FMS - Dokumentation

Daniel Grün

26. Februar 2003

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Was soll FMS leisten?

FMS ist ein virtueller Synthesizer zur Erzeugung von Klängen, Geräuschen und deren Strukturierung. Mögliche Einsatzgebiete sind Sprachsynthese, Klangdemonstrationen, Rauscherzeugung oder Abspielen musikalischer Abläufe. Es handelt sich um ein offenes, modulares System in ständiger Weiterentwicklung FMS soll auch eine leistungsfähige c++-API für Soundsynthesezwecke bieten und anderen Programmierern die Möglichkeit geben, sich an der Weiterentwicklung zu beteiligen.

Deshalb ist FMS unter der GNU General Public License [2] veröffentlicht und für jedermann frei und auch im Quellcode verfügbar [3].

1.2 Was kann FMS bisher leisten?

- Allgemein:

  - Speichern der Wellenformen in Tabellen
  - grafische Ausgabe (Oszillogramm)
  - Wiedergabe (Klänge) über zwei Kanäle (Stereo)
  - Unterstützung von .wav-Dateien

- Spezifikationen:

  - Erzeugen beliebiger Frequenzen und Lautstärken
  - Amplituden- und Ringmodulation
  - Frequenzmodulation
  - additive Mischung von Klängen
  - Rauscherzeugung durch gewichtete Zufallsfunktionen
  - Abfolge von Klangsequenzen („Soundpaketen“)
    bis hin zur Abspielung von Musikstücken
  - Emulation von Musikinstrumenten

Kapitel 2

User's Guide - Wie man FMS benutzt

2.1 Installation

Das z.B. von [3] heruntergeladene FMS-Sourcepaket wird nach dem Entpacken mit dem Linux-üblichen Installationsablauf von `make` und `make install` installiert. Voraussetzung für eine erfolgreiche Installation ist ein Linux-System mit gcc-g++-Compiler, einer Soundkarte die /dev/dsp unterstützt und den Bibliotheken SDL [5] und SDL_gfx [6] für grafische Anwendungen. Für die eingeschränkte Tcl/Tk-Oberfläche wird die `tclsh` und `wish` benötigt, die Qt-Oberfläche `qfms` läuft ab Qt-Version 3.1, die direkt von Trolltech [7] heruntergeladen werden kann.

2.2 fmautofile

Um die Eingabe natürlicher Klänge zu erleichtern gibt es fmautofile. Es wandelt eine WAV-Datei, die nur ein einzelnes Oszillogramm (also nur eine Iteration eines frequenten Klanges) enthält in eine FMS-Sounddatei im Modus 1 um.

fmautofile (-s x) (-f x)

Das Programm liest die im aktuellen Verzeichnis befindliche WAV-Datei in.wav ein und schreibt die Ausgabe in out.fms. Dabei kann mit -s ein Weichzeichnungsfaktor und mit -f der maximale Unschärfefaktor (Komprimierungsrate) der Ausgabedatei angegeben werden. Es ist empfehlenswert mit diesen Werten etwas herumzuexperimentieren und die Ergebnisse mit xfmdisplay zu überprüfen. Um diese Vorgänge weiter zu vereinfachen, kann man das Script autotest benutzen.

2.3 fmdisplay

Mit fmdisplay kann man FMS-Oszillogramme auf Konsolen-Ebene darstellen. Dabei übernimmt das Programm folgende Parameter:

fmdisplay [FMS-Datei] [x] [y]

Dabei steht x und y für die Anzahl der Zeichen auf der x- bzw. y-Achse. Im Allgemeinen sollten 80 und 24 gewählt werden.

2.4 fmfile

Fmfile liest die Angaben über eine FMS-Sounddatei manuell ein und schreibt die Datei auf die Festplatte. Programmaufruf:

fmfile [FMS-Datei]

Zunächst fragt das Programm nach dem Speicherungsmodus. Die drei möglichen Modi sind:

1. alle Werte
   In diesem Modus werden einzelne x-y-Werte vom Benutzer angegeben. Die dazwischenliegenden Werte werden von fmfile erzeugt und in die Datei geschrieben. Nur in seltenen Fällen lohnt es sich, diesen Modus gegenüber Modus 2 zu bevorzugen.

2. Werte mit Verbindungslinien
   Dieser Modus speichert einzelne x-y-Werte und den Typ der Verbindungslinie zwischen ihnen. Dies spart meist sehr viel Speicherplatz gegenüber Modus 1 und ermöglicht außerdem saubere Erzeugung von Kurven.

3. Midi-Modus
   Dieser Speicherungsmodus sollte gewählt werden, um Musikstücke in FMS-Midi-Dateien umzuwandeln.

Der weitere Programmdialog sollte selbsterklärend sein.

2.5 fmplay

Fmplay spielt FMS-Sounddateien aller Modi gleichzeitig und/oder nacheinander, optional mit schwankender Lautstärke und Frequenz ab. Eine ausführliche Liste der möglichen Abspieloptionen wird asugegeben, wenn man fmplay ohne weitere Parameter aufruft.

2.6 qfms

Um die Funktionen von fmplay auch ohne Konsolenbefehle zugänglich zu machen, arbeitet der Autor im Moment an einer Qt-GUI. Betatester können sich gerne bei mir melden[4].

2.7 xfmdisplay

Auf einfachste Weise stellt xfmdisplay auch gemischte und in der Frequenz verschiedene FMS-Sounddateien gleichzeitig in einer grafischen Umgebung dar. Der Programmaufruf ist dabei wie folgt:

xfmdisplay (-x x) (-y x) (-a) [FMS-Datei] (-r x) (-v x) ([FMS-Datei])

Dabei kann mit -r die Wiederholungen innerhalb des Fensters und mit -v die Lautstärke der zuvor angegebenen FMS-Datei festgelegt werden. Die Optionen -x und -y übergeben dem Programm die gewünschte Größe des geöffneten Fensters (in Pixeln), ist die Option -a gegeben, wird der Aufbau der Ausgabe animiert. Es kann eine beliebig lange Liste von FMS-Dateien mit -r und/oder -v-Optionen angegeben werden.

Kapitel 3

Technical Guide - Wie FMS funktioniert

3.1 Theorie

Die von FMS zur Erzeugung eines Klanges genutzten Daten sind die so genannten Oszillogramme der jeweiligen Klänge. Die Angaben über das Oszillogramm eines Klanges werden entweder Wert für

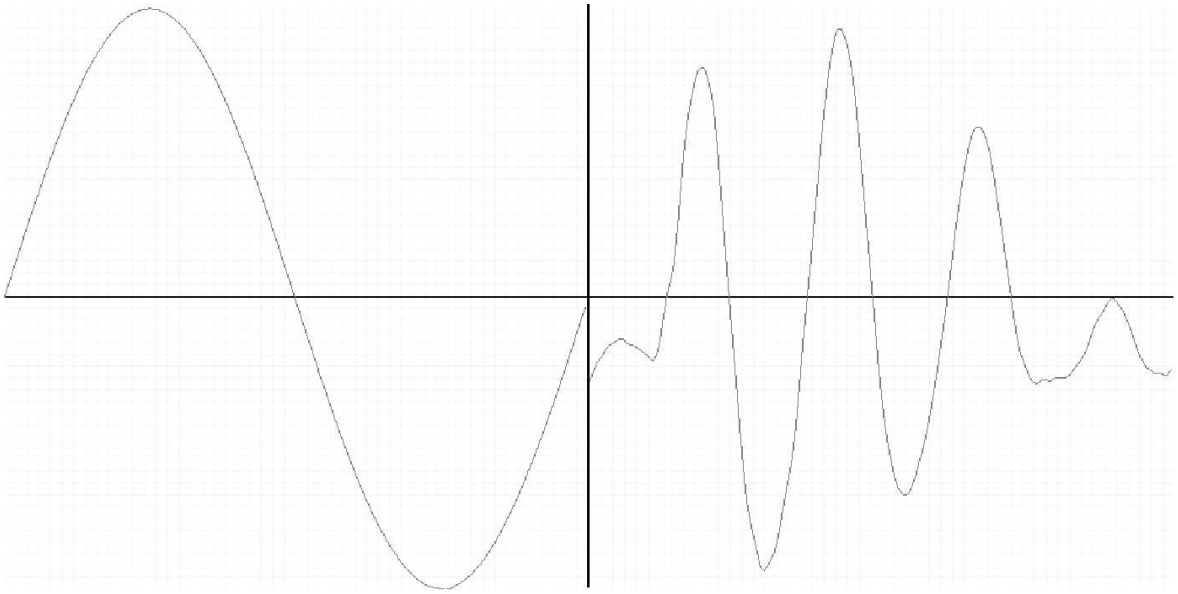
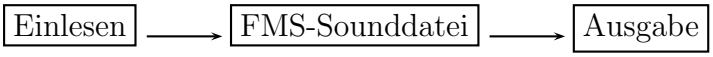


Abbildung 3.1: Oszillogramme von Sinuston und O-Laut

Wert in eine Datei gespeichert, oder durch nur wenige Werte mit Angabe ihres Zeitindexes und der Art der Verbindungslinie zwischen ihnen beschrieben und gespeichert. Beim gezeigten Sinuston wären dies nur fünf Werte, z.B. bei den Zeitindices 0, 100, 200, 300 und 400, wobei die Verbindungslinien Sinuskurvenabschnitte sind. Beim O-Laut werden mehr Werte benötigt.

