Ernst-Abbe-Fachhochschule Jena
Hochschule für angewandte Wissenschaften

Ernst-Abbe-Fachhochschule Jena
Fachbereich: Elektrotechnik und Informationstechnik

Beleg

Digital Signal Prozessors

MIDI-Synthesizer

eingereicht von: M. Bukovsky (633058)
M. Ludwig (633319)
Studiengang: Bachelor of Engineering – technische Informatik

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Bmp	-	Beats per minute
DDS	-	Direkte Digitale Synthese
DSP	-	Digital Signal Processors
MThd	-	Header-Chunk einer Midi Datei
MTrk	-	Track-Chunk einer Midi Datei
PPQN	-	Pulses per Quarter Note (Länge einer $^1/_4$ Note)

Formelzeichenverzeichnis

Formelzeichen		Bezeichnung	Einheit
c	-	Konstante	
f	-	Frequenz	Hertz
n	-	Notennummer	

1 Einleitung

In diesem Beleg soll erläutert werden, wie begleitend zu der Lehrveranstaltung Signal-Processors ein Projekt realisiert wurde, um Mittels eines DSP-Evaluation-Boards eine MIDI-Datei wiederzugeben. Die Entwicklung des MIDI- Synthesizers wurde in 2 Themenkomplexe unterteilt. Zum Einen musste das MIDI-File zur Verarbeitung ausgelesen werden. Desweiteren mussten die Töne generiert werden. Um Töne zu erzeugen bedarf es genauerer Kenntnisse über deren Eigenschaften und den Methoden der Synthese.
Dieses Dokument soll desweitern einen kurzen Einblick in die Akustik, die Verfahren der künstlichen Tonerzeugung und deren Umsetzung mit dem DSP geben. Es wird gezeigt welche Herangehensweisen prinzipiell möglich sind, wo ihre Grenzen liegen und welche einfach zu realisieren sind.

2 MIDI

2.1 Midi Aufbau

Eine MIDI Datei besteht aus dem Header-Chunk *MThd* und aus einem oder mehreren Track-Chunks *MTrk*. Die Daten des MIDI-Files sind im BigEndian-Format gespeichert und erschwerend kommt hinzu, dass oft im 7-Bit Format gearbeitet wird und dass manche Längenangaben variable sind.

Für dieses Projekt wurde folgende Notation verwendet:

Dateityp	Größe in Bytes	Wertebereich
Byte ($int8_t$)	1	-128 …127
Byte ($uint8_t$)	1	0 … 255
Short ($int16_t$)	2	-32768 … 32767
Short ($uint16_t$)	2	0 … 65536
Int ($int32_t$)	4	-2147483648 … 2147483647
Int ($uint32_t$)	4	0 … 4294967296

Tabelle 1: verwendete Datentypen

Bei der Verarbeitung der Midi Files wurden die Daten in Strukturen untergebracht, da sie die Möglichkeit geben, verschiedene Komponenten zusammen zu fassen. Diese strukturierten Daten ermöglichen einen großen Vorteil bei der Erzeugung von den Signalen. So ist der Header Chunk und der TrackChunk in jeweils eine Struktur zusammengefasst, welche wiederrum Typen von dem MidiTrack sind.

2.2 Header Chunk

Im Header-Chunk sind grundlegende Informationen über das Musikstück hinterlegt. Es beinhaltet die nicht nur die Anzahl der Tracks, die in der Datei gespeichert sind, sondern auch wie diese abgespielt werden und wie die Zeitangeben zu interpretieren sind.

Name	Datentyp	Deklaration des Datentyps		Beschreibung
header	header	Name	Typ	
		data[]	$uint8_t$	„MThd“ bzw. 0x4D546864
		header	$int32_t$	Zeiger
size	size	Name	Typ	
		data[]	$uint8_t$	Größe des Headers
		header	$int32_t$	
field	field	Name	Typ	
		data[]	$uint8_t$	Daten des Headers
		field[]	$int16_t$	TimeDevision

Tabelle 2: Struktur des Header Chunk

Mit dem Header kann man eine Midi Datei identifizieren.
Ein Header ist immer mit folgendem Schema aufgebaut:

$$4D\ 54\ 68\ 64\ 00\ 00\ 00\ 06\ ff\ ff\ nn\ nn\ dd\ dd$$

Das ASCII-Äquivalent der ersten 4 Bytes (**$4D$ 54 68 64**) sind „MThd“. Dies ist der Beginn des Headers und leitet somit das Auslesen der ersten Informationen ein. Nach dieser Zeichenkette folgt die Größe des Headers. Diese ist immer 4-Byte (**00 00 00 06**) lang. Anhand des Inhaltes dieses Wertes ist klar zu sehen, dass der „.mid-Header“ genau 6 Bytes groß ist.

