UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

# MP3 Player für den Nintendo DS

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von
Jan Wischniowski

Betreuer: Dipl.-Inform. Oliver Abert
Institut für Computervisualistik, AG Computergraphik

Koblenz, im Juli 2007

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

| | Ja | Nein |
|---|---|---|
| Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. | ☐ | ☐ |
| Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. | ☐ | ☐ |

.....
(Ort, Datum) (Unterschrift)

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

Die Geschichte der Aufzeichnung von Schall beginnt schon früh. Bereits im Jahre 1867 stellte Charles Cros, ein französischer Dichter und Philosoph, einen *automatischen Telegraphen* zur Aufzeichnung und Wiedergabe von Schall vor. Er konnte das Gerät zwar nicht vermarkten, jedoch war es die Grundlage für weitere Entwicklungen. Zehn Jahre später meldete Thomas Alva Edison ein Patent auf seinen *Phonographen* an, der eine Zinnfolie als Tonträger nutzte. 1881 versuchte Alexander Graham Bell den Phonographen zu verbessern, indem er Wachswalzen als Tonträger einsetzte, wodurch störende Nebengeräusche gemindert wurden. Charles Sumner Tainter gab dem Gerät den Namen *Graphophone* und meldete ein Patent darauf an. Um Patentrechte zu umgehen, wandelte Emil Berliner die Konstruktion leicht ab und stellte 1888 fest, dass es vorteilhafter ist, die Informationen nicht auf Walzen, sondern spiralförmig in eine flache Scheibe zu gravieren. Daraus entwickelte sich die *Schallplatte*.[16]

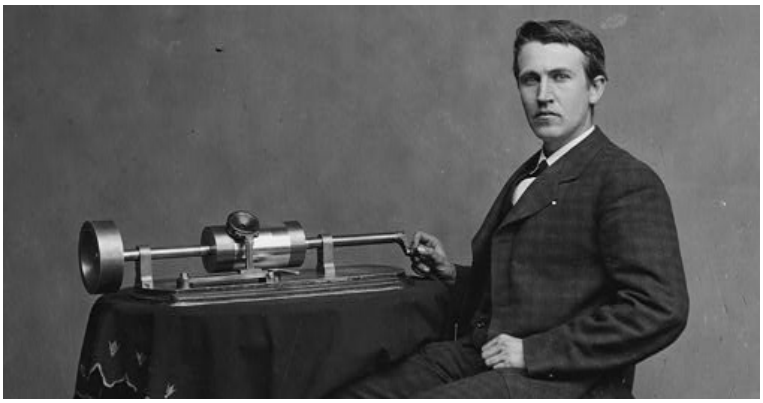

**Abbildung 1:** Thomas Edison mit seinem Zinnfolienphonographen [14]

Erst später, in den 1930er Jahren, kamen zu den mechanischen Verfahren magnetische Aufnahmeverfahren hinzu. Seit diesem Zeitpunkt dienten immer häufiger *Tonbandgeräte* zur Archivierung und Wiedergabe akustischer Informationen. Diese Geräte waren anfangs sehr teuer, unhandlich und schwer zu bedienen, so dass sie nur einem kleinen Personenkreis zur Verfügung standen. Erst im Jahre 1963 stellte die niederländische Firma Philips eine für den Massenmarkt taugliche Alternative auf der Internationalen Funkausstellung in Berlin vor[3]. Dies war die so genannte *Compact*

*Cassette* und das dazugehörige Abspielgerät[9]. Alternative Systeme von Grundig und Sony konnten sich nicht durchsetzen. Im Folgenden fand eine anhaltende Miniaturisierung der Abspielgeräte statt. Bis hin zum tragbaren *Walkman*, den Sony 1979 auf den Markt brachte[6].

Sowohl Schallplatte als auch Bandgeräte brachten jedoch eine ganze Reihe von Problemen mit sich. So nahm die Klangqualität bei der Nutzung durch Verschleiß oder Entmagnetisierung kontinuierlich ab. Schallplatten waren unhandlich und bei Kassetten war es schwierig zu einem bestimmten Abschnitt zu springen, weil nur sequentiell vor- oder zurück gespult werden kann. Abhilfe versprach die *Compact Disc* (CD), ein optischer Massenspeicher zur digitalen Speicherung von Daten, die 1982 von Philips und Sony auf den Markt gebracht wurde. Dadurch, dass die Daten nicht mehr mechanisch, sondern mit Hilfe eines Lasers ausgelesen wurden, gab es keine Abnutzung mehr. Durch die digitale Speicherung entfiel der Qualitätsverlust beim Kopieren der Daten und die Handhabung wurde sehr vereinfacht.
Allerdings reagieren die Wiedergabegeräte empfindlich auf ruckartige Bewegungen, da der Laser bei der Abtastung dann recht schnell die Spur verliert und neu ausgerichtet werden muss. Bei tragbaren CD Player sind deswegen aufwändige Korrekturmaßnahmen erforderlich.

Nachdem der Schritt von der Analogtechnik zur Digitaltechnik vollzogen war, ging es darum, den Speicherplatzbedarf der Audiodaten durch Komprimierung zu reduzieren. Dabei soll eine möglichst gute Klangqualität bei niedrigem Datenaufkommen erreicht werden. Ein weit verbreitetes Format zur Kompression ist das vom Fraunhofer Institut für integrierte Schaltungen entwickelte MPEG-1 Audio Layer III Format (MP3). Da die komprimierte Musik weniger Speicherplatz benötigt, können bei gleicher Datenträgergröße mehr Lieder gespeichert werden. Natürlich muss das Abspielgerät das Dekodieren einer solchen Musikdatei unterstützen. Aus diesem Grund werden vor allem tragbare CD Player und auch die CD Player in Fahrzeugen mit einem MP3 Decoder ausgestattet.

Der logische nächste Schritt war der Verzicht auf die immer noch verhältnismäßig unhandlichen CDs. Für die Speicherung von MP3 Dateien bieten sich eine Menge Datenträger an. Vor allem wiederbeschreibbare Flash-Speicher, die es in den verschiedensten Formen und Varianten gibt, aber auch kleine Festplatten. So entwickelten sich Abspielgeräte mit einem internen Speicher und einem Decoder für MP3s und mitunter auch andere Formate.
Die üblichen *MP3 Player* mit internem Flash-Speicher oder Festplatten sind sehr klein und handlich, können viele Stunden Musik speichern und bei Bedarf neu bespielt werden. Allerdings sind sie gewöhnlich umständlich

zu bedienen, da man nur einige wenige Knöpfe zur Steuerung und ein kleines Display zur Anzeige von Informationen zur Verfügung hat. An diesem Punkt setzt die Studienarbeit an.

## 1.2 Ziel der Studienarbeit

Das Ziel dieser Studienarbeit ist es, einen MP3 Player zu entwickeln, der eine Benutzerinteraktion ermöglicht wie es gängige Computerprogramme zur Wiedergabe von Musik tun. Das heißt, der Benutzer soll über eine grafische Oberfläche MP3 Dateien laden, abspielen und diese auch in Playlisten organisieren können. Darüber hinaus soll es möglich sein, Informationen, wie Titel, Autor, Genre, Veröffentlichungsjahr und vieles mehr, die in einer MP3 Datei als zusätzlicher *Tag* gespeichert werden können, zu editieren. Diese Informationen soll die Software auch beim Abspielen eines Musikstückes auslesen und dem Nutzer übersichtlich anzeigen. Hier scheitern die meisten MP3 Player aufgrund ihres kleinen Displays. Außerdem soll der MP3 Player auch rudimentäre Funktionen zur Echtzeitmanipulation der Musikwiedergabe bieten. Im konkreten Fall soll ein Equalizer, also ein Tongestalter, ermöglichen, Frequenzbereiche zu verstärken oder zu dämpfen. Dadurch können zum Beispiel die Höhen, Mitten und Tiefen nach Belieben verändert werden um den Klang anzupassen.
Als Hardware zum Abspielen der Musikdateien dient die Spielekonsole Nintendo DS, welche aufgrund ihrer beiden Displays genügend Anzeigemöglichkeiten für eine grafische Benutzerführung bietet. Darüber hinaus dient eines der beiden Displays als Touchscreen und kann für Eingaben verwendet werden.

## 1.3 Dokumentationen und Werkzeuge

Nintendo bietet für offizielle Entwickler Dokumentationen, Compiler, Debugger und Entwicklungshardware in Form des *NDS DevKit NITRO* an. Dieses steht für die Studienarbeit nicht zur Verfügung. Stattdessen können nur die Informationen und Werkzeuge verwendet werden, die im Laufe der Zeit in der so genannten "*Homebrew Szene*" zusammengetragen und entwickelt wurden.[1]

[1] Homebrew (engl. für selbstgebrautes) bezeichnet Software für Konsolen, die nicht von Entwicklern programmiert wurde, die durch den Hersteller der Hardware lizensiert sind.

# 2 Grundlagen

In diesem Kapitel werden zunächst die Grundlagen erläutert, die für das Verständnis wichtig sind. Zunächst wird die Hardware des Nintendo DS genauer betrachtet. Anschließend wird auf die Eigenschaften von Schall und Akustik eingegangen und erklärt, wie Geräusche digitalisiert und gespeichert werden können. Darauf aufbauend wird erklärt, wie die Komprimierung solcher Daten vonstatten geht.

## 2.1 Nintendo DS

Beim Nintendo DS handelt es sich um eine Spielekonsole. Hierbei steht die Bezeichnung DS für "Developers System" beziehungsweise "Dual Screen". Sie sollte vor dem Marktstart geändert werden. Da die Konsole jedoch schon vor der Veröffentlichung für Diskussionen sorgte und die Bezeichung für viele ein Begriff war, wurde sie beibehalten.
In Europa kam die Konsole im ersten Quartal 2005 auf den Markt.