Berechnet man die Werte des so gespeicherten Oszillogramms und gibt sie oft hintereinander als Soundbytes aus, so hört man aus dem Lautsprecher den vorher aufgenommenen Ton mit beliebiger Frequenz (je nachdem, wie viele Werte man pro Wiederholung ausgibt, wie viele ausgegebene Werte also einem Schritt im Zeitindex des Oszillogramms entsprechen). Durch Veränderung dieses Wertes, der bei jedem Zeitschritt zum Zeitindex des aktuell zu berechnenden Wertes hinzugezählt wird, lässt sich Frequenzmodulation realisieren. Die Multiplikation des jeweiligen berechneten Wertes mit dem Wert eines anderen Oszillogramms macht Hüllkurven möglich.

3.2 Schematischer Aufbau



Dabei können die verschiedenen Aufgaben durch den modularen Aufbau von FMS auch von verschiedenen Programmen erledigt werden. So gibt es für das Einlesen schon die Programme fmfile und fmautofile, eine grafische Lösung unter Verwendung der Qt-Bibliothek[7] ist in der Entwicklung. Zur Aufbereitung und Ausgabe der erzeugten FMS-Sounddateien gibt es fmplay (Ausgabe als Klang) oder fmdisplay und xfmdisplay (grafische Darstellung). Auch hier besteht bereits eine noch nicht vollständige GUI-Version.

Programmiertechnisch wird beim Schreiben in FMS-Sounddateien die fmofstream-API benutzt, die ohne große Probleme auch in andere c++-Programme integriert werden könnte. Weitere Informationen hierzu finden sich in dem entsprechenden Abschnitt des Developer's Guide.

Die in entsprechender Form gespeicherten Sounddaten können unter Benutzung der FMPlayer-API bzw. direkt der fmifstream-API gelesen und in tatsächliche Werte umgewandelt werden. Die Ausgabe kann dann in beliebiger Weise, etwa als Klang oder grafisch, erfolgen. Mehr Informationen zu den verwendeten APIs und Programmen können in den entsprechenden Abschnitten des Developer's und dieses Technical Guide gefunden werden.

3.3 fmfile/fmofstream-API

Die einfachste Art der Speicherung (Modus 0, in der Kommunikation mit dem User Modus 1 genannt) eines frequenten Klanges ist die Speicherung sämtlicher Werte, die innerhalb eines Frequenzverlaufes vorkommen. Die daraus resultierenden Sounddateien sind allerdings ziemlich speicherintensiv, weshalb diese Art der Speicherung nur noch aus Kompatibilitätsgründen unterstützt wird.

Der Einlesevorgang verlangt vom Benutzer natürlich nicht die Eingabe jedes einzelnen Wertes. Vielmehr werden bestimmte, frei wählbare x-y-Werte eingegeben und das Programm berechnet die dazwischenliegenden Werte als Gerade mit der aus zwei aufeinanderfolgenden Werten folgenden Steigung.

Ein frequenter Klang kann durch wenige Punkte mit bestimmtem Abstand zum vorhergehenden Wert und Beschreibung der dazwischenliegenden Verbindungslinien beschrieben werden (Modus 1). Dabei genügen vier Bytes (2 Bytes für den Abstand zum vorhergehenden Wert, ein Byte für den Wert selbst und ein Byte für den darauffolgenden Linienstil) um einen kompletten Wert zu speichern und 4 solche Werte um eine komplette Sinuskurve zu beschreiben.

Einem ganz anderen Zweck dient der dritte Speicherungsmodus (Modus 2). Mit ihm kann man die Noten von Musikstücken eingeben um diese zu speichern und später abspielen zu lassen. Die gespeicherten Werte pro Note bestehen daher aus einem 7 Bit großen Wert für die Tonhöhe (mit einem Tonumfang von 120 Halbtönen, also 10 Oktaven), einem 4 Bit großen Wert für den Notenwert und einem 5 Bit großen Wert für das zum Ton gehörige Instrument, einer beim Abspielen frei festlegbaren Kombination aus Klangdatei und Lautstärkenverlauf.

Eine genaue Dokumentation der verschiedenen Dateiformate und der fmofstream-API findet sich im Developer's Guide.

3.4 Berechnen und Abspielen

Die FMPlayer-API übernimmt alle Aufgaben, die für das Berechnen und Abspielen der Soundwerte aus gespeicherten FMS-Sounddateien nötig sind. Die Dokumentation der Programmierschnittstelle ist im Developer's Guide enthalten.

3.4.1 Berechnung der Werte für frequente Klänge

Das Berechnen der Werte ist der zeitaufwendigste Prozess bei der Erzeugung von Klängen. Die Funktion, die dies für jeweils ein einzelnes Soundpaket erledigt, befindet sich unter dem Namen `FMPackage::compute()` in der Datei `fmplayer.cpp`.

Als Erstes begrenzt FMS die Anzahl der berechneten Werte auf die minimale benötigte Anzahl. Diese berechnet sich aus der Samplingrate, also der Zahl der abgespielten Werte pro Sekunde, der Frequenzen der Klänge und der Frequenzen der Kurven, mit denen Lautstärke und Frequenz schwanken. Die Anzahl n der benötigten Werte eines Klanges mit Frequenz f und Samplingrate r wird daher mit folgender Formel berechnet:

$$n = \frac{r}{f} \tag{3.1}$$

Sollten auch Kurven verwendet werden, um die Frequenz und Lautstärke schwanken zu lassen oder Klänge mit anderen Frequenzen gleichzeitig abgespielt werden, so ist der Gesamtwiederholungswert das kleinste gemeinsame Vielfache der Wiederholungswerte der verschiedenen Frequenzen. Dabei muss insbesondere auch darauf geachtet werden, dass das berechnete n nicht die maximal benötigte Anzahl von Werten für eine Dauer t und eine Samplingrate r überschreitet:

$$n \leq t \cdot r \tag{3.2}$$

Sobald im Speicher Platz für die benötigten Werte geschaffen worden ist, können diese berechnet werden. Dies ist kein Problem bei im Modus 0 (jeder Wert in eine Datei geschrieben) gespeicherten Klängen, da hier lediglich der aktuelle Zeitindex hochgezählt und der entsprechende Wert ausgewählt werden muss. Anders verhält es sich mit Modus 1, bei dem nur wenige Werte mit Angaben zu Zeitindex und darauffolgendem Linienstil in einer Datei gespeichert werden. Hier muss mit Hilfe von je nach Linienstil verschiedenen Formeln der aktuelle Wert berechnet werden. Die folgenden Formeln beschreiben diesen Vorgang für alle unterstützten Linienstile. Dabei ist v_0 der dem zu berechnenden Wert vorhergehende und v_1 der folgende Wert, t_0 und t_1 die entsprechenden Zeitindices sowie v_x und t_x der aktuell zu berechnende Wert und sein Zeitindex.

Die Formel für Linienstil 1, eine gerade Linie, lautet:

$$v_x = \frac{v_1 - v_0}{t_1 - t_0} \cdot (t_x - t_0) + v_0 \tag{3.3}$$

$\frac{v_1 - v_0}{t_1 - t_0}$ ist dabei einfach der Steigungsfaktor $\frac{\Delta y}{\Delta x}$. Dieser wird mit dem Abstand zwischen vorhergehendem und aktuellem Wert multipliziert. Zum Ergebnis wird der vorhergehende Wert addiert.

