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
    a) Untersuchungsgegenstand (Florian)
    b) Verwendete Software (Florian)
2. Game Boy Advance
    a) Hardwareumgebung
        i. Übersicht der Audio-Register (Florian)
        ii. Übersicht der Sound-Master-Register (Florian)
    b) Plattformen
        i. DevkitPro (Ngoc)
        ii. BeLogic (Ngoc)
        iii. MaxMod (Ngoc)
        iv. Snappy (Ngoc)
    c) Softwareentwicklung
        i. API zur Audioverarbeitung (Ngoc)
        ii. Audiodaten im Speicher (Ngoc)
        iii. Snappy - „Audio-Reflektor“ (Ngoc)
        iv. Untersuchung des Assemblers (Ngoc)
3. Emulation mittels mGBA
    a) Was ist der mGBA? (Dominik)
    b) Emulation des Game Boy Advance
        i. Abgrenzung der Untersuchung (Florian)
        ii. Start des Emulators (Dominik)
        iii. Initialisierung des „mCore“ (Dominik)
        iv. Laden des ROM (Dominik)
        v. Starten des ROM (Dominik)
        vi. Ausführung des ROM (Dominik)
    c) Emulation des Soundsystems
        i. Audioverarbeitung im Assembler (Dominik)
        ii. Weiterverarbeitung im Emulator (Dominik)
        iii. Interaktion mit dem Betriebssystem (Florian)
        iv. ...

1 Einleitung

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 nur 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

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
- Untersuchung der Fragestellung mit Hilfe eines Beispielprogrammes
- Untersuchung der Interaktion des Beispielprogrammes mit dem Emulator
- Untersuchung der Interaktion von Emulator und Betriebssystem

1.2 Verwendete Software

- **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

2.1 Hardwareumgebung

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

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`
$0x070$	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`

Tabelle 3: Übersicht der Sound-Register - Teil 1

Die in Tabelle 3 und in Tabelle 4 gelisteten Register sind im mGBA als Felder der Enumeration *GBAIORegisters* (Datei: *$/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.

Offset	Kanal	Funktion	Bezeichnung
$0x090$	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

2.1.2 Übersicht der Sound-Master-Register

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 Sound-Systems möglich ist.

F	E	D	C	B	A	9	8	7	6 5 4	3	2 1 0
R4	R3	R2	R1	L4	L3	L2	L1	-	RV	-	LV

Tabelle 5: Register DMG Master Control

Bits	Name	Definition	Beschreibung
0-2	LV		Left volume
4-6	RV		Right volume
8-B	L1-L4	SDMG_LSQR1, SDMG_LSQR2, SDMG_LWAVE, SDMG_LNOISE	Channels 1-4 on left
C-F	R1-R4	SDMG_RSQR1, SDMG_RSQR2, SDMG_RWAVE, SDMG_RNOISE	Channels 1-4 on right

Tabelle 6: Registerinhalt DMG Master Control

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.

F	E	D	C	B	A	9	8	7 6 5 4	3	2	1 0
BF	BT	BL	BR	AF	AT	AL	AR	-	BV	AV	DMGV

Tabelle 7: Register Direct Sound Master Control

Bits	**Name**	**Definition**	**Beschreibung**
0-1	DMGV	SDS_DMG25, SDS_DMG50, SDS_DMG100	DMG Volume ratio 00: 25% 01: 50% 10: 100% 11: forbidden
2	AV	SDS_A50, SDS_A100	DSound A volume ratio. 50% if clear; 100% of set
3	BV	SDS_B50, SDS_B100	DSound B volume ratio. 50% if clear; 100% of set
8-9	AR,AL	SDS_AR, SDS_AL	DSound A enable Enable DS A on right and left speakers
A	AT	SDS_ATMR0, SDS_ATMR1	Dsound A timer. Use timer 0 (if clear) or 1 (if set) for DS A
B	AF	SDS_ARESET	FIFO reset for Dsound A. When using DMA for Direct sound, this will cause DMA to reset the FIFO buffer after it's used.
C-F	BR, BL, BT, BF	SDS_BR, SDS_BL, SDS_BTMR0, SDS_BTMR1, SDS_BRESET	As bits 8-B, but for DSound B

Tabelle 8: Registerinhalt Direct Sound Master Control

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.

