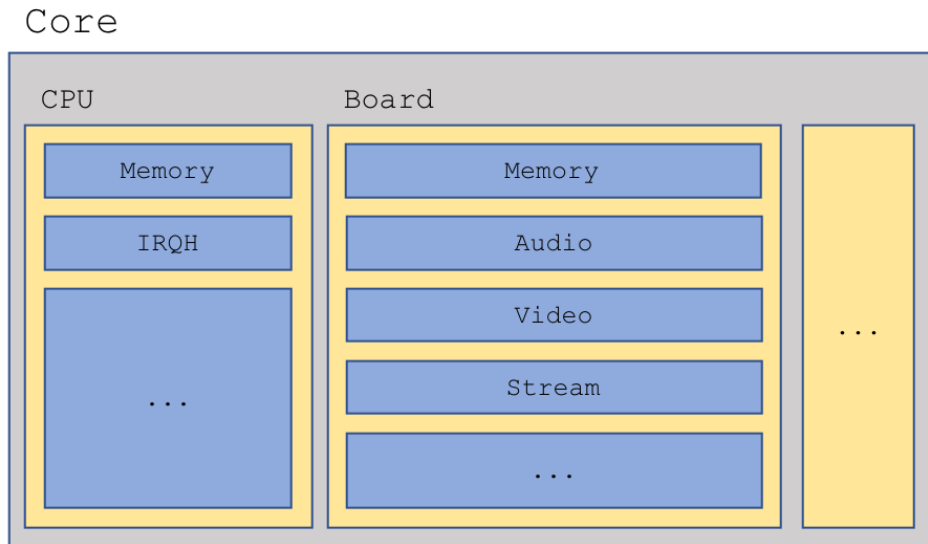


Abb. 7: Struktur des GBACore

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 allokiert. Eine abstrakte Darstellung der Struktur ist in Abbildung 7 zu finden. 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. 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 gehört 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 7: 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 Untersuchung 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 8: Ausschnitt aus der **GBAMemoryInit**-Methode

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

In Abbildung 8 ist die abstrakte Struktur der **GBAAudio** zu sehen. Das Feld `psg` wird dabei für eine Instanz der **GBAAudio** Struktur verwendet. Die Felder `ch1` und `ch2` dienen dabei als FIFOs für die je Kanal anliegenden Audiodaten. Die eigentlichen Kanalspezifischen Audioeinstellungen zur Tongenerierung finden sich jedoch im `psg`-Feld. Dort wird je Kanaltyp die entsprechende Steuerung vorgenommen.

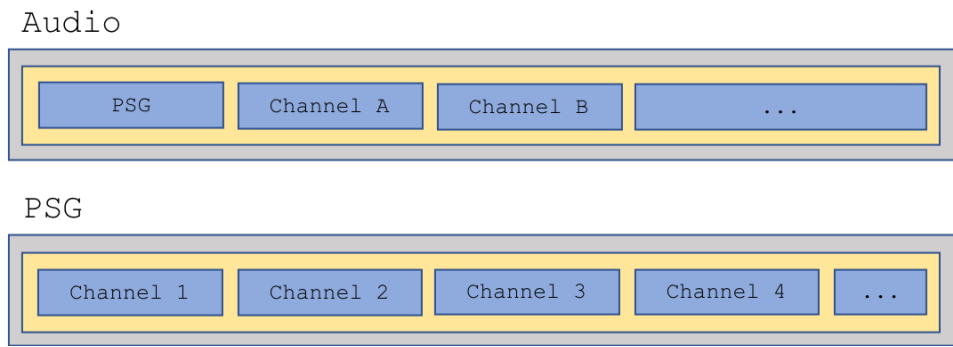


Abb. 8: Struktur des GBAAudio

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

Die Verarbeitung der definierten Ereignisse zur Audioverarbeitung findet durch Aufruf der Methode im Feld `processEvents` des **mCore** statt. Nach Durchlaufen der globalen **GBACoreCreate**-Methode ist das die globale Methode `_GBAProcessEvents`. Die Ausführung der Methode mündet in die **mTimingTick** welche letztendlich die callback-Methoden (siehe Tabelle 18) der einzelnen Ereignisse anstößt.