Die folgenden beiden Formeln berechnen die verschiedenen Sinusabschnitte, entweder mit geradem oder gekrümmtem Anfang (jeweils 90°):

$$v_x = v_0 + \sin\left(\frac{t_x - t_0}{t_1 - t_0} \cdot 90 + 0\right) \cdot (v_1 - v_0) \tag{3.4}$$

$$v_x = v_1 + \sin\left(\frac{t_x - t_0}{t_1 - t_0} \cdot 90 + 90\right) \cdot (v_0 - v_1) \tag{3.5}$$

$\frac{t_x - t_0}{t_1 - t_0}$ ist 1, wenn der aktuelle Wert ganz am Ende des Abschnittes steht und 0, wenn er am Anfang steht. Die Gleichungen sollten selbsterklärend sein.

Für die beiden Abschnitte einer Gauss-Kurve (vom x-Wert x_0 bis zur y-Achse bzw. von der y-Achse bis zum x-Wert x_1) sind folgende Formeln nötig:

$$v_x = v_1 + e^{-(x_0 \cdot 1 - \frac{t_x - t_0}{t_1 - t_0})^2} \cdot (v_0 - v_1) \tag{3.6}$$

$$v_x = v_0 + e^{-(x_0 \cdot \frac{t_x - t_0}{t_1 - t_0})^2} \cdot (v_1 - v_0) \tag{3.7}$$

Für die Berechnung von Parabelabschnitten auf der Basis von Angaben des x-y-Startwerts und Endwerts sind drei verschiedene Formeln nötig - je nachdem ob die Parabel symmetrisch ist und ob a 1 oder -1 beträgt. Bei asymmetrischen Parabelabschnitten lautet die Formel wie folgt, wobei x_0 und x_1 die Grenzen der zu berechnenden Abschnitte der Parabel auf der x-Achse bilden:

$$v_x = v_0 + \frac{v_1 - v_0}{x_0^2 - x_1^2} \cdot \left(x_0^2 - \left(x_0 + \frac{t_x - t_0}{t_1 - t_0} \cdot (x_1 - x_0)\right)^2\right) \tag{3.8}$$

Und für symmetrische Parabelabschnitte bei $a = 1$, wobei v_{max} für den Maximalwert und amp für die gewählte Amplitude steht

$$v_x = v_{max} - v_1 + amp \cdot \frac{t_a - t_0}{t_1 - t_0} \cdot \left(\frac{t_x - t_0}{t_1 - t_0} - 1\right) \tag{3.9}$$

beziehungsweise, bei $a = -1$:

$$v_x = v_{max} - v_1 - amp \cdot \frac{t_x - t_0}{t_1 - t_0} \cdot \left(\frac{t_a - t_0}{t_1 - t_0} - 1 \right) \tag{3.10}$$

Sämtliche Formeln sind in der Funktion `fmval()` in `fmval.cpp` enthalten.

Die berechneten Werte müssen natürlich mit den jeweiligen Lautstärkefaktoren (z.B. mit denen einer Hüllkurve) und insbesondere auch mit der relativen Lautstärke des berechneten Klanges zur Gesamtlautstärke aller Klänge multipliziert werden, um Übersteuern zu vermeiden. Alle Lautstärkefaktoren sind dabei Werte zwischen 0 und 1, da die normalerweise maximale Amplitude nicht noch vergrößert werden darf.

Nach Berechnung eines Wertes wird der aktuelle Zeitindex (t_x) weitergezählt, wobei es bei der Ermittlung des hinzugezählten Wertes auf die Frequenz (höhere Freqenz → größerer dazugezählter Wert) ankommt, die natürlich unter Benutzung von schwankenden Frequenzen nicht immer gleich ist.

3.4.2 Berechnung der Werte für Rauschen

Zur Erzeugung von Rauschen benötigt man Zufallsfunktionen. Erreicht wird das z.B. durch zufällige Variation der Frequenz einer Schwingung. Die C-Zufallsfunktion `rand()` gibt einen zufälligen Inte-

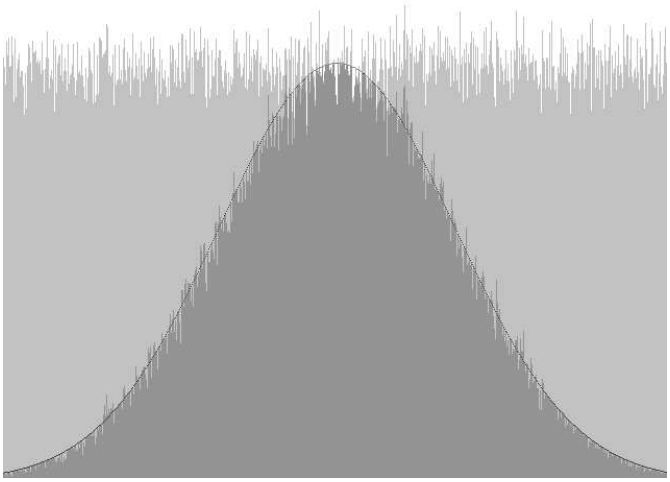


Abbildung 3.2: Zufallsfunktionen im Ereignisdiagramm

gerwert zurück, eine Zahl zwischen -2147483648 und 2147483648, wobei auf lange Sicht jede Zahl so häufig wie die andere ist (hellgraue Balken im Diagramm). Für den Zweck der Rauscherzeugung ist es jedoch wichtig, eine Verteilungskurve (z.B. Gauss-Verteilung) für den Zufall angeben zu können (dunkelgrau). Die Aufgabe der Erzeugung eines solchen Zufallswertes erledigt die Funktion `fmrand()` in `fmval.cpp`. Es wird ein Standart-Zufallswert erzeugt und berechnet, wie häufig dieser aufgrund der angegebenen Verteilung relativ zurückgegeben werden sollte. Diese Wahrscheinlichkeit wird mit einem weiteren Zufallswert verglichen und je nach Ergebnis zurückgegeben. Dabei wird ein Wert mit der Wahrscheinlichkeit 0.1 nur in zehn Prozent der Fälle akzeptiert. Entscheidet sich die Funktion gegen einen Wert, so ruft sie sich rekursiv selbst auf, um einen neuen Wert zu finden.

3.4.3 /dev/dsp

Linux bietet als Möglichkeit zum Abspielen von Klängen `/dev/dsp`, ein sogenanntes Device, im Prinzip eine Datei, auf die durch Schreib- und Lesezugriffe wie direkt auf die Soundkarte, allerdings durch dahinter stehende Kerneltreiber bequemer und portabler, zugegriffen werden kann. Informationen zur Initialisierung und Programmierung von `/dev/dsp` können entsprechenden HowTos [8] entnommen werden.

Aus der für jedes Soundpaket berechneten Reihe von Werten muss beim Abspielen immer der aktuell richtige ausgewählt werden. Dieser wird dann auf `/dev/dsp` geschrieben.

Zusätzliche Schwierigkeiten bietet die Tatsache, dass die Werte eines Soundpakets eventuell mehrfach hintereinander abgespielt werden müssen, sei es vom Benutzer explizit angegeben oder zur Rechenersparnis automatisch bestimmt. Bei Benutzung von zwei Soundkanälen, also im Stereo-Modus, muss darauf geachtet werden, dass immer abwechselnd ein Wert für den linken und ein Wert für den rechten Soundkanal geschrieben werden. All dies erledigt die Funktion `FMPlayer::play()` in `fmplayer.cpp`.

3.4.4 Wave-Dateien

Das Schreiben von Tondaten in eine .wav-Datei bringt generell ein Problem mit sich: das Dateiformat ist ziemlich kompliziert und schlecht dokumentiert, so dass selbst in Suchmaschinen hoch einsortierte Dokumentationen teilweise zumindest schwer verständlich, lücken- oder sogar fehlerhaft sind.

Zuerst muss der sogenannte wav-Header, der Anfang der Wavedatei mit Informationen über Länge, Bitrate, Anzahl der Kanäle u.ä., geschrieben werden. Dann können die rohen Sounddaten einfach als Reihe von Bytes hinzugefügt werden.

Im Developer's Guide zur fmwav-API sind genauere Beschreibungen zum Format und zur Benutzung der API gegeben.