F E D C B A 9 8	7	6 5 4	3	2	1	0
-	MSE	-	4A	3A	2A	1A

Tabelle 9: Register Master Sound Output / Status

Bits	**Name**	**Definition**	**Beschreibung**
0-3	1A-4A	SSTAT_SQR1, SSTAT_SQR2, SSTAT_WAVE, SSTAT_NOISE	Active channels. Indicates which DMA channels are currently playing. They do not enable the channels; that's what DMG Master Control 2.1.2 is for.
7	MSE	SSTAT_DISABLE, SSTAT_ENABLE	Master Sound Enable. Must be set if any sound is to be heard at all. Set this before you do anything else: the other registers can't be accessed otherwise, see GBATek for details.

Tabelle 10: Registerinhalt Master Sound Output / Status

2.2 Plattformen

2.2.1 DevkitPro

2.2.2 BeLogic

2.2.3 MaxMod

2.2.4 Snappy

2.3 Softwareentwicklung

2.3.1 API zur Audioverarbeitung

2.3.2 Audiodaten im Speicher

2.3.3 Snappy - „Audio-Reflektor“

2.3.4 Untersuchung des Assemblers

3 Emulation mittels mGBA

3.1 Was ist der mGBA?

3.2 Emulation des Game Boy Advance

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

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 Soundsystem notwendig sind.

3.2.2 Start der Anwendung

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`

3.2.3 Laden und Starten eines ROM

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/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 **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 11: 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.

3.2.4 Ausführung eines ROM

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*)