**Abbildung 2:** Der Nintendo DS Lite

Der Nintendo DS besitzt zwei Prozessoren, einen mit 67 MHz getakteten ARM946E-S Prozessor als Hauptprozessor und als Subprozessor einen etwas langsameren ARM7TDMI, mit einer Taktfrequenz von 33 MHz.[1] Er verfügt über 4 MB Arbeitsspeicher und 656 KB Grafikspeicher.
Die beiden Displays messen jeweils 61 x 46 mm und haben eine Auflösung

[1] http://arm.com/products/CPUs/index.html

von 256 x 192 Bildpunkten. Ein Mikrofon erlaubt die Aufnahme von Geräuschen, über die internen Lautsprecher oder einen Kopfhöreranschluss können Sounddaten wiedergegeben werden. Der Nintendo DS ist durch ein proprietäres Netzwerkprotokoll, das auf dem IEEE 802.11b Standard[1] basiert, auch zur drahtlosen Netzwerkkommunikation fähig.[7]
Die Befehlseingabe kann direkt über ein Steuerkreuz und die Tasten A, B, X, Y, L, R, Select und Start erfolgen. Das Auf- und Zuklappen des Deckels wird ebenfalls von einer Taste registriert. Darüber hinaus bietet der Nintendo DS die Möglichkeit, Eingaben über den Touchscreen vorzunehmen.

Eine Übersicht über den internen Speicher der Konsole bietet Abbildung 3. Für die Programmierung ist das Verständnis über die Aufteilung der Speicherbereiche und der Speicherzugriffe unabdingbar.

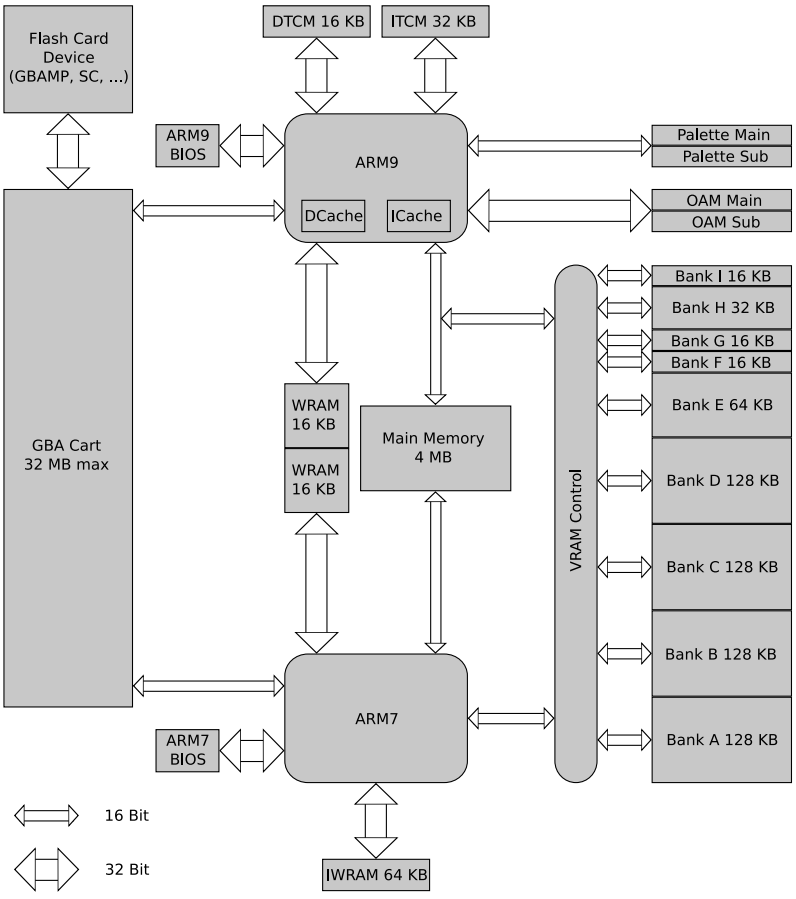


**Abbildung 3:** MemoryMap des Nintendo DS [4]

[1] http://standards.ieee.org/getieee802/802.11.html

Für das Laden von Programmcode stehen 4 MB Main Memory zur Verfügung, auf die sowohl der Hauptprozessor, als auch der Subprozessor zugreifen kann. Auf den lediglich 656 KB großen Videospeicher, der in neun Bereiche (Bank A bis Bank I) aufgeteilt ist, können die Prozessoren nicht direkt zugreifen, sondern müssen den Weg über einen VRAM Controller gehen. Die Bereiche selbst sind zwischen 16 KB und 128 KB groß und müssen, bevor sie genutzt werden können, in den Memoryspace der 2D Grafikengine gemappt werden.

Listing 1 zeigt, wie die Video RAM Bank A (VRAM_A) an die Adresse `0x6000000` gemappt wird, um den Speicher als 2D Background Memory zu nutzen. Schreibt man jetzt Daten in den Speicher an diese Adresse, werden sie in die Video RAM Bank A geschrieben.

**Listing 1:** VRAM Bank A an die Adresse 0x6000000 mappen

```
1 videoSetBankA(VRAM_A_MAIN_BG_0x6000000);
```

## 2.2 Vom analogen Audiosignal zur digitalen Datei

### 2.2.1 Analoges Audiosignal

Geräusche werden in Form mechanischer Schwingungen, den Schallwellen, über ein Transportmedium übertragen. Das menschliche Ohr empfängt die Schallwellen und wandelt sie in einen Sinneseindruck um. Dabei werden Tonhöhe, Lautstärke und Richtung ermittelt.

Hohe Frequenzen, also schnelle Schwingungen, werden als hohe Töne wahrgenommen, tiefe Frequenzen hingegen als tiefe Töne. Angegeben wird die Anzahl der vollständigen Schwingungen pro Sekunde in Hertz (Hz). Menschen können Frequenzen zwischen 20 Hz und 20 KHz hören.

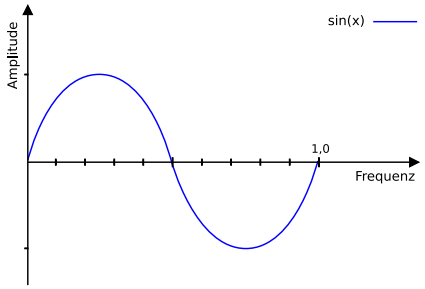


**Abbildung 4:** Sinusförmiges Audiosignal

Eine reine sinusförmige Schwingung ist ein Ton. Töne können künstlich erzeugt werden und kommen in der Natur nicht vor [11].

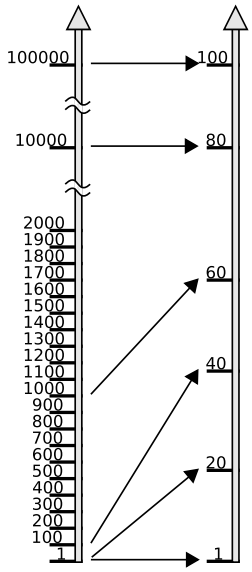


**Abbildung 5:** Umrechnung einer linearen in eine logarithmische Skala [11]

Mit steigender Amplitude nimmt die Lautstärke zu. Die Amplitude wird in Dezibel (dB) angegeben. Dabei handelt es sich um eine logarithmische Skala, da Menschen Lautstärken vom leisen Blätterrauschen bis zum Start eines Flugzeuges wahrnehmen können. Auf einer linearen Amplitudenskala müsste der Lautstärkebereich somit über viele Zehnerpotenzen aufgetragen werden, wodurch das Ablesen kleiner Änderungen kaum möglich wäre.

Die Richtung ermittelt das Gehirn durch die zeitliche Verzögerung, mit der die Schallwelle am linken und rechten Ohr ankommt.

Werden zu einem Grundton mit der Frequenz f weitere sinusförmige Schwingungen hinzugefügt, deren Frequenzen ganzzahlige Vielfache von f sind, entsteht ein harmonischer Klang. Überlagern sich nichtperiodische Schwingungen, entstehen Geräusche.

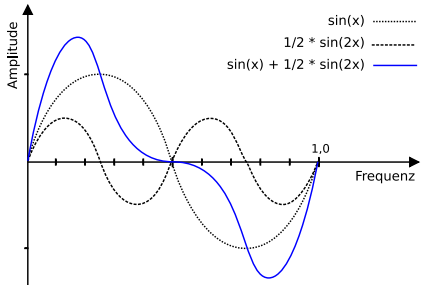


**Abbildung 6:** Überlagern sich mehrere Frequenzen, so addieren sich die Amplituden und es entsteht eine neue Schwingung

### 2.2.2 Digitales Audiosignal

Klänge sind analoge Signale, die beliebig stark differenziert werden können. Da auf Datenträgern jedoch nur begrenzter Speicherplatz zur Verfügung steht, wird das Signal in regelmäßigen Abständen abgetastet (*Sample*) und die Abtastwerte zwischengespeichert (*Hold*). Aus dem ursprünglichen stetigen Signal wird dadurch eine Treppenfunktion. Dabei besagt das Nyquist-Shannonsche Abtasttheorem, dass die Abtastfrequenz doppelt so hoch sein muss, wie die maximale im analogen Signal vorkommende Frequenz. Im Fall der Musikkodierung genügt es, mit der doppelten Frequenz abzutasten, die Menschen hören können. Niedrigere Abtastraten führen zu Aliasing Fehlern. Der Sound klingt weniger klar und wirkt dumpf. Da Menschen Schwingungen bis etwa 20 KHz hören, wird für eine Audio CD mit 44,1 KHz abgetastet, was etwas mehr als das Doppelte ist.[11]
Bei jeder Abtastung wird der Wert der Amplitude gespeichert. Die Genauigkeit, mit der dieser Wert gespeichert werden kann, hängt davon ab, wie viele Bits jeweils zur Verfügung stehen. Wird das Signal mit einer Auflösung von 8 bit gespeichert, stehen $2^8$, also 256 Stufen zur Verfügung. Bei 24 bit können $2^{24}$ unterschiedliche Stufen gespeichert werden. Damit sind 16,7 Millionen verschiedene Werte möglich.