3.5 fmifstream-API

Die fmifstream-API wird intern von FMPlayer genutzt, um FMS-Sounddateien zu öffnen und in verarbeitungsfähigem Format einzulesen. Als einzige Funktion technisch hervorzuheben ist diejenige, welche die Frequenzwerte der Töne bei Dateien im FMS-Midi-Modus berechnet. Da zwölf Halbtonschritte ja eine Verdoppelung der Frequenz bedeuten, muss das Verhältnis v zwischen einem Ton und dem darunterliegenden Halbton folgendermaßen zu berechnen sein:

$$v = \sqrt[12]{2} \tag{3.11}$$

Der n-te Halbton über einem Grundton mit der Frequenz f_0 hat die Frequenz f_1, die man also so berechnet:

$$f_1 = f_0 \cdot v^n \tag{3.12}$$

3.6 andere Elemente

Die anderen Elemente von FMS enthalten technisch nichts Interessantes und müssen deshalb nicht im Technical Guide erwähnt werden.

Kapitel 4

Developer's Guide - Wie FMS programmiert ist

4.1 Vorwort

Dieses Kapitel soll einerseits einen Einblick geben, was der Autor beim Programmieren gelernt hat und andererseits anderen Programmierern eine Möglichkeit geben, mit dem vorhandenen Sourcecode selbst mit FMS verwandte Programme zu schaffen.

FMS ist in c++ programmiert und sollte unter der Betriebssystemplattform Linux mit dem Compiler gcc/g++ 3.2 erfolgreich zu kompilieren sein. Die Einbindung der fms-Headerdateien (im Unterverzeichnis `include/` des Source Trees) ist in anderen c++-Programmen problemlos möglich. Da FMS allerdings nicht als Bibliothek ausgelegt ist, müssen die zu den jeweiligen Headerdateien gehörenden Sourcecode-Dateien mit ins Programm einkompiliert werden.

Dabei ist natürlich auch das Problem der Lizenzierung zu beachten. Der komplette FMS-Quellcode steht unter der GNU General Public License (GPL) [2], weshalb Programme, die auf FMS basieren, ebenfalls unter der GPL veröffentlicht werden müssen. Als auf FMS basierendes Werk gilt ein Programm, das irgendwelche Sourcecodedateien von FMS benutzt oder mit FMS-Bestandteilen verlinkt ist. Um der GPL gerecht zu werden, muss der Sourcecode eines solchen Programmes der Öffentlichkeit zugänglich gemacht werden, was am Besten über die FMS-Project-Page bei Sourceforge [3] möglich ist. Auf Anfrage per e-mail [4] teile ich gerne weitere Informationen dazu mit.

4.2 Headerdokumentation

Die FMS-APIs und sonstigen Klassen und Funktionen können durch Einbinden der jeweiligen Headerdateien auch in anderen c++-Programmen verwendet werden. Um die Benutzung für den Programmierer so einfach wie möglich zu gestalten, gibt es die folgende Dokumentation.

4.2.1 checkint.h

Die enthaltenen Funktionen `checkint()`, `checkfloat()`, in Sonderfällen auch `chooseint()` gewährleisten in der User-Kommunikation über Kommandozeile, dass eingegebene Zahlen tatsächlich zwischen der gewünschten Minimal- und Maximalzahl liegen, bzw. einem der möglichen Eingaben aus einer Reihe von erlaubten Eingaben entsprechen.

Im einfachsten Fall wird eine Zahl in die Integer-Variable x eingelesen und mit folgender Anweisung verifiziert, wobei `min` und `max` den Minimal- bzw. Maximalwert darstellen:

```
x = checkint(x, min, max);
```

Die Funktion wird sich rekursiv immer wieder selbst aufrufen, bis der Benutzer eine richtige Zahl eingegeben hat.

`chooseint()` übernimmt als Parameter den zu überprüfenden Wert, einen Pointer auf ein Array von erlaubten Werten und die Anzahl der Werte in diesem Array.

Bei Benutzung muss der Header `include/checkint.h` eingebunden und `checkint.o` sowie `fm-language_checkint.o` mit ins Programm gelinkt werden.

4.2.2 vtl.h

Der vtl-Header enthält Objekte, die Informationen über FMS-Sounddateien speichern. Um diese zu nutzen, muss nur der Header `include/vtl.h` eingebunden werden, ein Linken mit .o-Dateien ist nicht nötig.

Im Allgemeinen sollte es nicht nötig sein, die fmifstream-Funktionen manuell zu benutzen, sondern es empfiehlt sich vielmehr die Anwendung der komfortableren FMPlayer-API.

VTL

VTL-Objekte (Value-Time-Linetype) enthalten die Informationen über einen Soundwert im Modus 1. Dabei besteht der ganze Wertkomplex aus einem Zeitindex (bei Einlesen der Dateien durch fmifstream: absoluter Zeitindex, sonst relativ zum vorherigen Zeitindex), dem eigentlichen Wert selbst und der Angabe des Linienstils, also des Typs der Verbindungslinie zwischen dem jeweiligen und dem darauffolgenden Wert.

Variable	Bedeutung
`int VTL::time`	Zeitindex
`unsigned char VTL::value`	Wert (Ausschlag)
`unsigned char VTL::linetype`	Linienstil
`int VTL::attr1`	zusätzliche Eigenschaft 1
`int VTL::attr2`	zusätzliche Eigenschaft 2
`int VTL::attr3`	zusätzliche Eigenschaft 3

Der Zeitindex ist dabei beim ersten Wert innerhalb einer Sounddatei 0 und von da an ansteigend, wobei der letzte Wert von Fall zu Fall verschieden sein kann. Der eigentliche Wert ist eine Zahl zwischen 0 und 255, da FMS zur Zeit noch keine andere Byterate als ein Byte pro Wert unterstützt. Die verschiedenen Linienstile sind in der folgenden Tabelle aufgeführt:

Nr.	Bedeutung
0	letzter Wert
1	Gerade
2	Sinuskurvenabschnitt, gerader Anfang
3	Sinuskurvenabschnitt, gekrümmter Anfang
4	Gauss-Kurven-Abschnitt 1, bis zur y-Achse
5	Gauss-Kurven-Abschnitt 2, von der y-Achse an
6	Parabelabschnitt, positives a
7	Parabelabschnitt, negatives a

Für Linienstile 4 - 7 werden offensichtlich noch weitere Parameter benötigt. Diese sind in den `attr`-Variablen gespeichert. Bei den Parabelabschnitten enthalten `attr1` und `attr2` den Start- bzw. Endwert des zu berechnenden Abschnittes auf der x-Achse, `attr3` zusätzlich die Amplitude, die bei symmetrischen Parabelabschnitten benötigt wird. Für Gauss-Kurven muss nur `attr1` als x-Start- bzw. Endwert angegeben werden.

Auf diese `private`-Variablen kann über die Zugriffsfunktionen (get* und set*) zugegriffen werden.

PBS

PBS-Objekte (Pitch-Beats-Style) enthalten die nötigen Informationen über eine Note im FMS-Midi-Modus, die folgendermaßen aus den verschiedenen Variablen besteht:

Variable	Bedeutung
`float PBS::pitch`	Tonhöhe
`float PBS::beats`	Notenwert
`int PBS::style`	Instrumentenstil

Die Tonhöhe wird beim Einlesen berechnet, und gibt das Verhältnis zwischen Kammerton a und Frequenz des jeweiligen Tones an. Der Notenwert beträgt bei einer ganzen Note 1 und bei anderen Notenwerten entsprechende Bruchteile. Der Instrumentenstil schließlich gibt an, welchem beim Abspielen festgelegten Instrument (bestehend aus Oszillogramm + Lautstärkenverlauf) die Note zugeordnet wird. Dabei ist ein Instrument von 0 und 31 möglich.

VTLP

VTLP ist abgeleitet aus VTL und enthält zusätzlich einen `VTLP*` um verkettete Listen zu ermöglichen.

PBSP

Ähnlich verhält es sich mit PBSP, nur dass dieses Objekt nicht von PBS abgeleitet werden konnte, da der Einlesevorgang einige Änderungen am Klassendesign benötigt. Es wäre wünschenswert, dies in Zukunft zu ändern.

4.2.3 fmifstream.h

Das in fmifstream enthaltene Objekt `FMData` dient zum Lesen und Formatieren von FMS-Sounddateien.