3.2.4 Laden des ROM

Autor: Dominik Scharnagl

Wurde die Initialisierung des gesamten **mCore** abgeschlossen und die für die Emulation notwendigen Strukturen erzeugt, kann die eigentliche ROM Datei geladen werden. Ein Teil des Ladevorgangs ist dabei das Setup der zuvor geschaffenen „virtuellen“ Peripherie. Dies geschieht anhand diverser „Setup“-Routinen. Wie dann letztlich der Inhalt der ROM geladen und interpretiert wird, hängt vom Format und dem Inhalt der ROM-Datei ab.

_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**-Feldes 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**-Feldes 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 ausschließlich auf das **audio**-Feld des **board**-Feldes vom Typen der **GBA**-Struktur. Die dabei vorgenommenen Änderungen beziehen sich somit nur auf Felder der **GBAAudio**-Struktur.

3.2.5 Starten des ROM

Autor: Dominik Scharnagl

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 `threadContext`-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`)

Zu Beginn der Methode werden noch weitere Methoden der **mCore**-Struktur ausgeführt. Eine dieser ist die Methode, die im Feld `setSync` eingetragen ist. Nach Durchlaufen der globalen **GBACoreCreate**-Methode ist das die globale Methode `_GBACoreSetSync`. Sie wird dazu verwendet um die **mCoreSync**-Struktur des `m_threadContext->sync`-Feldes dem **mCore** bekannt zu machen. In ihr sind die bereits durch die **mCoreThreadStart**-Methode vorbereiteten Condition- und Mutex-Instanzen zur Thread-Synchronisation enthalten.

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**.

Wurden 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. Der in Abbildung 9 gezeigte Programmablauf veranschaulicht den Ablauf.

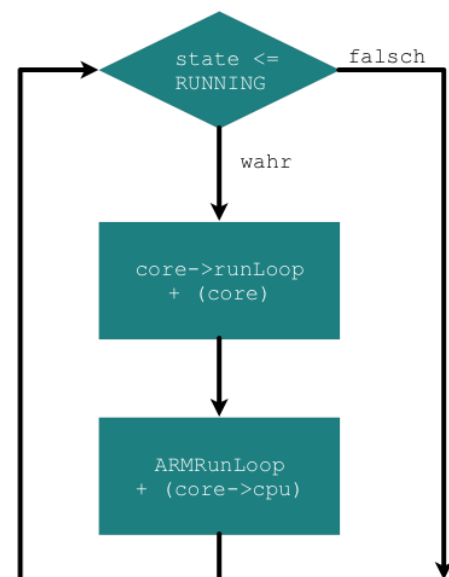


Abb. 9: Ablauf der Methode

3.2.6 Ausführung des ROM

Autor: Dominik Scharnagl

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 abgearbeitet 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 physikalischen 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 Prefetch-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 uint32_t instructionIndex = ((opcode >> 16) & 0xFF0) | ((opcode >> 4) & 0x00F);
12
13 ARMInstruction instruction = _armTable[instructionIndex];
14 instruction(cpu, opcode);
```

Snippet 9: 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 10 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 10: ARM Instruction Makro für **STRT**

Gemäß vorangegangenem Snippet 8 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 globalen **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`)

3.3 Emulation des Soundsystems

Autor: Dominik Scharnagl

Damit die vom ROM liegenden 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.

3.3.1 Audioverarbeitung im Assembler

Autor: Dominik Scharnagl

Im Rahmen der Untersuchungen wurde festgestellt, dass die Audiodaten entweder vorab im ROM vorliegen oder im Assembler generiert werden. Ausschlaggebend für die Audioverarbeitung ist dabei die korrekte Interaktion mit den zuständigen Registern. Im Allgemeinen wird immer zuerst ein Sample mit zugehörigen Audiosetup in die dafür vorgesehenen Register geschrieben. Nachdem alle für ein Sample notwendigen Konfigurationen getroffen wurden, wird über das Master Control Register die Audioausgabe angestoßen.

Wurde ein Sample vom „AudioProcessor“ verarbeitet, kann das Programm den nächsten Sample in die Register laden und wiederholt die Verarbeitung anstoßen. Im Wesentlichen basiert die Audioverarbeitung im Assembler primär darauf, dass Samples entweder getaktet oder mittels FIFO bereitgestellt werden. Liegen die Daten bereit, werden diese anschließend als „bereit zur Ausgabe“ markiert. Erst nach erfolgter Ausgabe wird der nächste Sample bereitgestellt, beziehungsweise noch während der Ausgabe aus einem FIFO dieser kontinuierlich weiter befüllt.

Die für die Verarbeitung notwendigen Register wurden bereits im Abschnitt 2.1 genannt. Essentiell sind aufgrund der eben beschriebenen Operationen Assembler-Befehle, die zum Laden und Speichern von Daten in und aus den besagten Registern notwendig sind. Ein Beispiel-Befehl ist hierfür der **STRT**-Befehl, dessen Interpretation im Bezug auf die **ARMStep**-Methode erklärt wurde. Betrachtet man dessen Umsetzung vom Assembler-Code zum C-Code, dann erkennt man die logischen Abläufe, welchen Einfluss ein Assembler-Befehl auf die Ausführung des ROMs im Emulator hat.

3.3.2 Weiterverarbeitung im Emulator

Autor: Dominik Scharnagl

Im Kontext der von der **VideoView** unabhängigen Abläufe 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. Der grobe Ablauf des Synchronisation im **AudioDevice** kann in Snippet 11 betrachtet werden.