Die Hexadezimalwerte ff stehen für das Format. Es gibt an, wie die einzelnen Tracks abgespielt werden. Es gibt 3 Formate:
0 - eine Spur
1 - mehrere synchrone Spuren. Dabei werden alle Tracks gleichzeitig abgespielt
2 - mehrere asynchrone Spuren. Hier werden die Tracks abhängig von ihrer Sequenznummer abgespielt.
In der Regel spricht jeder Track nur einen Kanal an

nn: nn gibt die Anzahl der Tracks in einer MIDI-Datei an, die nach dem Header-Chunk folgen. Natürlich ist diese Angabe abhängig von dem zuvor beschriebenen Format. Bei Dem Format 0 sollte die Anzahl der Tracks 1 betragen. Bei mehreren Spuren größer 1.

dd: dd beschreibt wie die Zeitangaben richtig zu interpretieren sind. Dieses wird auch TimeDevision genannt. Es gibt 2 Formate, welches am an anhand des ersten Bits unterscheiden kann. Ist dieses nicht gesetzt, wird im PPQN- andernfalls im SMPTE Format gearbeitet.

Name	Aufbau	Beschreibung
PPQN	0XXXXXX XXXXXXX	Pulses per Quarter Note Wie lang eine $^1/_4$Note ist.
SMPTE	1XXXXXX YYYYYYY	Ticks / Frame / Second XXXXXX Frame / Second 24, 25, 29 oder 30 YYYYYYY Ticks / Frame

Tabelle 3: Formate der TimeDevision

Durch diese Angabe ist es möglich die genauen Zeitabstände der Track-Events genau zu interpretieren.

2.3 Track Chunk

Nach dem Header Chunk folgen die Track Chunks. Dabei hat jeder Track Chunk einen eigenen Header. Dieses besitzt wie der Header Chunk zur Erkennung ein 4-Bytes langes ASCII-Äquivalent - „MTrk“. Gefolgt wird es von einem 4-Byte großen Wert, welches die Länge des Tracks - ohne Track-Kopf - beinhaltet.
Mit diesen Informationen können nun die darauf folgenden Track-Events ausgelesen werden. Vor jedem Ereignis steht eine Deltazeit. Diese Zeit steht im festen Verhältnis zu den zuvor im Header Chunk ausgelesenen TimeDevision. Die Deltatime beschreibt, wann das Ereignis ausgelöst werden soll. Dieses Verfahren hat den Vorteil, dass keine kleinen Zahlen benötigt werden.
Jedes MIDI-Kommando hat 2 Teile. Dabei wird das Kommando in ein linkes und ein rechtes Nybble aufgeteilt (1 Nybble = 4 Bit). Im Linken Nybble steht das Kommando und im Rechten der dazugehörige Kanal.

In diesem Projekt werden die Hauptkommando's umgesetzt. Dazu gehört das NOTE_ON und NOTE_OFF Kommando. Die Kommandos für das Abklingen der Note, der Tonhöhenänderung und der Kanalausklang wurden vorerst nicht berücksichtigt.

Name	Datentyp	Deklaration des Datentyps		Beschreibung
Chunk	chunk	Name	Typ	
		Data[]	$uint8_t$	„MTrk“ bzw. 0x4D546864
		header	$int32_t$	Zeiger
Size	size	Name	Typ	
		Data[]	$uint8_t$	Größe des Headers
		header	$int32_t$	
Counter	$uint16_t$			Zähler bei der Sinuserzeugung
delayCounter	$uint16_t$			Zähler bis zum nächsten Ereignis
commnad	$int16_t$			Platzhalter für das aktuelle Event

Tabelle 4: Struktur der Track-Chunk

2.4 Midi Tracks

Um die Verarbeitung der Signalerzeugung zu verbessern wurde die Tracks in eine „Midi-Track“ Struktur beschrieben. Dabei wurde nur auf die wesentlichen Informationen für die Tonerzeugung Wert gelegt, um eine

Name	Datentyp	Deklaration des Datentyps		Beschreibung
numTracks	$int8_t$			Track Nummer
tracks	trackChunk	Name	Typ	
		Chunk	chunk	Repräsentiert einen Chunk
		Size	size	Größe des Chunk
		Counter	$uint16_t$	Zähler für Signalerzeugung
		delayCounter	$uint16_t$	Zähler, nächstes Ereignis
		commnad	$int16_t$	Platzhalter, aktuelle Event
commands	command	Name	Typ	
		Command	$uint16_t$	Kommando (NoteON | NoteOFF)
		Channel	$uint16_t$	Stereokanal (Links | Rechts)
		deltaTime	$uint16_t$	Delta-Zeit zum Nächsten Event
		noteNumber	$uint16_t$	Abzuspielende Notennummer
		noteVelocity	$uint16_t$	Abspielgeschwindigkeit der Note

Tabelle 5: Struktur MidiTrack

Zur Verdeutlichung der Struktur MidiTrack hier eine bildliche und beispielhafte Darstellung.