Um fmifstream zu benutzen muss der Header `fmifstream.h` eingebunden und `fmifstream.o` mit ins Programm gelinkt werden.

Im Allgemeinen sollte es nicht nötig sein, die fmifstream-Funktionen manuell zu benutzen, sondern es empfiehlt sich vielmehr die Anwendung der komfortableren FMPlayer-API.

FMData

In der Headerdatei `include/fmifstream.h` ist die Klasse FMData enthalten. Sie beinhaltet Funktionen zum Einlesen und Formatieren von in Dateien gespeicherten FMS-Sounddaten.

Normalerweise werden die FMData-Funktionen indirekt über FMPlayer aufgerufen, aber es ist natürlich auch möglich, sie direkt zu verwenden. Allerdings ist dies zeitaufwändiger und fehleranfälliger, weshalb diese Möglichkeit nur in Ausnahmefällen genutzt werden sollte. Vielmehr könnte es sinnvoll sein, die benötigten zusätzlichen Funktionen zu FMPlayer hinzuzufügen. Dazu ist allerdings natürlich auch Kenntnis über fmifstream von Nöten.

Um dem FMData-Objekt a einen Dateinamen „out.fms“ zuzuweisen, diese zu öffnen und auszulesen ist ganz einfach die Anweisung

```
a.init("out.fms");
```

Diese Anweisung kann in einen try-catch-Block eingeschlossen werden, um eventuelle Fehler zu analysieren. Dabei werden als Fehlerindikator Integerwerte verwendet. Zu deren Analyse können die Konstanten `FMD_OPEN_ERR` (Fehler beim Öffnen), `FMD_READ_ERR` (Fehler beim Lesen) und `FMD_FORM_ERR` (Fehler im Dateiformat) verwendet werden.

Nach der erfolgreichen Initialisierung ist die Benutzung der Daten möglich. Dabei ist der Zugriff auf die Bezeichnung, den Speicherungsmodus, die Anzahl der Werte und Bytes pro Wert mit den folgenden Funktionen möglich:

```
char * FMData::getName();
int FMData::getMode();
int FMData::getVlen();
int FMData::getBytes();
```

Die Bezeichnung ist dabei ein zurückgegebener Textstring, der bei der Erstellung der Datei gewählt wurde. Der Modus beschreibt die Art der Speicherung: 0 steht für die Speicherung aller Werte, 1 für die Speicherung einiger Werte mit Angabe der jeweiligen Zeitindices und des Types der Verbindungslinie zwischen ihnen und 2 für die Speicherung im Midi-Modus von FMS. Dabei ist es wichtig, je nach Speicherungsmodus auf die richtigen Werte in fmifstream zuzugreifen. Im Modus 0 befindet sich einfach ein Array von `unsigned char`s hinter dem Pointer `fmifstream::val`. Auf den $n$ten Wert könnte in unserem Beispiel als `a.val[n]`, beginnend mit $n = 0$ zugegriffen werden. Der letzte Wert hat dabei die Ordnungszahl `a.getVlen()-1`. Im Modus 1 dagegen sind die Werte in einer Reihe von VTL-Objekten eingelesen, die sich hinter dem Pointer `fmifstream::vtl` verbergen. Im Modus 2 schließlich sind die einzelnen Tonwerte in je einem PBS-Objekt gespeichert, der Pointer `fmifstream::pbs` zeigt auf das erste. Die VTL- und PBS-Klassen sind im Abschnitt über `vtl.h` beschrieben.

4.2.4 fmlanguage.h

Die fmlanguage-Header und -cpp-Dateien enthalten von den FMS-Programmen verwendete Textstrings. Diese können problemlos in verschiedene Sprachen übersetzt werden, wenn sich ein freiwilliger Übersetzer findet. Vorerst sind sie nur in Deutsch und Englisch verfügbar.

Die jeweiligen Header müssen eingebunden und die dazugehörigen Object Code - Dateien mitgelinkt werden.

4.2.5 fmofstream.h

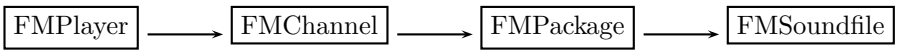
Wenn man selbst Programme zur Erstellung von FMS-Sounddateien schreiben will, ist die Verwendung von fmofstream ratsam. Der Header enthält einige Funktionen, die hier dokumentiert werden sollen.

- `int fmopen(char *)`: Diese Funktion öffnet die Datei mit dem als Parameter übergebenen Dateinamen und gibt die systeminterne Kennziffer für diese Datei zurück. Diese sollte im Programm als Variable gespeichert werden, da sie beim Schreiben der Werte benötigt wird. Es ist zu beachten, das die folgenden Funktionen etwaige schon vorhandene Dateien einfach überschreiben.
- `int mode(char, int)`: Die Funktion schreibt die Kennziffer für den Speicherungsmodus an den Anfang der geöffneten Datei. Es müssen eine char-Variable als Modus-Kennziffer (0-2) und ein Integer als systeminterne Kennziffer für die geöffnete Datei übergeben werden.
- `bool name(char *, int)`: Schreibt die Klangbezeichnung zusätzlich anderer benötigter Parameter in die Datei. Die Übernommenen Parameter sind die Bezeichnung und die systeminterne Kennziffer für die geöffnete Datei.
- `bool values(...)`: Überladene Funktion, die je nach Speicherungsmodus unterschiedliche Variablen als Parameter übernimmt:
  - Modus 0: Die Funktion übernimmt einen Integerwert für die Anzahl der Werte, einen für die Anzahl der Bytes (der 1 sein muss), die systeminterne Kennziffer für die geöffnete Datei und einen `unsigned char *` auf das Wertearray.
  - Modus 1: Die Funktion übernimmt einen Integerwert für die Anzahl der Werte, einen für die Anzahl der Bytes (der 1 sein muss), die systeminterne Kennziffer für die geöffnete Datei und einen `VTLP *` auf das erste Element der verketteten Liste.
  - Modus 2: Die Funktion übernimmt einen Integerwert für die Anzahl der Noten, die systeminterne Kennziffer für die geöffnete Datei und einen `PBSP *` auf das erste Element der verketteten Liste.
- `bool cleanup(int)`: Schließt die Datei mit der übergebenen systeminternen Kennziffer.

Es ist geplant, alle diese Funktionen und Objekte in einer bequemen API, ähnlich der FMPlayer-API, zu vereinen und zusätzliche Fähigkeiten einzubauen, die die Arbeit mit fmofstream weiter erleichtern und weniger fehlerintensiv machen. Um fmofstream zu benutzen, muss `fmofstream.h` eingebunden und das Programm mit `fmofstream.o` verlinkt werden.

4.2.6 fmplayer.h

Ein wirklich praktisches Werkzeug zur Berechnung und zum Abspielen von Werten auf der Basis von FMS-Sounddateien stellt die FMPlayer-API dar. Dabei genügt es, ein FMPlayer-Objekt zu erzeugen und die eingebauten Funktionen zu nutzen. Über dieses Objekt kann man auf alle FMChannel-, FMPackage- und FMSoundfile-Objekte zugreifen. Die Hierarchie der verschiedenen Objekte stellt sich folgendermaßen dar:



Das bedeutet, dass jeder FMPlayer mehrere FMChannel enthalten kann, die ihrerseits mehrere FMPackages enthalten können, die wiederum mehrere FMSoundfiles enthalten können. Dabei ist eine FMSoundfile ein Klang mit Eigenschaften wie z.B. der Frequenz, ein FMPackage ein fertig gemischtes „Soundpaket“ aus verschiedenen Klängen mit Eigenschaften wie z.B. der Dauer, ein FMChannel eine Folge von Soundpaketen die hintereinander abgespielt einen Soundkanal ergeben und der FMPlayer ein Objekt mit ein oder zwei Kanälen und globalen Eigenschaften wie der Samplingrate. Die Reihen der Soundpakete und Sounds sind dabei als verkettete Listen implementiert.

Um die im Folgenden dokumentierten Objekte zu nutzen, muss `fmplayer.h` eingebunden und das Programm mit `fmplayer.o`, `fmwav.o`, `fmval.o` und `fmifstream.o` verlinkt werden.

Die in diesem Kapitel enthaltenen Listen von Funktionen und Variablen erheben keinen Anspruch auf Vollständigkeit. Vielmehr sind weniger wichtige Funktionen, die fast nur API-intern verwendet werden, nicht extra aufgeführt.