3.2.5 Interaktion mit dem Soundsystem

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

3.3.1 Audioverarbeitung im Assembler

3.3.2 Weiterverarbeitung im Emulator

3.3.3 Interaktion mit dem Betriebssystem

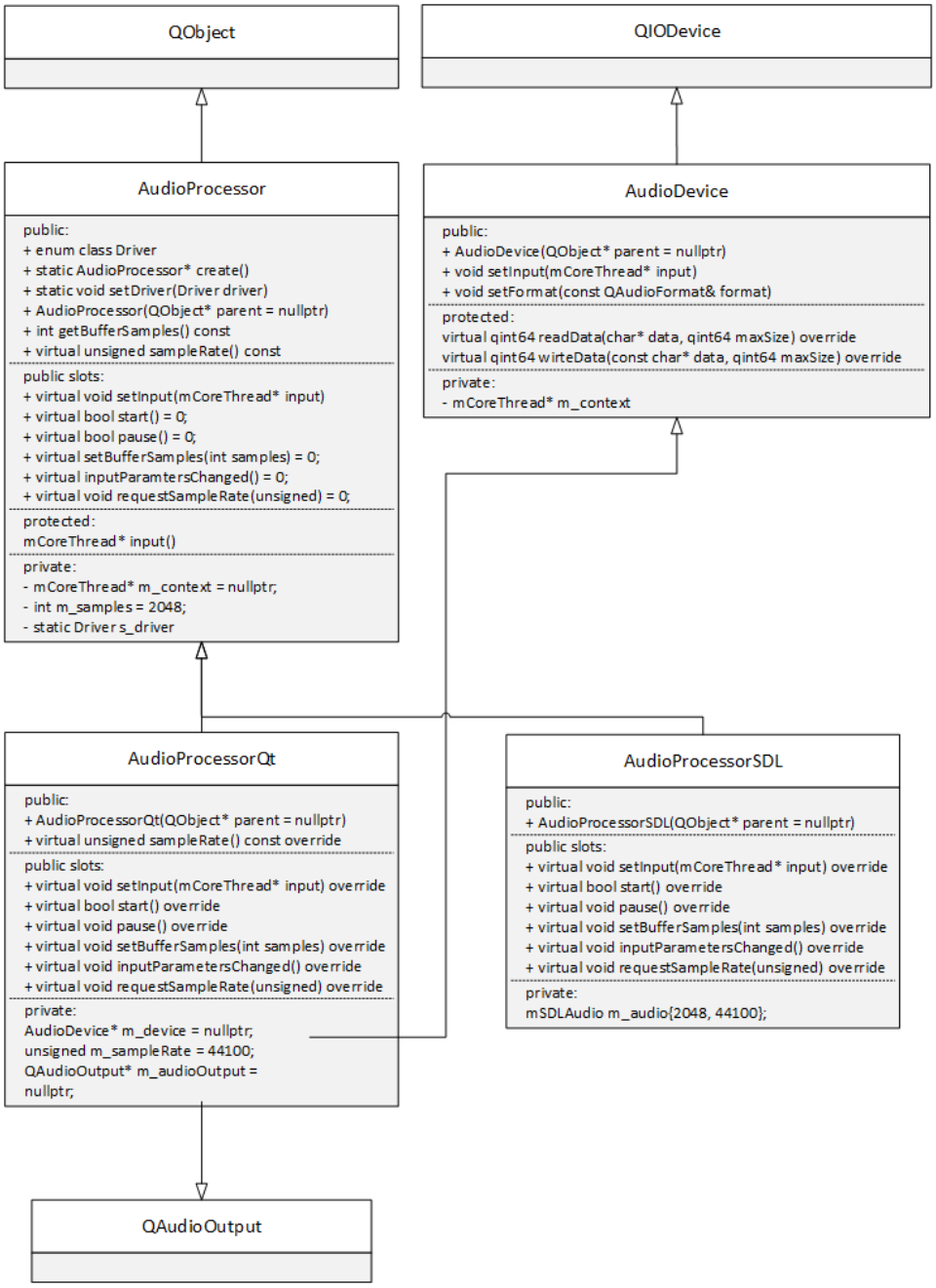

Abbildung 2: Audioklassen in der QT-Anwendung

Wie in Abildung 2 dargestellt, besteht die Audioverarbeitung auf Seitens der Qt-Anwendung aus drei wesentlichen Klassen. Die Klasse **AudioDevice**, **QAudioOutput** und **AudioProcessorQt**. Dabei leitet **AudioDevice** von QIODevice und **AudioProcessorQt** von AudioProcessor ab.

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

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

Der Konstruktor verfügt über keine Logik und ist somit leer. Abhängig von dem im Makefile angegebenen Treiber kann über die Methode **AudiProcessor::Driver** der Treiber in die Member-Variable s_driver gesetzt werden. mGBA nutzt standardmäßig den Treiber QT_MULTIMEDIA. Aus diesem Grund, wird auf eine genauere Betrachtung der Klasse AudioProcessorSDL verzichtet.

Über die Methode **AudiProcessor::create()** wird abhängig vom gesetzten Treiber eine neue Instanz von AudioProcessorQt oder AudioProcessorSDL erzeugt und liefert diese als Rückgabewert zurück.

AudiProcessor::getBufferSamples const() liefert die Anzahl der Samples zurück. Diese ist initial vorgegeben in der Variablen m_samples und beträgt 2048. 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.

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

Auch hier beinhaltet der Konstruktor keine Logik. Aktiv wird der **AudioProcessorQt** durch den Aufruf der „start“ Methode **TODO: ref auf Abschnitt von Dominik**.

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

Zunächst wird überprüft, ob der mCoreThread bereits übergeben und gesetzt wurde. Ist dies nicht der Fall, terminiert die Methode mit dem Rückgabewert **false**. Anschließend wird, falls dies noch nicht geschehen ist, ein **AudioDevice** erzeugt und in die Zeiger-Variable **m_device** geschrieben.

AudioDevice::AudioDevice(QObject* parent) (*$/src/platform/qt/AudioDevice.cpp*)

Die Klasse **AudioDevice** stellt das Bindeglied zwischen AudioProcessorQt und dem mCoreThread dar. Wie schon erwähnt, erbt diese 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.

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

Nach dem erfolgreichen Anlegen eines AudioDevices, wird im nächsten Schritt, falls dies noch nicht vorhanden ist, ein **QAudioFormat** erzeugt. 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** angelegt. **QAudioOutput** stellt ein Interface zur Verfügung mit dem Audio-Daten zu einem Audio-Gerät gesendet werden können(z.B. Lautsprecher, Kopfhörher usw.) Die **setCatecory** Methode setzt den Modus auf „Game“. Diese Methode wird nicht von allen Plattformen unterstützt.

Am Ende der „start“ Methode wird dem Aufruf **m_audioOutput→start(m_device)** das QAudioOutput gestartet. Dieses transferiert nun kontinuierlich Audio-Daten vom **AudioDevice** zum Audioausgang des Systems. Schließlich wird der Status der QAudioOutput Instanz auf Aktiv gesetzt.

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 2: *Übersicht der Audioklassen in der Qt Anwendung*