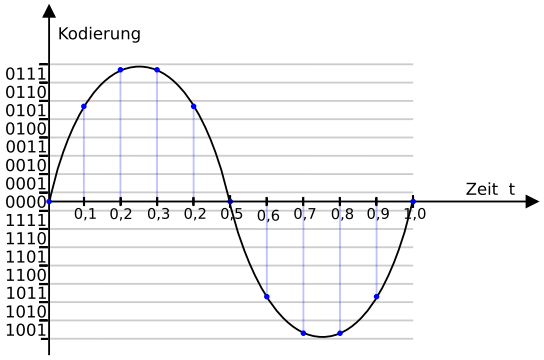


**Abbildung 7:** Sample and Hold. Die Abtastung erfolgt 10 mal je Sekunde. Dies entspricht einer Abtastrate von 10 Hz. Kodiert wird mit 4 bit, was 16 Stufen ermöglicht.

Bei der Stufenbildung entsteht immer ein Quantisierungsfehler, der größer wird, je weniger Stufen zur Verfügung stehen. Dies macht sich bei der Wiedergabe in einem hörbaren Rauschen, dem Quantisierungsrauschen, bemerkbar. Je mehr Bits zur Verfügung stehen, um so besser kann das Ursprungssignal approximiert werden.

## 2.3 Das MP3 Format

### 2.3.1 Kompression

Die Datenmenge in Bit eines unkomprimierten Sounds kann folgendermaßen berechnet werden:

$$D = A * f_A * Z * t$$

Dabei ist $D$ die Datenmenge, $A$ die Auflösung in Bit, $f_A$ die Abtastfrequenz in Hz, $Z$ die Kanalanzahl und $t$ die Aufnahmezeit in Sekunden.
Eine Minute Audio Rohmaterial in Stereo CD Qualität (16 bit, 44.100 Hz, Stereo) benötigt etwa 10 MB Speicherplatz:

$$D = 16 * 44.100 * 2 * 60 = 10.584.000\ Byte = 10,1\ MB$$

Da solch große Dateien für viele Anwendungsbereiche, etwa der Übertragung übers Internet oder bei der Wiedergabe unterwegs, zu viel Speicherplatz benötigen, werden sie gewöhnlich komprimiert gespeichert. In diesem Fall ist das MP3 Format von Interesse, da der im Rahmen der Arbeit programmierte Player dieses unterstützt. MP3 steht für MPEG-1 Audio Layer III. MPEG ist ein von der Moving Picture Experts Group[1] definierter Überbegriff einer Familie, um die Entstehung verschiedener zueinander inkompatibler Formate zu verhindern. MPEG wird in die Hauptklassen MPEG-1, MPEG-2, MPEG-4 und MPEG-7 unterteilt, welche wiederum in Subklassen, so genannte Layer, aufgeteilt sind. Die Layer nehmen jeweils an Komplexität zu. So ist ein Layer III Codec wesentlich komplexer als ein Layer I Codec, hat dafür jedoch die beste Wiedergabequalität pro Datenrate. Da die Dekodierer abwärtskompatibel sind, kann ein Layer III Decoder auch Dateien der Layer I und II dekodieren.[8]

Um den Speicherplatzbedarf zu reduzieren wird das Signal in mehreren Stufen komprimiert. Zunächst transformiert eine Filterbank das Audiosignal vom Zeit- in den Frequenzbereich. Dabei unterteilt es das Signal in 32 gleichbreite Frequenzbänder. Da bei der Anwendung eines psychoakustischen Modells im nächsten Schritt eine feinere Spektralauflösung nötig ist, werden die Frequenzbereiche durch eine modifizierte diskrete Kosinus-Transformation (MDCT) jeweils in weitere 18 Bereiche unterteilt. Abbildung 8 verdeutlicht den Ablauf.

[1] http://www.chiariglione.org/mpeg/

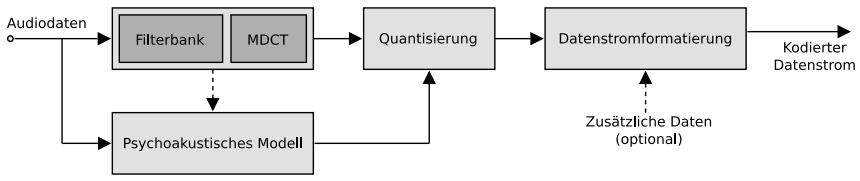


**Abbildung 8:** Aufbau eines MP3 Kodierers [17]

Aufgrund verschiedener Wahrnehmungseffekte der Psychoakustik und darauf beruhende empirische Ermittlungen, wurde ein Modell entwickelt, anhand dessen bestimmte Informationen aus dem Signal entfernt werden können, ohne dass der Verlust hörbar ist.[17]

- Töne müssen eine Hörschwelle überschreiten, also eine Mindestlautstärke haben, um gehört zu werden.
- Menschen nehmen Töne nur zwischen 20 Hz und 20 KHz war.
- Folgt ein leiser Ton einem lauten Ton oder geht ihm kurz voraus, wird der leise Ton nicht wahrgenommen.
- Töne lassen sich erst ab einem bestimmten Frequenzunterschied differenzieren.
- Simultane Maskierung. Ein lauter Ton, der Maskierungston, überdeckt Töne ähnlicher Frequenz, so dass diese nur noch ab einer bestimmten Lautstärke zu hören sind. Die Lautstärke, die die maskierten Töne mindestens haben müssen um noch gehört zu werden, hängt von der Intensität des Maskierungstons ab. Mit zunehmender Frequenz des Maskierungstons wird der maskierte Bereich breiter. Je höher die Frequenz ist, desto mehr umliegende Frequenzen werden überdeckt.
- Zeitliche Maskierung. Zwei ähnliche Töne können nur wahrgenommen werden, wenn sie einen bestimmten zeitlichen Abstand voneinander haben.

Der Encoder generiert auf Grundlage dieses psychoakustischen Modells ein Signal, das weniger Informationen enthält als das Ursprungssignal, indem nicht Hörbares weggelassen wird. Dazu wird jedes der 32 Subbänder untersucht und gemäß dem psychoakustischen Modell auf Frequenzen verzichtet, die nicht gehört werden können. Nun wird jedes Subband abgetastet und der Wert als Sample mit einer Größe von 16 bit gespeichert. Da

bei diesem Schritt Informationen verloren gehen, handelt es sich um eine verlustbehaftete Kodierung.[5]

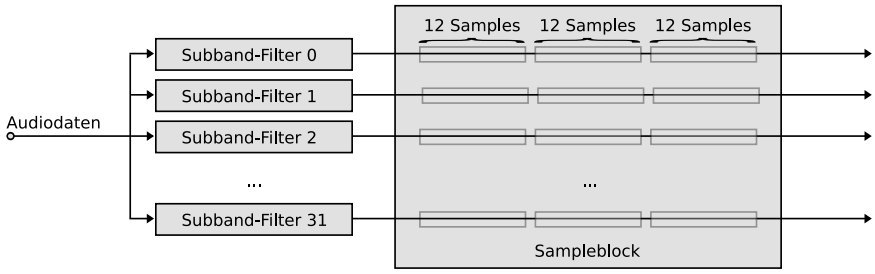


**Abbildung 9:** Aufbau eines MP3 Sampleblocks [17]

Darauf aufbauend erfolgt die Quantisierung, die am meisten Rechenzeit in Anspruch nimmt. Die Samples werden hierfür zu Blöcken, den *Frames*, mit je 1152 Samples zusammengefasst (12 Samples x 32 Subbänder x 3) und komprimiert. Nicht für jedes Sample sind 16 bit nötig, um den Pegel darzustellen. Führende Nullen können zum Beispiel weggelassen werden. Für die Rekonstruktion müssen dann zwar zusätzliche Informationen gespeichert werden, dies geschieht jedoch so geschickt, dass weniger Speicherplatz nötig ist. Für ein Signal, das ursprünglich mit 16 bit, 44100 Hz, Stereo aufgenommen wurde, ergibt sich ein Datenstrom von

$$16bit * 44100Hz * 2 = 1411200bit/s = 1378Kb/s$$

Soll das Signal nun durch Kompression auf 128 Kb/s reduziert werden, ergibt sich für jeden Datenblock eine Größe von

$$1152Samples * 128000bit/44100Hz = 3344bitjeBlock$$

Werden in einem Block nicht alle verfügbaren Bits benötigt, dienen die übrig gebliebenen Bits in diesem Frame als Reservoir. Wenn sich ein Block nicht ohne hörbaren Qualitätsverlust auf die vorgegebene Größe reduzieren lässt, können die zusätzlichen Daten in verfügbare Reservoirs ausgelagert werden. Dadurch sind die Frames teilweise voneinander abhängig.