Datentypen

Die API verwendet drei eigene Datentypen (enumerations), die zuerst beschrieben werden sollen

- `PlayMode`: Abspielmodus des FMPlayers; kann `DSP` (Sound-Ausgabe), `WAV` (Ausgabe in Wave-Datei) oder `NONE` (keine Ausgabe) sein
- `FileMode`: Verwendung der Sounddatei; kann `Sound` (normaler Klang), `Noise` (Rauschen durch zufällige Haltewerte, die Haltedauer ist nach dem Oszillogramm verteilt), `RandNoise` (wie `Noise`, aber auch Werte sind nach dem Oszillogramm verteilt) oder `Rand` (wie `RandNoise`, Werte werden aber nicht gehalten) sein
- `HFMode`: Modus der Hüllkurve; kann `Envelope` (Ausschlag vom Minimalwert wird moduliert), `AmpMod` (Ausschlag vom Mittelwert wird moduliert) oder `Ring` (Ring-Oszillator) sein.

FMPlayer

Das übergeordnete Objekt, das ein Abspielgerät darstellt, ist der FMPlayer. Die enthaltenen Funktionen und als `public` deklarierten Variablen sind:

- `FMChannel * first`: Pointer auf den ersten (oder einzigen) Soundkanal
- `FMChannel * second`: Pointer auf den zweiten Soundkanal im Stereo-Modus
- `FMChannel * channel`: Pointer auf den aktuell konfigurierten Soundkanal
- `void switchChannel()`: Schaltet auf den anderen Soundkanal um bzw. erzeugt diesen, falls noch nicht vorhanden. Danach können dessen Eigenschaften und die Eigenschaften seiner Soundpakete konfiguriert werden. Die kanalinternen Pointer auf aktuelles Soundpaket und aktuellen Sound des anderen Kanals, sowie die paketinternen Pointer des anderen Kanals bleiben erhalten.
- `void firstChannel()`: Schaltet wieder auf den ersten Kanal um, falls man sich im Programmablauf nicht mehr sicher sein sollte, welcher Kanal gerade konfiguriert wird. Die kanalinternen Pointer auf aktuelles Soundpaket und aktuellen Sound des zweiten Kanals (sofern vorhanden) bleiben erhalten.
- `FMPackage * package`: Pointer auf das aktuell konfigurierte Soundpaket, das sich immer im aktuell konfigurierten Soundpaket befindet

- `void nextPackage(bool=1)`: Schaltet auf das nächste Soundpaket im aktuellen Kanal um bzw. erzeugt dieses, wenn es nicht besteht und nicht 0 als Parameter angegeben wurde. Die Funktion setzt auch den kanalinternen Pointer auf das neue Soundpaket und seinen ersten Sound. Der paketinterne Pointer auf den aktuellen Sound im vorher aktuellen Paket bleibt erhalten.
- `void firstPackage()`: Schaltet zurück auf das erste Soundpaket im aktuellen Kanal.
- `FMSoundfile * sound`: Pointer auf den aktuell konfigurierten Sound, der sich immer im aktuellen Paket des aktuellen Kanals befindet
- `void nextSound()`: Schaltet auf den nächsten Sound im aktuellen Soundpaket um bzw. erzeugt diesen, wenn er nicht besteht. Die Funktion setzt auch den kanalinternen und paketinternen Pointer auf den aktuellen Sound.
- `void firstSound()`: Schaltet zurück auf den ersten Sound im aktuellen Soundpaket des aktuellen Kanals.
- `void resetPlayer()`: Entfernt alle untergeordneten Objekte aus dem Speicher und führt den Konstruktor erneut aus.
- `void setRate(int)` & `int getRate()`: Zugriffsfunktionen für die Samplingrate; diese wird standartmäßig auf 44100 gesetzt, sollte aber, besonders bei grafischer Darstellung der Werte, oftmals besser andere Werte einnehmen.
- `void setPlayMode(PlayMode)` & `PlayMode getPlayMode()`: Zugriffsfunktionen für die Abspielmodus, siehe oben
- `void setWavFile(char *)`: Legt den Dateinamen der Wave-Ausgabedatei fest und setzt gleichzeitig den PlayMode auf WAV.
- `void playInit()`: Initialisiert den Abspielprozess.
- `void playVal()`: Spielt einen einzelnen Wert ab.
- `void playCleanup()`: Beendet den Abspielprozess.
- `void play()`: Fasst die drei Abspielfunktionen zusammen.

FMChannel

Ein FMChannel ist eine Serie von Soundpaketen, die hintereinander abgespielt werden. Folgende Funktionen sind interessant:

- `void setPlay(bool)` & `bool play()` Zugriffsfunktionen auf die Bool-Variable, die bestimmt, ob der Kanal abgespielt werden soll
- `void setRepeat(int)` & `int getRepeat()` Zugriffsfunktionen auf die Anzahl der Wiederholungen des Kanals; -1 bedeutet Endloswiederholung

FMPackage

Ein FMPackage ist ein abspielbereites „Soundpaket“ aus gleichzeitig abgespielten Klängen, das folgende Eigenschaften (Variablen) und Funktionen hat:

- `FMChannel * channel`: Pointer auf den Soundkanal, zu dem das Paket gehört
- `FMPackage * next`: Pointer auf das darauffolgende Soundpaket, um eine verkettete Liste zu ermöglichen; ist `FALSE`, falls es kein nächstes Soundpaket gibt.
- `FMPackage * self`: Pointer auf jenes Soundpaket, von dem das jeweilige Paket nur eine Kopie darstellt; ist `this`, falls das Paket keine Kopie ist.
- `FMHFFile * hvol`: Pointer auf das FMHFFile-Objekt, nach dessen Oszillogramm sich die Gesamtlautstärke des Pakets ändert; ist 0, wenn das für das Paket nicht der Fall ist.

- `void setHVol()`: Erzeugt ein FMHFFile-Objekt aus dem Pointer, wenn das nicht schon geschehen ist. Die Funktion sollte vor jedem Zugriff auf `FMPackage::hvol` benutzt werden, um sicherzustellen, dass `hvol` existiert.
- `FMSoundfile * sound`: Pointer auf den aktuell konfigurierten Sound
- `float * values`: Pointer auf das Array der berechneten Werte
- `void nextSound()` & `void firstSound()`: siehe FMPlayer
- `void setTime(float)` & `float getTime()`: Zugriffsfunktionen auf die Dauer des Soundpaketes (in Sekunden)
- `void setVolume(float)` & `float getVolume()`: Zugriffsfunktionen auf die Gesamtlautstärke; es werden Werte zwischen 0 und 1 angenommen, mit denen alle Soundwerte multipliziert werden.
- `void setTVol(float)` & `float getTVol()`: Zugriffsfunktionen auf die Summe der Lautstärken aller Klänge
- `void setRepeat(float)` & `float getRepeat()`: Zugriffsfunktionen auf die Anzahl der Wiederholungen des Soundpakets
- `void setSelf(int)` & `void setSelf(FMPackage *)`: Setzt fest, von welchem Paket das aktuelle Paket eine Kopie ist. Die überladene Funktion kann einen Integerwert als Nummer des Originals übernehmen, wobei bei negativem Wert im anderen Soundkanal nach dem Original gesucht wird. Anderenfalls ist ein `FMPackage *` auf das Original als Argument möglich.
- `void deleteMe()`: entfernt dieses Soundpaket

Wenn von einer „Kopie eines anderen Soundpakets“ die Rede ist, so bedeutet das nur, dass die Werte nicht vom Paket selbst berechnet werden, sondern `values` einfach mit dem Wertepointer des Originalpakets gleichgesetzt wird. Die einzige Eigenschaft, die dann noch eine Auswirkung hat ist die Anzahl der Wiederholungen, da sich diese direkt beim Abspielen auswirkt.

Auf die Funktionen und Variablen des aktuellen Soundpakets kann über `FMPlayer::package->` zugegriffen werden.