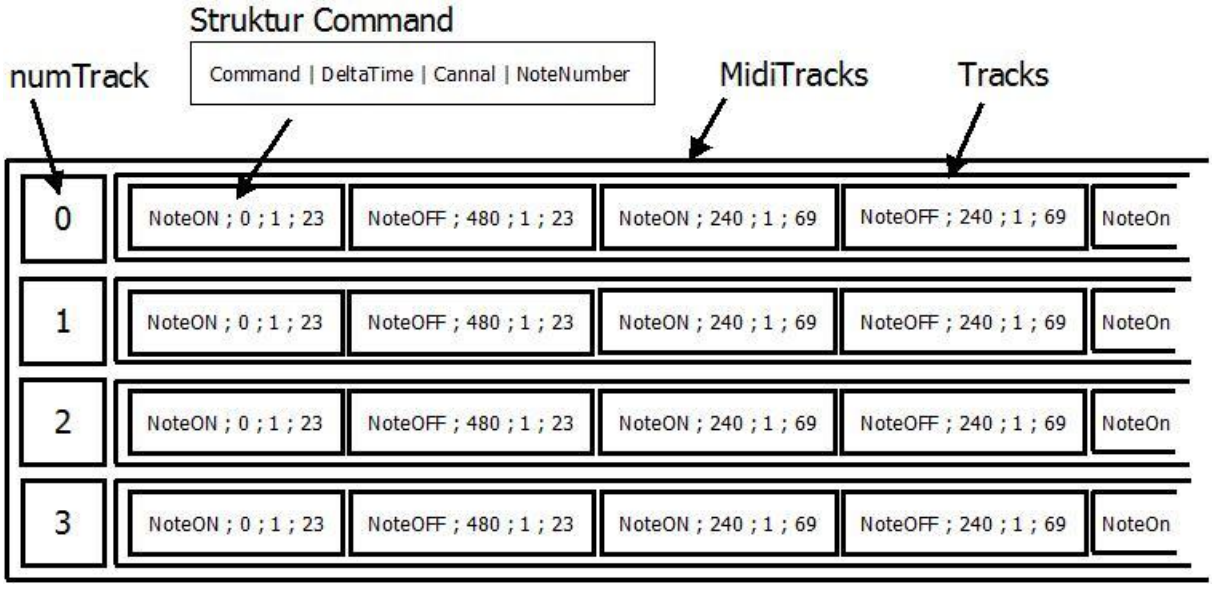

Abbildung 1: Struktur Midi-Tracks

3 Musiktheorie

Um nun ein Ton aus der Midi Datei zu generieren, bedarf es ein wenig Hintergrundwissen in Bezug auf die Musiktheorie, sowie der Signalerzeugung.
Die Definition des Tons kann variieren. Für die meisten Menschen ist ein Ton eine sinusförmige Schalldruckwelle - Sinuston. Schall ist physikalisch gesehen eine Welle, die sich meist longitudinal durch ein Medium, mit einer für diese charakteristische Geschwindigkeit, ausbreitet. Töne verstehen sich somit als Welle.

Unterscheiden lassen sich Töne in ihrer Tonhöhe. Die Amplitude der Sinuskurve gibt die Laustärke, die Periodendauer die Tonhöhe/Tonfrequenz in Hz an. Wenn man mehrere Töne übereinander lagert, ergibt die ein Geräusch. Sind diese Töne harmonisch zueinander ist das Geräusch ein Klang. Somit kann man sagen, das ein Schlagzeug ein Geräusch erzeugt und ein Klavier einen Klang.

Klänge haben keinen oder mehrere Obertöne, einen Grundton und keinen oder mehrere Untertöne. Dabei nehmen Obertöne ein Vielfaches und Untertöne ein Bruchteil der Frequenz des Grundtons ein.

Eine Oktave besteht aus 7 Stammtönen(C, D, E, F, G, A, H) und 5 Halbtönen (Cis, Dis, Fis, Gis, Ais).

3.1 Vom Ton zum Klang

Ein weiteres wichtiges Kriterium ist die Wellenform, welche maßgeblich die Klangfarbe eines Tons bestimmt. Hiermit ist nicht die Form einer Schwingung als solche gemeint, wie z.B. die Rechteck, Dreieck, oder Sinusschwingung, viel mehr setzen sich Töne in der Regel aus mehreren Einzelschwingungen, den sogenannten Obertönen, zusammen. Dadurch entsteht die für ein bestimmtes Instrument charakteristische Wellenform.