Zuletzt werden die Daten Huffman-entropiekodiert. Die Huffman-Kodierung wird in unterschiedlichen Bereichen angewendet und ist verlustlos. Die Idee, welche diesem Verfahren zugrunde liegt ist es, Zeichen, die häufig vorkommen, mit möglichst wenigen Bits zu kodieren. Dazu wird festgestellt, welche Symbole vorkommen und deren Häufigkeit bestimmt. Jedes

Symbol mit seiner Häufigkeit ist ein Blatt eines Binärbaums, der darauf aufbauend erstellt wird. Nun werden die Blätter durchsucht und zwei Blätter ermittelt, welche die kleinsten Häufigkeiten haben. Diese werden verbunden, was zu einem neuen Knoten mit der Summe der Wahrscheinlichkeiten der zusammengefassten Blätter führt. Der Vorgang wird so lange wiederholt, bis zu jedem Blatt ein Pfad von der Wurzel hinführt. Für jedes Symbol kann nun von der Wurzel ausgehend zum Blatt mit diesem Symbol die Kantenmarkierung abgelesen werden. Symbole mit großer Häufigkeit haben den kürzesten Pfad. [13]

Die Zeichenfolge "DBACCDBDBC" enthält

1 x A (10 %)
2 x B (20 %)
3 x C (30 %)
4 x D (40 %)

Der Binärbaum dazu sieht folgendermaßen aus:

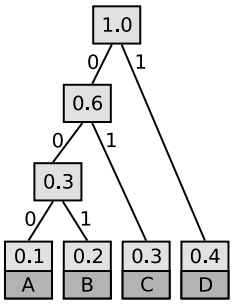


**Abbildung 10:** Binärbaum, wie er bei der Huffman-Kodierung erzeugt wird

Damit entsprechen die Symbole A, B, C und D den Codewörtern:

| Symbol | Codewort |
|---|---|
| A | 000 |
| B | 001 |
| C | 01 |
| D | 1 |

Das Symbol A, mit der geringsten Häufigkeit, hat also das längste Codewort, das Symbol D, das am häufigsten vorkommt, ist mit dem geringsten Wort kodiert.

Bei der Kodierung von MP3 Dateien spielt die Huffman-Kodierung bei polyphonen Passagen eine große Rolle, also solchen Abschnitten, in denen viele Töne gleichzeitig auftreten.

Gewöhnlich werden für Audiodaten zwei Kanäle gespeichert, einen für den linken und einen für den rechten Lautsprecher. Erfahrungsgemäß unterscheiden sich die Kanäle nur unwesentlich, es liegt also eine hohe Korrelation der Daten vor. Dies machen sich zwei Verfahren zu Nutze:
Das so genannte *Intensity Stereo Verfahren*, welches lediglich einen Monokanal und Richtungsinformationen für die beiden Kanäle speichert und die Stereodaten daraus rekonstruiert. Dabei gehen allerdings Phaseninformationen verloren, weshalb es sich um ein verlustbehaftetes Verfahren handelt, das kaum noch Anwendung findet.
Gängiger ist das verlustfreie *Mid/Side Stereo Verfahren*. Dabei wird der Mittelwert beider Kanäle in einem Kanal gespeichert (*mid channel*) und die Differenzen in einem anderen Kanal (*side channel*). Die Differenzwerte sind aufgrund der gewöhnlich hohen Korrelation sehr klein und können effizient gespeichert werden.

Während der Algorithmus zum Dekodieren festgelegt ist, kann die Funktionsweise des Kodierers weitgehend beliebig implementiert werden. Lediglich Richtlinien für eine gewisse einheitliche Struktur werden vorgegeben, weshalb sich die Ergebnisse unterscheiden können, wenn die gleiche Audiodatei mit verschiedenen Programmen zu einer MP3 Datei umgewandelt wird. Dadurch kann das Kodierverfahren nachträglich verbessert werden, während die bestehenden Encoder weiterhin verwendet werden können.

### 2.3.2 Aufbau

In den 80er Jahren des 20. Jahrhunderts hatten Forscher das Ziel ein Format zu entwickeln, das es ermöglicht, Sprache und Musik in guter Qualität über Telefonleitungen und anderen Medien zu übertragen, bei denen nur wenige Ressourcen zur Verfügung stehen und deshalb mit der Datenmenge sparsam umgegangen werden muss. Bei diesen Entwicklungen wurden die Grundlagen für das MP3 Format gelegt. Später kam auch hinzu, dass die Daten *streambar* sein sollen. *Streaming* im Internet ist vergleichbar mit *Broadcasting* beim Hörfunk. Die Daten werden kontinuierlich sequentiell gesendet und können von anderen Stationen empfangen und wiedergegeben werden. Da Empfangsgeräte aber zu jedem beliebigen Zeitpunkt eingeschaltet werden können, auch wenn die Übertragung bereits läuft, muss es ihnen möglich sein, zu jedem Zeitpunkt Informationen (Bitrate, Channel Mode, ...) zu den gestreamten Daten zu bekommen, sonst können sie die

Daten nicht korrekt interpretieren. Aus diesem Grund besteht eine MP3 Datei aus *Frames*, denen jeweils ein *Header* mit diesen Informationen vorangestellt ist. Ein Empfangsgerät muss jetzt lediglich die kurze Zeit abwarten, bis das letzte Frame zu Ende gesendet wurde und bekommt schon mit dem nächsten Frame alle Informationen, die es zum Dekodieren benötigt. Dadurch werden auch variable Bitraten möglich, denn die Bitrate kann für jeden Frame neu angegeben werden. Ruhige Passagen in einem Lied können dann mit einer niedrigen Bitrate kodiert werden, während bei aufwändigen Abschnitten mehr Bits zur Verfügung gestellt werden, um auch feine Unterschiede in den Klängen noch speichern zu können.


**Abbildung 11:** Aufbau einer MP3 Datei

Ein Frame besteht aus einem Header- und einem Datenteil und enthält immer 26 Millisekunden Audiodaten, was etwa 38 Frames pro Sekunde entspricht. Wie bei einem Film dienen Frames dazu, eine konstante Abspielrate auf unterschiedlichen Abspielgeräten zu ermöglichen. Bei höherer Bitrate benötigt ein Frame mehr Speicherplatz als bei geringerer Bitrate. Die Anzahl der in einem Frame gespeicherten Abtastwerte ist bei Layer III der MPEG Kompression konstant mit 1152 Samples pro Frame (32 Subbänder x 36 Samples). Bei komplexen Musikstücken kann es vorkommen, dass der verfügbare Speicherplatz in einem Frame nicht ausreicht. Für diesen Fall ist beim MP3 Format ein Reservoir vorgesehen, welches als Puffer dient.

### Header

Bei MPEG Audio Dateien ist jedem Frame ein Header vorangestellt. Dieser ist 4 Byte, also 32 bit groß. Ein Header enthält Informationen, die der Decoder benötigt, um den entsprechenden Frame zu dekodieren.

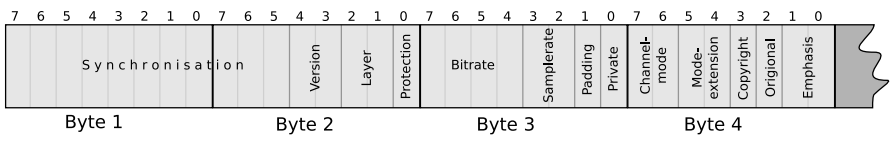


**Abbildung 12:** Aufbau eines MP3 Headers

*Synchronisation*
Die ersten 11 bit sind immer gesetzt und ermöglichen dem Decoder, den Header zu finden.

*Version*
Gibt die Audio Version ID an. Zur Auswahl stehen MPEG Version 1, MPEG Version 2 und MPEG Version 2.5. Relevant sind in diesem Kontext MPEG Version 1 Dateien.

*Layer*
MPEG Dateien der verschiedenen Versionen sind noch einmal unterteilt in drei Varianten unterschiedlicher Komplexität. Layer I, Layer II und Layer III, wobei Layer I am einfachsten strukturiert ist und Layer III am komplexesten. Von Bedeutung sind hier Dateien mit dem Layer III.

*Protection*
Gibt an, ob eine Prüfsumme für die Erkennung fehlerhafter Frames vorhanden ist.

*Bitrate*
Je mehr Bits zur Verfügung stehen, desto exakter kann das ursprüngliche Signal gespeichert werden. Dies führt gewöhnlich zu einer höheren Qualität, aber auch zu einem größeren Speicherplatzverbrauch. Bei einer MPEG-1 Layer II Datei stehen verschiedene Bitraten zwischen 32 Kb und 320 Kb zur Verfügung, die von Frame zu Frame variieren können.

*Samplerate*
Spezifiziert die Anzahl der Abtastungen des Ausgangssignals pro Sekunde. Bei einer MPEG-1 Datei sind 32000 Hz, 44100 HZ und 48000 Hz möglich.

*Padding*
Die Anzahl der Bits, die pro Sekunde zur Verfügung stehen, ergibt sich aus der Bitrate. Es ist jedoch nicht immer möglich alle Bits zu verwenden, da die Menge der benötigten Bits vom Audiosignal abhängt. Um die Differenzen auszugleichen, werden einzelne Frames jeweils um ein Bit verlängert. Damit der Decoder weiß, ob das letzte Bit im Frame zum Signal gehört oder lediglich ein Füllbit ist, wird zusätzlich ein Bit im Header gesetzt, das Aufschluss darüber gibt, ob dem Frame ein Füllbit angehängt ist.

*Private*
Das Private Bit steht für benutzerdefinierte Zwecke zur Verfügung und hat keine fest definierte Bedeutung.

*Channelmode*
Gibt an, ob die Audiodaten *Stereo*, *Joint Stereo*, *Dual Channel* (Zwei Mono Channels) oder *Single Channel* (ein Mono Channel) kodiert sind.

*Modeextension*
Angaben zur Optimierung von Stereo-Informationen, die beim *Joint-Stereo-Verfahren* Verwendung finden.

*Copyright*
In diesem Bit kann gespeichert werden, ob die Datei urheberrechtlich geschützt ist.

*Original*
Enthält Informationen darüber, ob es sich um ein Original oder eine Kopie handelt.

*Emphasis*
Wurden die Frequenzgänge angepasst, um die Qualität subjektiv zu verbessern, kann dies hier angegeben werden. Der Decoder hat dann die Möglichkeit entsprechend zu reagieren.