FMFile

FMFile ist eine Basisklasse, von der die Klassen FMSoundfile und FMHFFile abgeleitet sind. Deshalb enthält sie alles, was für die Berechnung eines Wertes eines Oszillogramms aus einer FMS-Datei nötig ist:

- `FMPackage * package`: Pointer zum FMPackage-Objekt, zu dem die Datei gehört
- `FMHFFile * ffreq`: Pointer zum FMHFFile-Objekt, nach dessen Oszillogramm die Frequenz schwankt
- `void setFFreq()`: Erzeugt ein FMHFFile-Objekt aus dem Pointer, wenn das nicht schon geschehen ist. Die Funktion sollte vor jedem Zugriff auf `FMFile::ffreq` benutzt werden, um sicherzustellen, dass `ffreq` existiert.
- `void unsetFFreq()`: entfernt eventuelles ffreq-Objekt
- `FMHFFile * hvol`: Pointer zum FMHFFile-Objekt, nach dessen Oszillogramm die Lautstärke schwankt
- `void setHVol()`: Erzeugt ein FMHFFile-Objekt aus dem Pointer, wenn das nicht schon geschehen ist. Die Funktion sollte vor jedem Zugriff auf `FMFile::hvol` benutzt werden, um sicherzustellen, dass `hvol` existiert.
- `void unsetHVol()`: entfernt eventuelles hvol-Objekt
- `void setFilename(char *` & `char * getFilename()`: Zugriffsfunktionen auf den Dateinamen

- `void setFreq(float)` & `float getFreq()`: Zugriffsfunktionen auf die Frequenz (in Hz)
- `void setVolume(float)` & `float getVolume()`: Zugriffsfunktionen auf die Lautstärke
- `void setFileMode(FileMode)` & `FileMode getFileMode()`: Zugriffsfunktionen auf den `FileMode` (siehe oben)
- `void setMLen(float)` & `float getMLen()`: Zugriffsfunktionen auf die mittlere Anzahl der Halteschritte bei `Noise`-Filemodi
- `void setAtime(float)`: Zugriffsfunktion zum Setzen des aktuellen Zeitindexes
- `void incAtime()`: Funktion, die nach der Berechnung eines Wertes den Zeitindex - je nach Samplingrate und Frequenz - hochzählt bzw. zurücksetzt
- `void openFile()` & `void closeFile()`: Funktionen zum Öffnen und Auslesen bzw. zum Schließen der Datei
- `protected float value(bool=1)`: Gibt den aktuellen Wert der Kurve (von 0 - 255) zurück. Wenn keine 0 als Parameter übergeben wird, wird der Zeitindex weitergezählt. Die Funktion ist `protected` und kann deshalb nur von FMFile-Funktionen oder aus abgeleiteten Klassen (FMSoundfile, FMHFFile) aufgerufen werden.

Auf die FMFile-Funktionen und Variablen des aktuellen Sounds kann über `FMPlayer::sound->` zugegriffen werden.

FMSoundfile

FMSoundfile ist ein von FMFile abgeleitetes Objekt, das für einen Klang steht. Es hat deshalb folgende zusätzliche Eigenschaften:

- `FMMidi * midi`: Pointer zum FMMidi-Objekt, das im Falle einer Datei im Modus 2 zum Einsatz kommt
- `void setMidi()`: Erzeugt ein FMMidi-Objekt aus dem Pointer, wenn das nicht schon geschehen ist. Die Funktion sollte vor jedem Zugriff auf `FMSoundfile::midi` benutzt werden, um sicherzustellen, dass `midi` existiert.
- `FMSoundfile * next`: Pointer auf den darauffolgenden Sound im gleichen Soundpaket, um eine verkettete Liste zu ermöglichen; ist `FALSE`, falls es keinen nächsten Sound gibt.
- `float value(bool=1)`: Wählt die richtige Funktion zur Bestimmung des aktuellen Soundwerts (entweder `FMFile::value()` oder `midi->MidiValue()`) und gibt deren Ergebnis zurück. Außerdem wird die absolute Lautstärke und Hüllkurvenlautstärke der Datei miteinbezogen. Wird `FALSE` als Parameter übergeben, so zählt die Funktion nicht die jeweiligen aktuellen Zeitindices weiter.

Auf die FMSoundfile-Funktionen und Variablen des aktuellen Sounds kann über `FMPlayer::-sound->` zugegriffen werden.

FMHFFile

Immer wenn eine Frequenz oder Lautstärke einem Oszillogramm folgend schwanken soll, ist ein FMHFFile-Objekt die praktikabelste Lösung. Das Objekt ist von FMFile abgeleitet und enthält zusätzlich folgende Variablen und Funktionen:

- `float HFValue(bool=1)`: Die Funktion gibt den aktuellen Wert zwischen dem Minimal- und Maximalwert zurück. Wenn keine 0 als Parameter übergeben wird, wird der Zeitindex weitergezählt.
- `void setMin(float)` & `float getMin()`: Zugriffsfunktionen auf den Minimalwert, der zurückgegeben werden soll. Bei Oszillogrammen, die die Lautstärke bestimmen, so genannten Hüllkurven oder Envelopes, muss dieser Wert kleiner als 1 und mindestens 0 sein.

- `void setMax(float)` & `float getMax()`: Zugriffsfunktionen auf den Maximalwert, der zurückgegeben werden soll. Bei Hüllkurven muss dieser Wert kleiner als 1 und mindestens so groß wie der Minimalwert sein.
- `void setIter(float)` & `float getIter()`: Zugriffsfunktionen auf die Anzahl der Wiederholungen der Hüll- oder Frequenzkurve während einem Durchgang des Soundpakets.
- `void setMinIter(float)` & `float getMinIter()` & `void setMaxIter(float)` & `float getMaxIter()`: Zugriffsfunktionen auf die minimale bzw. maximale Anzahl der Wiederholungen dieser Hüll- oder Frequenzkurve bei Nutzung einer Modulationskurve (`ffreq`) für die Frequenz. item `void setHFMode(HFMode)` & `float getHFMode()`: Zugriffsfunktionen auf den HFMode (siehe oben)

Auf die Funktionen der aktuellen Sound-Hüll- und Frequenzkurve kann über `FMPlayer::sound->hvol` bzw. `FMPlayer::sound->ffreq` zugegriffen werden. Bei der Soundpaket-Hüllkurve ist `FMPlayer::package->hvol` zu verwenden. Natürlich muss dazu das jeweilige FMHFFile-Objekt existieren.

FMMidi

Das FMMidi-Objekt enthält alle besonderen Funktionen, die beim Abspielen einer FMS-Midi-Datei benötigt werden. Als da wären:

- `void initMidi()`: Öffnet alle Dateien, die für das Abspielen benötigt werden.
- `void setMidiI(int,char *)`: Zugriffsfunktion auf die Dateinamen der Midi-Instrumente.
- `void setMidiH(int,char *)`: Zugriffsfunktion auf die Dateinamen der Midi-Instrument-Hüllkurven.
- `void setMidiHMin(int nr, float dhmin)` & `float getMidiHMin(int)`: Zugriffsfunktionen auf die Minimalwerte der Midi-Instrument-Hüllkurven.
- `void setMidiHMax(int nr, float dhmax)` & `float getMidiHMax(int)`: Zugriffsfunktionen auf die Maximalwerte der Midi-Instrument-Hüllkurven.
- `void setMidiHIter(int nr, float dhiter)` & `float getMidiHIter(int)`: Zugriffsfunktionen auf die Anzahl der Wiederholungen der Midi-Instrument-Hüllkurven.
- `void setBps(float)` & `float getBps()`: Zugriffsfunktionen auf die Beats Per Second, die Funktionen werden automatisch aufgerufen.
- `float MidiValue(bool=1)`:Die Funktion gibt den aktuellen Midi-Soundwert zurück. Wenn keine 0 als Parameter übergeben wird, wird der Zeitindex weitergezählt.

Dabei muss immer beachtet werden, dass bei Optionen die auf ein bestimmtes Midi-Instrument zutreffen immer die Nummer dieses Instruments als erster Parameter angegeben werden muss. Dies wurde in obiger Aufzählung nicht einzeln beschrieben.

Es ist empfehlenswert, sich durch `fmplay.cpp` zu arbeiten, um einen Überblick über die Benutzung der API zu bekommen. Viele der beschriebenen Funktionalitäten werden dabei genutzt, und das auf sehr einfache Weise. Die Beispielprogramme (*test.cpp) können weiteren Einblick liefern.