```
1 mCoreSyncLockAudio(&m_context->sync);
2 ...
3 blip_read_samples(
4     m_context->core->getAudioChannel(m_context->core, 0),
5     &reinterpret_cast<GBAStereoSample*>(data)->left,
6     ...);
7 ...
8 mCoreSyncConsumeAudio(&m_context->sync);
```

Snippet 11: AudioDevice - „Lock / Consume“

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. Wie in etwa die Methode bei der Synchronisation vorgeht, kann im folgenden Snippet 12 nachvollzogen werden.

```
1 mCoreSyncLockAudio(audio->p->sync);
2 ...
3 blip_add_delta(audio->left, audio->clock, sampleLeft - audio->lastLeft);
4 ...
5 bool wait = produced >= audio->samples;
6 mCoreSyncProduceAudio(audio->p->sync, wait);
7 ...
```

Snippet 12: `_sample` - „Lock / Consume“

3.4 Interaktion mit dem Betriebssystem

3.4.1 Start des Emulators

Autor: Florian Boemmel

Im Kapitel 3.2.2 wurde bereits der Ablauf bis zur Erstellung der AudioProcessor-Klasse, sowie das Setzen des Treibers beschrieben. Auf diesen Ablauf wird im Folgenden nun aufgesetzt. 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.

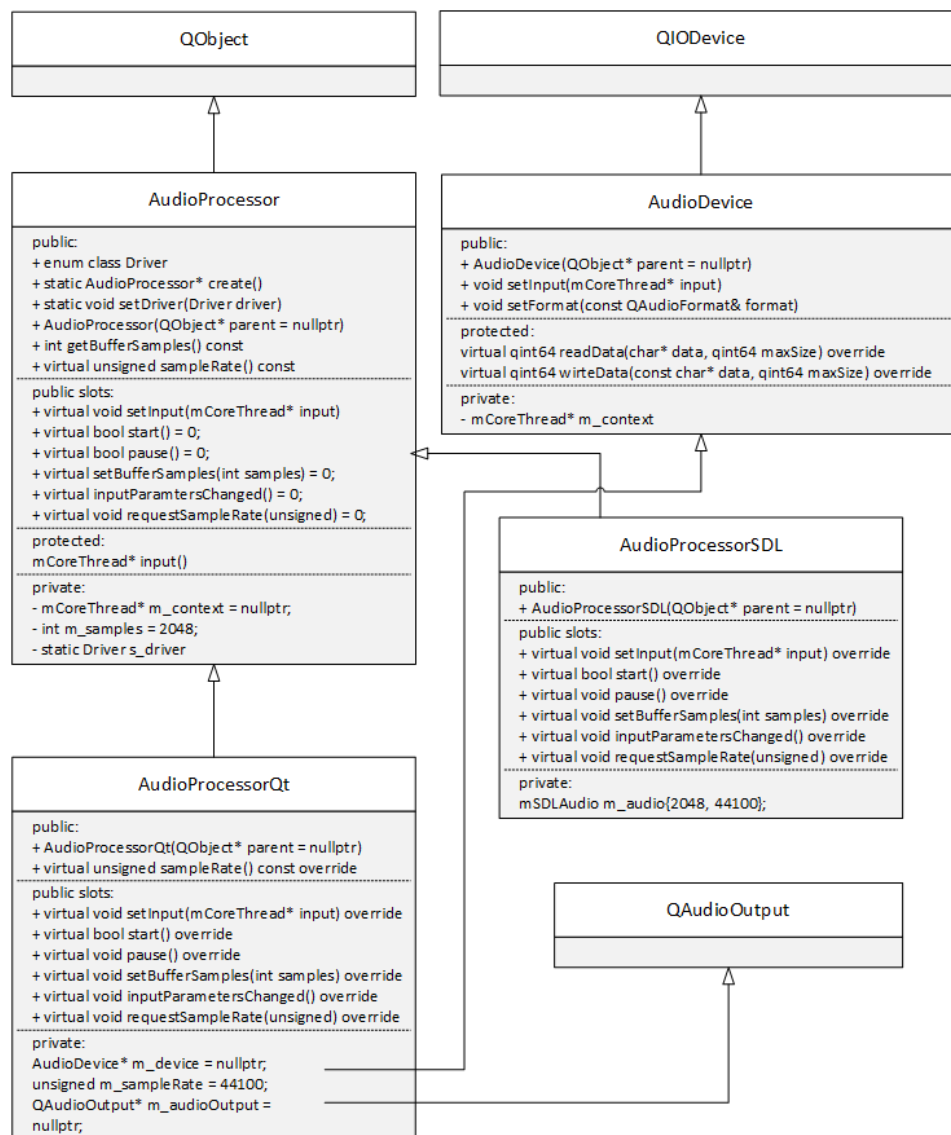


Abb. 10: Audioklassen in der QT-Anwendung

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.

`AudioProcessor::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 **`AudioProcessor::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.

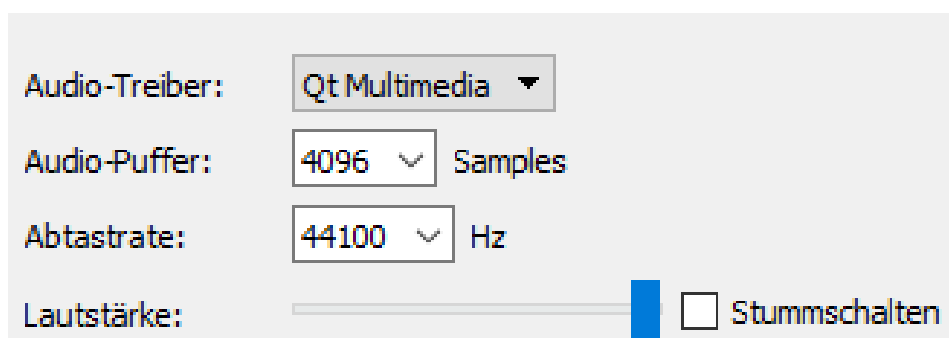


Abb. 11: Audioeinstellungen im mGBA

In Abbildung 11 sind die für den 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 gängigsten Anzahl an Samples von 512 bis 4096. Kann aber auch benutzerspezifisch 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 erhält 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