**ID3 Tag**

Bei der Entwicklung des MP3 Formats ist es nicht vorgesehen zusätzliche Informationen, wie etwa Interpret und Titel, in der Datei zu speichern. Diese Informationen können lediglich im Dateinamen abgelegt werden, was natürlich erhebliche Nachteile, wie lange Dateinamen, mit sich bringt. Außerdem sind Einschränkungen seitens der Dateisysteme zu beachten, die viele Sonderzeichen oder zu lange Namen nicht erlauben. Aus diesem Grund wurde ein Teil der Datei für solche Metadaten reserviert. Man spricht hier vom *ID3 Tag*, was für Identify an MP3 steht. Dabei handelt es sich um einen informellen Standard, der bislang nicht zum MP3 Standard gehört. Im wesentlichen gibt es die zwei Versionen ID3v1 und ID3v2.
ID3v1 wurde 1996 eingeführt. Da zur damaligen Zeit noch kein Player mit den Informationen umgehen konnte, wurden sie ans Ende der Datei hinter die Datenende-Markierung angehängt. Der Aufbau der 128 Byte großen Sektion ist einfach gehalten. Zu beachten ist, dass das Genre lediglich als Zahl gespeichert wird. Anhand einer Liste, kann der Wert interpretiert werden.

| Offset | Länge | Bedeutung |
|---|---|---|
| 0 | 3 | Kennung ”TAG” zur Kennzeichnung eines ID3v1-Blocks |
| 3 | 30 | Songtitel |
| 33 | 30 | Künstler/Interpret |
| 63 | 30 | Album |
| 93 | 4 | Erscheinungsjahr |
| 97 | 30 | beliebiger Kommentar |
| 127 | 1 | Genre |

Eine der großen Schwächen des ID3v1 Tags waren die Beschränkungen, wie etwa die Länge der Datenfelder auf weniger als 30 Zeichen. Aus diesem Grund wurde 1998 ein neues Format entworfen, der ID3v2 Tag, der jetzt nicht zwangsläufig am Dateiende gehängt werden muss, sondern auch am Anfang stehen kann. Er ist so kodiert, dass ihn Player, die ihn nicht verstehen, überspringen können.
Da Tags der 2. Generation wesentlich aufwändiger zu identifizieren und interpretieren sind als die ursprünglichen Tags, werden sie vom Programm, das im Rahmen dieser Arbeit entwickelt wurde, nicht beachtet.

# 3 Implementierung

Das Programm wurde unter Linux größtenteils in C++, teilweise in C implementiert und enthält abschnittsweise auch Assemblercode. Zum Kompilieren wurde die Toolchain *devkitARM* (Release 19b) verwendet, die Teil des *DevKit Pro* ist und unter anderem den GCC Compiler für ARM Prozessoren zur Verfügung stellt. Im Folgenden wird eine Übersicht gegeben, welche Bibliotheken und Programme nötig sind.

## 3.1 Verwendete Bibliotheken

Das SDK für den Nintendo DS, das *NDS DevKit NITRO*, ist lediglich für offizielle Entwickler von Software für die Konsole verfügbar und kann im Rahmen der Studienarbeit somit nicht verwendet werden. Um dennoch Anwendungen für die Konsole erstellen zu können, wurde ein inoffizielles SDK, das *DevKit Pro* entwickelt, welches verschiedene Bibliotheken zur Verfügung stellt, die grundlegende Funktionen und Aufgaben übernehmen. Das Devkit Pro basiert auf der *GNU Compiler Collection* und enthält weitere Werkzeuge und Bibliotheken, welche die Konsolenprogrammierung unterstützen.[1]

### 3.1.1 Software Development Kit für ARM Prozessoren

Das *devkitARM*, ein Software Development Kit für ARM Prozessoren, ist Teil des DevKit Pro und enthält unter anderem die Compiler um ausführbaren Code für ARM Prozessoren zu erstellen.

### 3.1.2 Standardbibliothek

Eine wichtige Bibliothek, die ebenfalls zum DevKit Pro gehört, ist die von Michael Noland und Jason Rogers entwickelte und von Dave Murphy gewartete *libnds*. Sie unterstützt nahezu alle Funktionen des Nintendo DS, einschließlich der 2D und 3D Hardware, des Touchscreens, des Mikrofons, und vieles mehr.

[1] http://sourceforge.net/project/showfiles.php?group_id=114505

### 3.1.3 FAT Dateisystem

Mittels spezieller Adapter-Karten kann der Nintendo DS auf diverse Speichermedien, wie CF- oder SD-Karten, zugreifen. Als Dateisystem auf der Speicherkarte bietet sich das FAT System an, da für Zugriffe auf dieses Dateisystem bereits eine Bibliothek von Michael Chisholm entwickelt wurde, die *fatlib*. Der Entwickler ersetzte die Bibliothek aber nach einiger Zeit durch die *libfat*, welche einige Erweiterungen, wie etwa das Sortieren der Datei- und Verzeichnisausgabe bietet und mittlerweile auch Teil des Devkit Pro ist. Eine der wichtigsten Neuerungen jedoch war, dass die Treiber für die Hardwarezugriffe auf die Speicherkarte nicht mehr im Quelltext verankert sind, sondern dynamisch gelinkt werden (DLDI, Dynamically Linked Device Interface). Dadurch können unabhängige Entwickler Treiber für neue Adapter und Speicherkarten programmieren, die dann nachträglich ins Programm gelinkt werden können, ohne dass der Quelltext neu kompiliert werden muss. Dies erlaubt es, das Programm im Nachhinein so zu erweitert, dass es mit weiteren Karten und Adaptern kompatibel ist.[1]
Da jedoch der Emulator *Dualis* in der Version 20.3, der für die Entwicklung nötig ist, nicht mit der neuen Bibliothek zurecht kommt und die alte Bibliothek bei Schreibzugriffen durch den Nintendo DS auf die Speicherkarte Probleme bereitet, wurde bei der Studienarbeit eine modifizierte Version der alten fatlib verwendet, die *Dragon libfat* von Shaun Taylor.[2]

### 3.1.4 MAD Decoder

Die Programmierung eines MP3 Decoders würde den Rahmen dieser Arbeit bei weitem sprengen und so wurde auf die *MAD Bibliothek* (MPEG Audio Decoder) der Firma Underbit Technologies zurückgegriffen. Der Decoder unterstützt verschiedene MPEG-Formate und den Layern I bis III. Darüber hinaus ist der Decoder Fixed-Point basiert, rechnet also mit Integer-Werten. Dies ist sehr wichtig, da die Hardware des Nintendo DS keine Gleitkomma-Operationen unterstützt und sie emulieren müsste. Dies ist entsprechend aufwändig.[3]

[1] http://chishm.drunkencoders.com/libfat/
[2] http://www.dragonminded.com/?loc=ndsdev/LibFATDragon
[3] http://www.underbit.com/products/mad/

### 3.1.5 GUI

Grundlage für die Benutzeroberfläche ist das *GUI Toolkit* von Tobias Weyand. Es stellt alle wichtigen Elemente wie zum Beispiel Labels, Buttons und Listboxen zur Verfügung. Darüber hinaus nimmt die GUI die Eingaben des Benutzers entgegen und reagiert entsprechend darauf.[1]

## 3.2 Die wichtigsten Funktionen

Die Software besteht grundlegend aus zwei Programmteilen. Zum einen dem Code für den ARM7 Prozessor und zum anderen aus dem Code für den ARM9 Prozessor. Die Quellen werden jeweils kompiliert und anschließend zu einer ausführbaren Datei gelinkt.

Der ARM7 Prozessor überprüft regelmäßig ob Eingaben getätigt wurden. Hat der Nutzer eine Taste gedrückt oder das Touchpad berührt, wird der Status in den dafür vorgesehen Registern gespeichert und kann nun von anderen Programmteilen abgefragt werden. Darüber hinaus übernimmt der ARM7 auch die Ausgaben, wie zum Beispiel das Abspielen des Soundpuffers.

Die eigentliche Arbeit erledigt der schnellere ARM9 Prozessor. Nach dem Programmstart werden Initialisierungen für die Grafikausgabe vorgenommen und anschließend das Dateisystem und die GUI initialisiert. Nachdem dies abgeschlossen ist, wartet das Programm auf Eingaben.

Der Nutzer kann nun Playlisten erstellen, speichern und laden. Er kann Audiodateien abspielen und dabei die Wiedergabe mittels Equalizers manipulieren. Darüber hinaus ermöglicht das Programm das Editieren der ID3v1 Tags einer MP3 Datei. Da der Kern des Programms das Abspielen der Musikdateien ist, soll dieser Vorgang nun genauer dargestellt werden. Nach dem Start prüft das Programm regelmäßig, ob eine Datei ausgewählt ist und abgespielt werden soll. Wenn dies der Fall ist, wird die Datei lesend geöffnet und der ID3 Tag ausgelesen, sofern er vorhanden ist. Die Informationen werden dann angezeigt und die Funktion `decodeMP3(FAT_FILE *handle)` aufgerufen. Ihr wird der Handle, ein Zeiger auf die geöffnete Datei, mitgegeben.

[1] http://tobw.net/

### 3.2.1 decodeMP3

Die Funktion `decodeMP3(FAT_FILE *handle)` reserviert zunächst Speicherplatz für einen Eingabepuffer (`inBuffer`) und zwei Ausgabepuffer (`outBufferL` und `outBufferR`). In den Eingabepuffer werden später die zu dekodierenden Daten geschrieben, in die beiden Ausgabepuffer werden die dekodierten Informationen für den linken und rechten Audiokanal geschrieben. Anschließend wird der MAD Decoder initialisiert. Dazu bekommt er Zeiger auf Speicherbereiche und Funktionen mitgegeben, die zum Dekodieren mitunter optional, teilweise zwingend nötig sind.
Nach der Initialisierung wird der Decoder aufgerufen und beginnt die Dekodierung. Nachdem er einen Teil der Daten dekodiert hat, schreibt er das Ergebnis in den Ausgabepuffer für den linken Lautsprecher und in den Puffer für den rechten Lautsprecher und wartet auf neue Daten.
Ist die Dekodierung abgeschlossen, wird der Speicher aufgeräumt und die Funktion `decodeMP3(FAT_FILE *handle)` beendet.