4.2.7 fmwav.h

Die nicht ganz triviale Aufgabe, Rohwerte im Speicher in eine Wave-Datei zu speichern und auch wieder zu lesen, kann mit Hilfe von fmwav gelöst werden. In fmwav sind nur vier Funktionen enthalten: `writeWavHeader()` zum Öffnen der Datei und Schreiben des so genannten Wav-Headers, wozu es als Parameter den Dateinamen, die Anzahl der Werte, die Samplingrate, die Anzahl der Kanäle (0 für Mono, 1 für Stereo) und die Anzahl der Bytes pro Wert benötigt, außerdem `writeWavByte()` zum Schreiben eines einzelnen Bytes, das als Parameter übergeben wird, in die Wave-Datei und `cleanupWav()`, das keine Parameter benötigt zum Schließen der geschriebenen Wave-Datei.

`wavFile readWav(char *)`: Die Funktion liest die angegebene WAV-Datei ein und gibt ein wav-File-Struct zurück, auf dessen Attribute `length`, einem Integerwert, der die Anzahl der Werte angibt und `data` einem `float *` auf das Array der Werte. Dieses Struct muss wieder mit `delete` aus dem Speicher entfernt werden, um memory leaks zu vermeiden.

4.2.8 version.h

Der version-Header enthält nur zwei Informationen: die Versionsnummer der vorliegenden FMS-Version und den Debug-Level. Setzt man den Debug-Level auf 1 oder 2 und rekompiliert, so werden beim Programmaufruf mehr oder weniger ausführliche Debug-Informationen ausgegeben. Standartmäßig steht der Debug-Level auf 0 und es werden deshalb keine Informationen ausgegeben. Um die Informationen über Versionsnummer und Debug-Level zu nutzen, muss nur der Header `include/version.h` eingebunden werden.

4.3 Dateiformat

FMS unterhält drei verschiedene Dateiformate, die folgendermaßen aufgebaut sind:

Modus 0: jeder Wert

Das Format 0 (in der Kommunikation mit dem User Format 1 genannt) ist die einfachste denkbare Möglichkeit zur Speicherund eines Oszillogrammes. Es wird dabei einfach jeder Wert (natürlich mit einem bestimmten, mehr oder weniger groben, Raster) in die Datei geschrieben. Da für diese Art der Speicherung relativ viel Festplattenplatz benötigt wird, besonders wenn das Ergebnis genau sein soll, ist es nur noch in Sonderfällen empfehlenswert, Dateien im Format 0 zu speichern. Das komplette Format baut sich byteweise wie folgt auf, wobei Werte in eckigen Klammern für beispielhafte Stellen stehen:

Stelle	Bedeutung	Beispiel
0	Modus, beim Modus 1 natürlich immer 1	1
1	Anzahl der Buchstaben der Bezeichnung	5
2[-6]	Bezeichnung	Sinus
[7]	Anzahl der gespeicherten Werte, 65536er-Stelle	0
[8]	Anzahl der gespeicherten Werte, 256er-Stelle	0
[9]	Anzahl der gespeicherten Werte, 1er-Stelle	5
[10]	Anzahl der Bytes pro Wert	1

Die komplette Anzahl der Werte setzt sich also aus drei Stellen eines 256er-Systems zusammen. Bisher wird keine andere Bytezahl pro Wert als 1 unterstützt.

Modus 1: einzelne Werte, verbunden mit Linien verschiedener Art

Das Format 1 (in der Kommunikation mit dem User Format 2 genannt) ist in Hinsicht auf Speicherplatz und Wiedergabe von Kurven sehr viel besser geeignet. Es werden nur wenige Werte gespeichert, zusammen mit ihrem Zeitindex und dem darauffolgenden Linienstil. Da die eigentlichen Ausgabewerte zur Laufzeit berechnet werden, benötigt das Abspielen mehr Prozessorleistung, allerdings ist die Platzersparnis und die bessere Kurvenwiedergabe das wert. Das Format setzt sich byteweise wie folgt zusammen:

Stelle	Bedeutung	Beispiel
0	Modus, beim Modus 1 natürlich immer 1	1
1	Anzahl der Buchstaben der Bezeichnung	5
2[-6]	Bezeichnung	Sinus
[7]	Anzahl der gespeicherten Werte, 65536er-Stelle	0
[8]	Anzahl der gespeicherten Werte, 256er-Stelle	0
[9]	Anzahl der gespeicherten Werte, 1er-Stelle	5
[10]	Anzahl der Bytes pro Wert	1
[11-35]	Werte	siehe unten

Die Anzahl der Bytes pro Wert gibt nur die Bytes für den eigentlichen Wert, nicht aber die für Zeitindex und Linienstil an. Dabei setzt sich ein Wertekomplex meist aus vier Bytes wie folgt zusammen:

Stelle	Bedeutung	Beispiel
0	Zeitindex-Offset seit letztem Wert, 256er-Stelle	0
1	Zeitindex-Offset seit letztem Wert, 1er-Stelle	10
2	Wert	255
3	darauffolgender Linienstil	siehe unten

Bei Linienstil 6 und 7 werden die zusätzlichen Parameter einfach hinter den oben beschriebenen Wertekomplex in die Datei geschrieben.

Die 8 Linienstile sind:

Nr.	Bedeutung
1	gerade Linie
2	Sinuskurvenabschnitt, linearer Anfang
3	Sinuskurvenabschnitt, gekrümmter Anfang
4	Gauss-Kurven-Abschnitt 1, bis zur y-Achse
5	Gauss-Kurven-Abschnitt 2, von der y-Achse an
6	Parabelabschnitt, positives a
7	Parabelabschnitt, negatives a
0	letzter Wert

Durch diese Linienstile werden die angegebenen Punkte verbunden.

Modus 2: FMS-Midi

Modus 2 besteht aus einer Aneinanderreihung von Informationen über Einzeltöne, also am ehesten einer klassischen Midi-Datei. Binär ist das Format folgendermaßen aufgebaut:

Stelle	Bedeutung	Beispiel
0	Modus, beim Modus 2 natürlich immer 2	2
1	Anzahl der Buchstaben der Bezeichnung	14
2[-15]	Bezeichnung	Menuett - Bach
[16]	Anzahl der gespeicherten Töne, 256er-Stelle	0
[17]	Anzahl der gespeicherten Töne, 1er-Stelle	83
[18-182]	Töne	siehe unten

Ein Ton wird in nur 2 Bytes gespeichert. Diese Komprimierung benötigt „Byte-Sharing“ zwischen verschiedenen Attributen, weshalb das Format sehr kompliziert in einer Tabelle darzustellen ist. Interessierte können die Bitkonfigurationen `fmofstream.cpp` entnehmen.

Kapitel 5

Anhang

5.1 Glossar

- **Amplitudenmodulation**: Variation der Schwingungsamplitude
- **API**: Application Programming Interface, Objekte und Funktionen zum Einbau in Programme
- **Frequenzmodulation**: Variation der Schwingungsfrequenz
- **Midi-Mapping**: Ausgabe von Mididaten ohne Unterstützung eines klassischen Synthesizers
- **Oszillogramm**: Visualisierung von Schwingungsvorgängen, bei einem Sinuston z.B. der Graph der Sinusfunktion von 0°bis 360°
- **Ringmodulation**: Spezialform der Amplitudenmodulation
- **Samplingrate**: Anzahl der pro Sekunde über die Soundkarte ausgegebenen Werte

5.2 Dank

Ich danke

- Herrn Deffner für die Anregungen und Unterstützung
- den Teilnehmern des Schülerseminars im Studio für Elektronische Musik und Computermusik für ihre Ideen
- allen gedultigen Testern und Nutzen von FMS
- allen Programmierern, auf deren Arbeit ich FMS aufgebaut habe
- Ihnen, weil Sie sich das hier bis zum Ende durchgelesen haben

Literaturverzeichnis

[1] GNU Emacs, der Editor, der sogar einen eingebauten Psychiater hat: `www.gnu.org/software/emacs/emacs.html`

[2] GNU General Public License: `www.gnu.org/copyleft/gpl.html`

[3] FMS Download-Adresse: `www.sourceforge.net/projects/fmsynth`

[4] e-mail-Adresse des Autors: `daniel_gruen@web.de`

[5] SDL: `www.libsdl.org`

[6] SDL_gfx: `www.ferzkopp.net/Software/SDL_gfx-2.0`

[7] Qt/Trolltech: `www.trolltech.com`

[8] z.B. das FMS-DSP-HowTo: `www.fmsynth.sourceforge.net/dsp.dvi`