8  // Little- oder Big-Endian
9  format.setSampleType(QAudioFormat::SignedInt);
10
11 // Sample Typ
12 m_audioOutput = new QAudioOutput(format, this);
13 m_audioOutput->setCategory("game");
```

Snippet 13: Ausschnitt aus **AudioProcessorQt::start()**

Wie bereits am Ende von Snippet 13 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, Kopfhörer usw.) Die **setCategory** Methode setzt den Modus auf „`game`“. Einige Plattformen 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 Transferierung 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 verfügbaren Samples im **m_coreThread** abgefragt. Überschreitet die Anzahl der Samples die Größe des Puffers, geteilt durch die Strukturgröße **GBAStereoSample**, wird die Anzahl der verfügbaren 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

Entsprechend der Aufgabenstellung zur Studienarbeit für die Vorlesung Computer Architektur haben die drei Studenten Ngoc Luu Tran, Florian Boemmel und Dominik Scharnagl in kooperativer Arbeit die Aufgabenstellung bearbeitet. Im Rahmen dieser Arbeit war es Ziel die Emulation des Soundsystems eines Game Boy Advance Emulators zu untersuchen. Die dabei durchgeführte Untersuchung nahm das Open-Source Projekt mGBA als Ausgangspunkt zur Beschreibung aller notwendigen Abläufe der Emulation.

Eingangs werden die Hardwareanforderungen an den Emulator grob umrissen. Während darauf die verfügbaren Plattformen zur Entwicklung eines Game Boy Advance Spiels erläutert werden, folgte jeweils ein kurzer Einblick in diese. Ziel des darauf anschließenden Kapitels Softwareentwicklung ist es einen tieferen Einblick in die technischen Anforderungen zur Entwicklung einer Game Boy Advance Software zu geben, welche Soundausgaben durchführt.