**Listing 2:** Dekodieren einer MP3

```
1  int decodeMP3(FAT_FILE *handle) {
2      struct Buffer bufferLocal;
3      struct mad_decoder decoder;
4      int result;
5
6      isStopped = false;
7      isFinished = false;
8      stop = false;
9      pause = false;
10     fastForward = false;
11     fastBackward = false;
12
13     initBuffers();
14
15     bufferLocal.handle = handle;
16     bufferLocal.size = FAT_GetFileSize();
17
18     mad_decoder_init(&decoder, &bufferLocal, input, header, filter,
           output, error, 0 /* message */);
19
20     initAudio();
21
22     CommandPlaySampleSound(outBufferL, OUTPUT_BUFFER_SIZE, 44100,
           127*volume, 0, 0);
23     CommandPlaySampleSound(outBufferR, OUTPUT_BUFFER_SIZE, 44100,
           127*volume, 127, 0);
24
25     result = mad_decoder_run(&decoder, MAD_DECODER_MODE_SYNC);
26
27     mad_decoder_finish(&decoder);
28
29     cleanUp();
30
31     return result;
32 }
```

Listing 3 zeigt noch einmal die Initialisierung des Decoders.

**Listing 3:** MAD Decoder initialisieren

```
1  mad_decoder_init(&decoder, &bufferLocal, input, header, filter,
       output, error, 0 /*message*/);
```

Der erste Parameter der Funktion, `&decoder`, ist eine Referenz auf die Struktur `mad_decoder`, der zweite Parameter, `&bufferLocal`, ist eine Referenz auf einen Speicherbereich, in dem die zu dekodierenden Daten geschrieben werden. Die Referenz auf den Speicherbereich, sowie die restlichen Angaben im Funktionsaufruf werden in die Struktur eingetragen, sofern sie vorhanden sind. Zwingend nötig sind lediglich die Funktionen `input` und `output`, da der MAD Decoder ein reiner Decoder ist und keinerlei Funktionalität zum Einlesen und Ausgeben der Daten besitzt. Das ermöglicht eine weitgehende Unabhängigkeit und damit Flexibilität. Die anderen zusätzlichen Funktionen sind optional und dazu gedacht, das Dekodieren beeinflussen zu können, ohne die Bibliothek umschreiben zu müssen.

Nun folgen detaillierte Erläuterungen zu den genannten Funktionen, die dem Decoder übergeben werden und anschließend wird eine Struktur besprochen, die die Kommunikation zwischen den Prozessoren ermöglicht.

### 3.2.2 input

`input` ist eine Funktion, die den Eingabepuffer (`inBuffer`) mit den zu dekodierenden Daten füllt. Der `inBuffer` ist gewöhnlich kleiner als die zu dekodierende Datei. Also wird der erste Teil der zu dekodierenden Datei ausgelesen und in den Puffer geschrieben. Der Inhalt kann nun dekodiert werden. Ist der komplette Inhalt abgearbeitet, wird der Puffer mit dem nächsten Abschnitt der zu dekodierenden Datei gefüllt. Dies geschieht so lange, bis die komplette Datei dekodiert ist.

**Listing 4:** Füllen des Eingabepuffers

```
1  enum mad_flow input(void *data, struct mad_stream *stream) {
2      buffer = reinterpret_cast<Buffer*>(data);
3      uint32 readSize, remaining;
4      u8 *readStart;
5
6      if(stream->next_frame!=NULL) {
7          remaining = stream->bufend-stream->next_frame;
8          memmove(inBuffer, stream->next_frame, remaining);
9          readStart = inBuffer+remaining;
10         readSize  = INPUT_BUFFER_SIZE - remaining;
11     } else {
12         readSize = INPUT_BUFFER_SIZE;
```

```
13          readStart = inBuffer;
14          remaining = 0;
15      }
16
17      readSize = FAT_fread((void*)readStart, sizeof(u8), readSize,
            buffer->handle);
18      if (readSize <= 0) {
19          isFinished = true;
20          return MAD_FLOW_STOP;
21      }
22
23      // Pipe the buffer content to libmad's stream decoder facility
24      mad_stream_buffer(stream, inBuffer, readSize+remaining);
25
26      u32 size = buffer->size;
27      u32 pos = FAT_ftell(buffer->handle);
28      if (!changePositionInStream)
29          positionInStream = ((100.0f/size)*pos)/100.0f;
30
31      duration = (FAT_GetFileSize()/(headerBitrate/1000)*8)/1000;
32      playtime = (pos/(headerBitrate/1000)*8)/1000;
33
34      return MAD_FLOW_CONTINUE;
35  }
```

### 3.2.3 output

Der Decoder schreibt das Ergebnis in eine Struktur vom Typ `mad_pcm`. Die Funktion `output` liest die Daten für die beiden Kanäle aus dem Struct aus und schreibt sie in die Ringpuffer, für den linken und den rechten Lautsprecher (`outBufferL` und `outBufferR`), die der Nintendo DS kontinuierlich abspielt. Da die Daten schneller dekodiert als wiedergegeben werden, unterbricht die Funktion das Dekodieren und wartet bis die Ausgabepuffer abgespielt wurden, bevor sie neuen Daten hinein schreibt.

**Listing 5:** Schreiben in den Ausgabepuffer

```
1   enum mad_flow output(void *data, struct mad_header const *header,
        struct mad_pcm *pcm) {
2       static uint32 writePos = 0;
3       unsigned int nchannels, nsamples;
4       mad_fixed_t const *left_ch, *right_ch;
5
6       nchannels   = pcm->channels;    // Number of channels
7       nsamples    = pcm->length;      // Number of samples/channel
8       left_ch     = pcm->samples[0];  // PCM output samples
9       right_ch    = pcm->samples[1];  // PCM output samples
10
11      while (weHaveToWait(writePos, writePos+nsamples*2));
12
13      while (nsamples--) {
14          signed int sample;
15
16          // outBufferL and outBufferR are implemented as ringbuffers
17          if (writePos+2 >= OUTPUT_BUFFER_SIZE) {
```

```
18              writePos = 0;
19          }
20
21          sample = scale(*left_ch++);
22          outBufferL[writePos] = (char)((sample >> 0) & 0xff);
23          writePos++;
24          outBufferL[writePos] = (char)((sample >> 8) & 0xff);
25          writePos++;
26
27          if (nchannels == 2) {
28              sample = scale(*right_ch++);
29              outBufferR[writePos-2] = ((sample >> 0) & 0xff);
30              outBufferR[writePos-1] = ((sample >> 8) & 0xff);
31          }
32      }
33
34      enum mad_flow madFlow = handleInstruction();
35
36      return madFlow;
37  }
```

### 3.2.4 header

Vor dem Dekodieren jedes Frames liest der Decoder den Header aus und speichert die Informationen in eine Struktur vom Typ `mad_header`. Da einige der Informationen auch für den Nutzer von Interesse sind, liest die Funktion `header` die Werte aus und macht sie für die GUI verfügbar, wo sie dem Benutzer angezeigt werden. Dadurch ist er zum Beispiel darüber informiert, welche Bitrate die MP3 hat oder ob sie mono beziehungsweise stereo wiedergegeben wird.

**Listing 6:** Auslesen von Informationen aus den Frame-Headern

```
1  enum mad_flow header(void *data, const mad_header *header) {
2      headerLayer = header->layer;
3      headerBitrate = header->bitrate;
4      headerSamplerate = header->samplerate;
5      headerChannelmode = header->mode;
6      headerEmphasis = header->emphasis;
7
8      return MAD_FLOW_CONTINUE;
9  }
```

### 3.2.5 filter

Die Funktion `filter` wird vom Decoder nach dem Dekodieren aufgerufen und ermöglicht es, die vorliegenden Audiodaten zu manipulieren, bevor sie ausgegeben werden.
Um dem Nutzer die Möglichkeit zu geben, den Klang an seine Wünsche

anzupassen und zum Beispiel hohe Frequenzen zu dämpfen und dafür die Bässe hervorzuheben, wurde als Filter ein Equalizer implementiert. Der Equalizer macht es sich zu Nutze, dass der Decoder das Audiosignal bereits in 32 Frequenzbänder aufgeteilt hat. So ist es ohne großen Aufwand möglich, einzelne Frequenzbereich nachträglich zu regulieren.

**Listing 7:** Dekodiertes Signal nachbearbeiten

```
1  enum mad_flow filter(void *data, const struct mad_stream *stream,
       struct mad_frame *frame) {
2
3      mad_fixed_t Filter[32];
4
5      Filter[ 0] = mad_f_tofixed(subbands01);
6      Filter[ 1] = mad_f_tofixed(subbands01);
7      Filter[ 2] = mad_f_tofixed(subbands01);
8      Filter[ 3] = mad_f_tofixed(subbands01);
9      Filter[ 4] = mad_f_tofixed(subbands02);
10     Filter[ 5] = mad_f_tofixed(subbands02);
11     Filter[ 6] = mad_f_tofixed(subbands02);
12
13     [..]
14
15     Filter[28] = mad_f_tofixed(subbands10);
16     Filter[29] = mad_f_tofixed(subbands10);
17     Filter[30] = mad_f_tofixed(subbands10);
18     Filter[31] = mad_f_tofixed(subbands10);
19
20     int channel, sample, samples, subband;
21
22     samples=MAD_NSBSAMPLES(&frame->header);
23
24     if (frame->header.mode!=MAD_MODE_SINGLE_CHANNEL)
25         for (channel=0; channel<2; channel++)
26             for (sample=0; sample<samples; sample++)
27                 for (subband=0; subband<32; subband++) {
28                     frame->sbsample[channel][sample][subband] =
                           FIX_MUL(frame->sbsample[channel][sample][
                           subband], Filter[subband]);
29                 }
30
31     else
32         for (sample=0; sample<samples; sample++)
33             for (subband=0; subband<32; subband++) {
34                 frame->sbsample[channel][sample][subband] = FIX_MUL(
                       frame->sbsample[channel][sample][subband],
                       Filter[subband]);
35             }
36
37     return MAD_FLOW_CONTINUE;
38 }
```

### 3.2.6 error

Die Funktion `error` ermöglicht es, im Fehlerfall das Dekodieren zu unterbrechen. Da das Dekodieren jedoch nicht unterbrochen werden soll, wenn

hin und wieder ein Fehler auftritt, wird von dieser Möglichkeit kein Gebrauch gemacht.

**Listing 8:** Fehler abfangen

```
1 enum mad_flow error(void *data, struct mad_stream *stream, struct
      mad_frame *frame) {
2     return MAD_FLOW_CONTINUE;
3 }
```

### 3.2.7 message

`message` ist die letzte Funktion, die dem Decoder mit auf den Weg gegeben werden kann, von der aber bei diesem Programm kein Gebrauch gemacht wurde. Sie findet lediglich im asynchronen Modus Anwendung, und wird aufgerufen, wenn ein Elternprozess eine Nachricht sendet.

### 3.2.8 Kommunikation zwischen den Prozessoren

Die Daten in den Ausgabepuffern werden vom ARM7 Prozessor des Nintendo DS kontinuierlich ausgelesen und daraus das Signal für die Lautsprecher errechnet. Allerdings erst, nachdem der Vorgang explizit gestartet wurde. Folglich muss zeitgleich mit dem Dekodieren durch den ARM9 Prozessor auch das Abspielen der dekodierten Daten durch den ARM7 Prozessor gestartet werden. Die Kommunikation zwischen den beiden Prozessoren, die hierfür notwendig ist, ist durch eine Command-Struktur realisiert. Die Informationen, die von einem zum anderen Prozessor übertragen werden sollen, werden vom ersten Prozessor in einen Speicherbereich geschrieben, auf den auch der andere Prozessor zugreifen kann.

**Listing 9:** Command Struktur für den Abspielbefehl

```
1 struct PlaySampleSoundCommand {
2     void* data;      // Points to outbuffer
3     int length;      // Size of outbuffer
4     int frequency;   // Frequenzy in Hz
5     int volume;      // Playbackvolume
6     int panning;     // Balance between left and right
7     int format;      // Bitrate, 8 or 16 bit
8 };
```

Listing 9 zeigt die Struktur, in die der ARM9 Prozessor die Informationen schreibt, die der ARM7 Prozessor benötigt um die Lautsprecher anzusteuern. Hierdurch ist es dem ARM9 möglich, dem ARM7 mitzuteilen, in welchem Speicherbereich sich Soundinformationen befinden. Damit der Ne-

benprozessor die Daten korrekt abspielen kann, benötigt er noch weitere Informationen zur Abtastfrequenz und Bitrate. Außerdem muss er wissen, mit welcher Lautstärke das Signal wiederzugeben ist und mit welcher Gewichtung auf dem linken, beziehungsweise rechten Lautsprecher.

## 3.3 Benutzeroberfläche

Die Interaktion mit dem Programm findet über die Benutzeroberfläche statt. Durch Anklicken der Buttons und Listeneinträge ist es möglich, Audiodateien im MP3 Format in Playlisten zu organisieren, wiederzugeben und die Wiedergabe zu steuern. Darüber hinaus kann zum Beispiel mittels Equalizer der Klang über Schieberegler beeinflusst werden. Da trotz der verhältnismäßig großen Displays nicht genügend Platz zur Verfügung steht, um alle nötigen Elemente der Benutzeroberfläche gleichzeitig darzustellen, sind die Funktionen in fünf Bereiche gruppiert und über einen Karteireiter auswählbar.

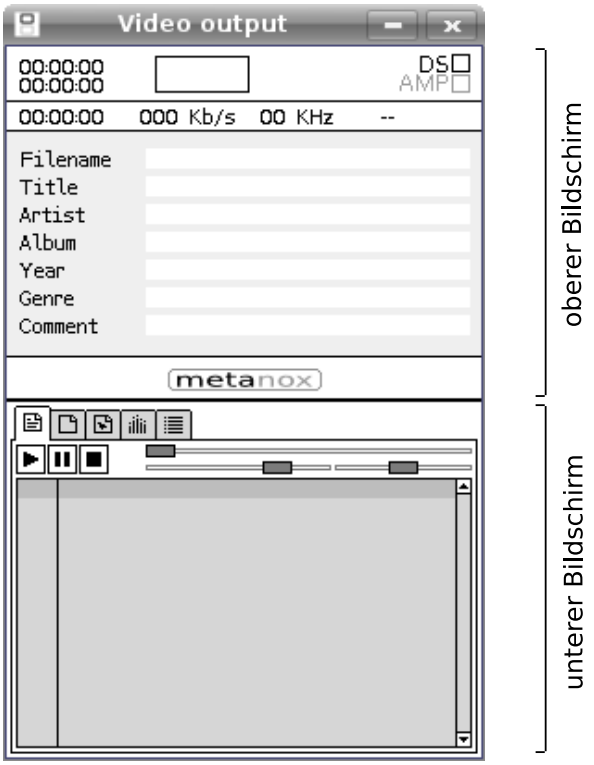


**Abbildung 13:** Der Screenshot zeigt die Benutzeroberfläche des MP3 Players, wie sie nach dem Start angezeigt wird

Folgende Auflistung gibt eine Übersicht über die Bereiche des Programms:

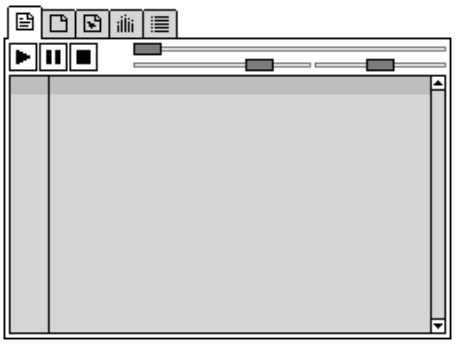

**Abbildung 14:** Steuerung der Wiedergabe und Anzeigen einer Playliste

**Playliste abspielen**
Der erste Karteireiter umfasst alle für die Wiedergabe relevanten Komponenten. Das ist zunächst einmal die Playliste, aus der die Audiodatei gewählt werden kann, die abgespielt werden soll. Darüber hinaus natürlich Buttons zum Starten, Stoppen und Pausieren des Abspielvorgangs. Die Slider ermöglichen es, in der Wiedergabe vor- und zurückzuspringen, die Lautstärke zu regulieren und die Balance zu variieren.


**Abbildung 15:** Laden einer Playliste

**Playliste laden**
Der zweite Karteireiter enthält einen Dateibrowser, der es ermöglicht, durch die Ordnerstruktur auf der Speicherkarte zu navigieren und gespeicherte Playlisten zu laden.

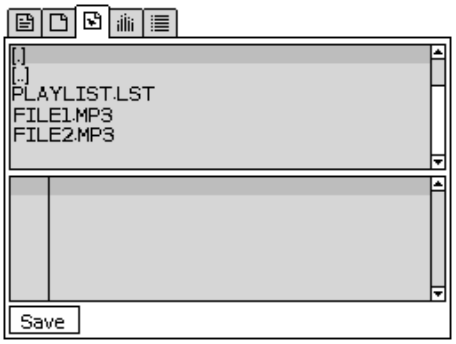


**Abbildung 16:** Erstellen und editieren der Playlisten

**Playliste editieren**
Der dritte Karteireiter enthält einen Dateibrowser und eine Listbox, so dass im Dateibrowser zu den Ordnern mit MP3 Dateien navigiert werden kann. Durch Anklicken dieser Dateien werden sie in der Listbox hinzugefügt. Die erstellte Auflistung kann als Playlist gespeichert werden.
Bestehende Playlisten können ebenfalls durch Anklicken im Browser geladen und anschließend editiert werden.

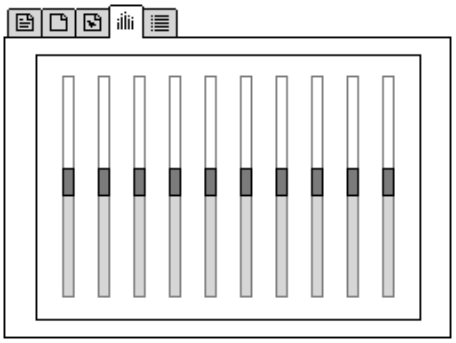

**Abbildung 17:** Der Equalizer

**Equalizer**
Der vierte Karteireiter ermöglicht den Zugriff auf den Equalizer. Über die zehn Schieberegler kann die Wiedergabelautstärke einzelner Frequenzbereiche verstärkt oder abgeschwächt und dadurch der Klang angepasst werden.

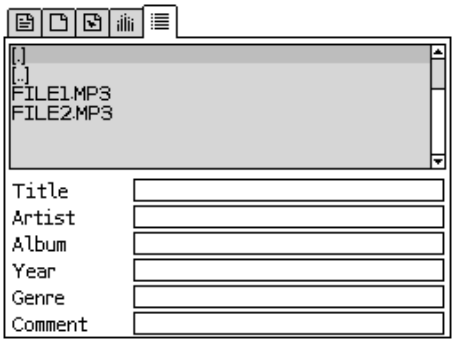


**Abbildung 18:** Editieren der ID3 Tags

**MP3Tag editieren**
Der fünfte Karteireiter gibt die Möglichkeit, die Zusatzinformationen, die in einer MP3 Datei als ID3 Tag abgelegt werden können, zu editieren. Hierzu kann im Dateibrowser zur entsprechenden Datei navigiert und deren Tag durch Anklicken geladen werden. Die Einträge werden im unteren Bereich in Textfeldern angezeigt.