Im Gegensatz zum Kapitel „Game Boy Advance“, indem ausschließlich die Plattform hardware- als auch software-technisch durchläuchtet wird (bis in den Assembler-Code), steht das Kapitel „Emulation mittels mGBA“. Ziel dieses Kapitels ist es eine logische Trennung der notwendigen Softwarekomponenten zur Emulation zu bekommen. Zugleich wird am Beispiel des mGBA-Projektes veranschaulicht, welche Aufwände bei der Interpretation von OpCodes über die Ausführung ihrer Anweisungen bis hin zur Audioausgabe anfallen.

Am Ende der Studienarbeit angekommen, kann man den vollständigen Ablauf der Emulation des Soundsystems eines Game Boy Advance beschreiben und aufgrund der Inhalte des Kapitels „Game Boy Advance“ eine eigene Software für einen Game Boy Advance entwickeln. Wie die Audioverarbeitung von dieser Software durch einen Emulator stattfindet erfuhr man am Beispiel des mGBA-Projektes in Kapitel „Emulation mittels mGBA“.

4.2 Fazit zur Studienarbeit

Startschwierigkeiten gibt es in jedem Projekt, insbesondere bei unbekannten Softwareprojekten. Diese Studienarbeit machte dabei auch keine Ausnahme. Dank diverser Quellen und kurzer Vorführung des Debuggers IDA Pro durch Enrico Pozzobon war es für uns kein gänzlicher Sprung ins kalte Wasser.

Dominik Scharnagl

Der größte Anreiz für mich war weniger zu verstehen, wie ein Game Boy Advance Spiel emuliert wird, sondern viel mehr, wie man einen Emulator - am praktischen Beispiel mGBA - umsetzen und implementieren kann. Das zeigt sich auch in der Wahl der von mir ausformulierten Themen und Abschnitte wieder.

Florian Boemmel

Für mich persönlich war es sehr interessant einen Blick hinter die Kulissen der Audioverarbeitung des mGBA zu werfen. Ich hatte bereits einige male mit grafischen Oberflächen zu tun. Jedoch kam dabei das Thema Audio nicht zum Einsatz, da dieses Thema recht komplex ist. Eine praktische Umsetzung, in diesem Fall der mGBA, zu analysieren brachte mir die Audio-Verarbeitung ein gutes Stück näher.

Ngoc Luu Tran

Als Jemand der noch nie richtige Berührungspunkte mit Reverse Engineering hatte, fand ich das Thema der Studienarbeit sehr Interessant. So war für mich das Arbeiten mit IDA Pro völlig neu und mit viel Trial und Error verbunden. Am Ende konnte ich dann doch trotz fehlendem Wissen über MIDI Dateien vieles über die Verarbeitung und Speicherung von Musik in Game Boy Advance Spielen lernen.

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
- [6] EMUGEN: *mGBA*
<http://emulation.gametechniki.com/index.php/MGBA>, Juni 2018
- [7] mGBA: *About*
<https://mgba.io/about.html>, Juni 2018
- [8] mGBA: *FAQs*
<https://mgba.io/faq.html>, Juni 2018
- [9] Problemkaputt: *GBATEK*
<https://problemkaputt.de/gbatek.htm#gbamemorymap>, Juni 2018

Bilder

- Abbildung 1: *Game Boy Advance - Blue Edition*
<https://d3nevf7k7ii3be.cloudfront.net/igi/L3WryntCMswfDks1.large>, Mai 2018
- Abbildung 2: *Anfang der Songtabelle in Hex-Ansicht*
- Abbildung 3: *IDA-View der Songtabelle und erstem Song*
- Abbildung 4: *IDA-View von Voice-Table und Sample-Format*
- Abbildung 5: *IDA-Graph-View und Proximity-Browser-View für SelectSong Funktion*
- Abbildung 6: *Sappy Audio-Reflektor*
- Abbildung 7: *Struktur des GBACore*
- Abbildung 8: *Struktur des GBAAudio*
- Abbildung 9: *Ablauf der RunLoop-Methode*
- Abbildung 10: *Übersicht der Audioklassen in der Qt Anwendung*
- Abbildung 11: *Audioeinstellungen im mGBA*