Neben diesen fünf Bereichen bietet die Oberfläche noch weitere Interaktionsmöglichkeiten. Zum einen eine Tastatur für die Eingabe von Schriftzeichen und zum anderen Messageboxen für Rückmeldungen:

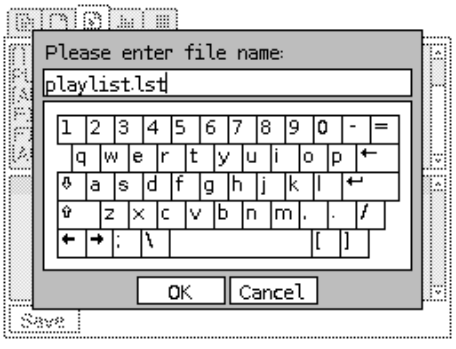


**Abbildung 19:** Tastatur zur Eingabe von Texten

Beim Abspeichern der Playlisten muss ein Dateiname angegeben werden und zum Editieren der ID3 Tags muss ebenfalls die Möglichkeit der Zeicheneingabe bestehen. Da die Konsole selbst kein Tastenfeld besitzt, blendet das Programm eine Tastatur ein, wenn sie benötigt wird. Durch Anklicken der Tasten mit Hilfe des Eingabestiftes kann ein beliebiger Text verfasst werden.


**Abbildung 20:** Messageboxen zur Benachrichtigung

Messageboxen ermöglichen es, den Nutzer auf wichtige Dinge hinzuweisen. Das ist zum Beispiel dann der Fall, wennn versucht wird, eine leere Playliste abzuspeichern. Da dies nur wenig Sinn macht, informiert das Programm durch eine Messagebox darüber.

Der zweite Bildschirm wird ebenfalls verwendet. Da er aber nicht als Touchscreen nutzbar ist, dient er lediglich der Visualisierung zusätzlicher Informationen:

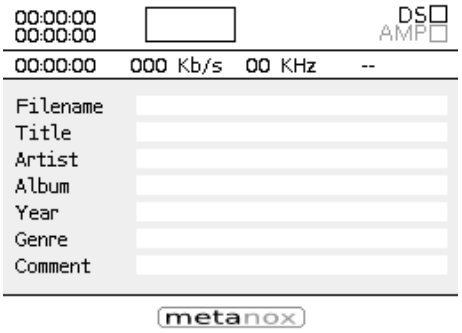


**Abbildung 21:** Anzeigen der ID3 Tags

Über den zweite Bildschirm werden Informationen bezüglich der aktuell abgespielten Audiodatei angezeigt. Darunter die Abspielzeit, Bitrate, Samplingfrequenz, ob Mono- oder Stereowiedergabe und, falls vorhanden, die Einträge der ID3 Tags. Darüber hinaus gibt es eine Visualisierung der Frequenzgänge.

# 4 Ergebnisse

Im Wesentlichen wurden die Ziele dieser Studienarbeit erreicht. Das Programm ermöglicht das Abspielen von MP3 Dateien, die sich auf einer Speicherkarte befinden. Die Bedienung erfolgt über eine grafisch ansprechende Oberfläche, wie man sie von gängiger Software für Computer kennt. Für die Implementierung war ein tiefergehendes Verständnis bezüglich Audiodateien und deren Kompression und insbesondere des Aufbaus und der Besonderheiten des MP3 Formats nötig. Die Aneignung dieses Wissens erwies sich als aufwändiger als vermutet, da MP3 Dateien weitaus komplexer sind als erwartet. Neben den technischen Aspekten fließen bei der Enkodierung auch Wahrnehmungseffekte der Psychoakustik, die auf empirischen Studien beruhen, mit ein.

Ohne die verfügbaren Bibliotheken, die das Programmieren für den Nintendo DS wesentlich erleichtern, wäre es kaum möglich gewesen in diesem Zeitrahmen ein solches Programm zu schreiben. Jedoch mussten die Bibliotheken und deren Funktionsweise erst verstanden werden, was zwar einen hohen Aufwand für die Einarbeitung, aber ebenso einen großen Lerneffekt zur Folge hatte.

Das Entwickeln eines Programms für eine Konsole bringt eine Reihe an Schwierigkeiten mit sich. Da der Quelltext auf einem Rechner erstellt und kompiliert wird, muss das Programm zum Testen jedes mal auf eine Speicherkarte übertragen und davon geladen werden. Alternativ ermöglichen Emulatoren die Ausführung des Programms direkt auf dem Computer, auf dem es programmiert wird. Dies vereinfacht den Testzyklus, allerdings unterstützen die Emulatoren nicht die gesamte Funktionalität der echten Hardware. Das führt dazu, dass man ein Programm entwickeln muss, dass sowohl auf dem Nintendo DS läuft, als auch von mindestens einem Emulator ausgeführt werden kann.
Ebenso war es eine ungewohnte Erfahrung, nur sehr begrenzt Ressourcen zur Verfügung zu haben. Vor allem der Grafikspeicher, der lediglich 656 KB groß ist, war ein limitierender Faktor.

## 4.1 Ausblick

Der MP3 Player für den Nintendo DS ist voll funktionsfähig, aber dennoch gibt es zahlreiche Verbesserungsmöglichkeiten. Grundlegend wäre ein Austausch verschiedener Bibliotheken, wie etwa der libnds und der libfat sinnvoll, um neu hinzugekommene Funktionen nutzen zu können.

Am Programm selbst können auch noch einige Verbesserungen vorgenommen werden. So ist das Editieren der Playliste mitunter etwas unintuitiv. Die Möglichkeit, die Reihenfolge von Einträgen per Drag&Drop zu verändern, wäre eine wünschenswerte Funktion.
Auch der Equalizer bietet bislang nur eine sehr rudimentäre Ausstattung. So ist es nicht möglich, die Einstellungen zu speichern, oder gar Default-Einstellungen für verschiedene Stilrichtungen wie Klassik, Rock, Jazz und so weiter zu laden.
Neben den ID3 Tags der Version 1 gibt es mittlerweile auch eine 2. Version, die es einerseits ermöglicht noch mehr Informationen zur Audiodatei abzuspeichern, andererseits allerdings weitaus komplexer ist. Diese Version wird vom Programm noch nicht unterstützt. Die Einträge können weder angezeigt noch editiert werden. Auch hier besteht somit noch Verbesserungsbedarf.
Bei vielen Programmen ist es mittlerweile üblich, dass der Nutzer das Erscheinungsbild durch das Laden neuer Skins an seine Wünsche und seinen Geschmack anpassen kann. Diese Möglichkeit bietet der MP3 Player ebenfalls noch nicht.

# Literatur


[1] DevKit Pro.
http://sourceforge.net/project/showfiles.php?group_id=114505), [Online, Juli 2007].

[2] DevScene. http://dev-scene.com/, [Online, Juli 2007].

[3] Die Geschichte der Musikkassette.
http://www.mabroselvisworld.com/MC/mc-geschichte.htm, [Online, Juli 2007].

[4] MemoryMap. http://dev-scene.com/NDS/Tutorials_Day_2, [Online, Juli 2007].

[5] MP3-Grundlagen: Aufbau und Funktion.
http://www.tecchannel.de/index.cfm?pid=315&pk=401059&p=1, [Online, Juli 2007].

[6] Sony Celebrates Walkman 20th Anniversary.
http://www.sony.net/SonyInfo/News/Press_Archive/199907/99-059/, [Online, Juli 2007].

[7] Technical Specs for Nintendo DS. http://www.nintendo.com/techspecds, [Online, Juli 2007].

[8] The MPEG Home Page. http://www.chiariglione.org/mpeg/, [Online, Juli 2007].

[9] Tonbandmuseum - Die CC Kassette.
http://www2.tonbandmuseum.info/die-cc-kassette.0.html, [Online, Juli 2007].

[10] Chism. LibFAT. http://chishm.drunkencoders.com/libfat/, [Online, Juli 2007].

[11] Joachim Böhringer et. al. In *Kompendium der Mediengestaltung*, 2002.

[12] Fraunhofer Institut. Die MP3-Geschichte.
http://www.iis.fraunhofer.de/bf/amm/mp3history/index.jsp, [Online, Juli 2007].

[13] Hans W. Lang. Huffman-Code. http://www.inf.fh-flensburg.de/lang/algorithmen/code/huffman/huffman.htm, [Online, Juli 2007].

[14] Library of Congress Prints and Photographs. Thomas Edison mit Zinnfolienphonograph.
http://memory.loc.gov/service/pnp/cwpbh/04000/04044v.jpg, [Online, Juli 2007].

[15] Bertrand Petit. Dokumentiertes Beispiel der low-level API der LibMAD.
http://www.bsd-dk.dk/

[16] Ralf Rankers. Geschichte der Tonaufzeichnung.
http://www.tonaufzeichnung.de, [Online, Juli 2007].

[17] Thorsten Scheuermann. MP3-Audiokompression. http://www.rz.uni-karlsruhe.de/ Thomas.Stirner/uni/Theorie/vortrag14/vortrag14.html, [Online, Juli 2007].

[18] Shaun Taylor. LibFAT Dragon. http://www.dragonminded.com/?loc=ndsdev/LibFATDragon, [Online, Juli 2007].

[19] Underbit Technologies. LibMAD. http://www.underbit.com/products/mad/, [Online, Juli 2007].

[20] Tobias Weyand. NDS Project. http://tobw.net/, [Online, Juli 2007].