Bei Saiteninstrumenten zum Beispiel besteht die Obertonreihe (Gesamtheit der Obertöne) aus, zumindest annäherungsweise, ganzzahligen Vielfachen der Grundschwingung.

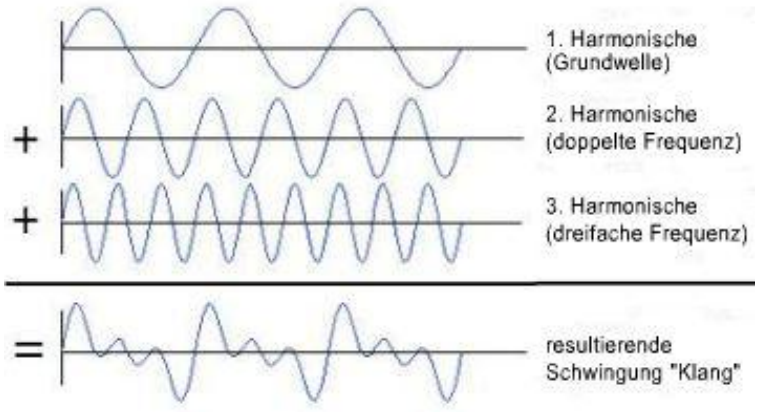

Abbildung 2: Grundwelle + 2 Oberwellen = Faltung [Quelle 1]

Gemeinsam bleibt bei Ton und Klang die Periodizität des Musters, das sich jeweils nach der Dauer einer Grundschwingung wiederholt.

Um eine Melodie zu erzeugen, müssen mehrere Klänge nacheinander und auch parallel abgespielt werden.

3.2 Notenlänge

Weiterhin ist es notwendig sich mit den Notenlängen auseinander zu setzen. In der elektronischen Musik hat sich als Einheit für die Geschwindigkeit eines Stücks die sogenannten "beats per minute" eingebürgert. „Beats per minute“ ist ein Maß für die Anzahl der Viertelnoten pro Minute. Somit lässt sich die Notenlänge leicht anhand folgender Formel errechnen.

$$f = \frac{1}{bpm}$$

Für 240 bpm ergibt dies:

- $\frac{1}{1} Note = 1000ms$
- $\frac{1}{2} Note = 500ms$
- $\frac{1}{4} Note = 250ms$
- $\frac{1}{8} Note = 125ms$
- ...

4 Künstliche Signalerzeugung

Im Folgenden werden kurz die Verfahren zur künstlichen Klangerzeugung vorgestellt. Bis hin zur Auswahl des Verfahrens, welches bei der Implementierung mit dem DSP-Board zum Einsatz kommt.

4.1 Additive Synthese

Die Additive Synthese basiert auf den Überlegungen des französischen Mathematikers Fourier. Nach diesem lässt sich jeder denkbare Klang aus einer Mischung von verschiedenen Sinusschwingungen darstellen, wie schon bei den Obertonreihen gesehen. Da komplexere Klänge aber zum Teil aus sehr vielen verschiedenen Frequenzen bestehen, ist dieses Syntheseverfahren für unsere Zwecke nur bedingt einzusetzen.

Die Additive Synthese endet allerdings bei der Erzeugung von Akkorden bzw. bei mehrstimmigen Liedern Verwendung. Somit ist es kein adäquates Mittel bei der Signalerzeugung einer MIDI-Datei.

4.2 Subtraktive Synthese

Der Additiven Synthese gegen über steht die Subtraktive Synthese. Der ursprüngliche Klang wird durch einen Oszillator erzeugt und ist reich an Obertönen. Das "klangliche Rohmaterial" wird durch Verwendung von Filtern oder Hüllkurven um bestimmte Frequenzanteile vermindert.
Dieses Verfahren soll nur der Vollständigkeit halber vorgestellt werden, und findet im Weiteren keine Anwendung.

4.3 Direkte Digitale Synthese

Die Direkte Digitale Synthese (DDS) ist ein Verfahren welches häufig in Funktionsgeneratoren Anwendung endet. Sie ist einfach zu realisieren und hat nur geringe Ansprüche an die verwendete Hardware.

Bei der Direkten Digitalen Synthese wird eine Periode eines vorher aufgezeichneten Signals im Speicher gehalten. Um eine hohe Qualität zu erreichen, werden hierbei möglichst viele Werte mit einer möglichst hohen Amplitudenauflösung gesampelt.

Die Werte dieses Referenz-Signals werden nun in gleichen Zeitabständen wieder ausgegeben. Um verschiedene Frequenzen zu erzeugen überspringt man einzelne Stützstellen des Signals. Um z.B. eine Verdopplung der Grundfrequenz zu erreichen wird bei der Ausgabe jede zweite Stützstelle ausgelassen. Durch Auslassen vereinzelter Stützwerte erreicht man eine nur geringfügig höhere Frequenz.

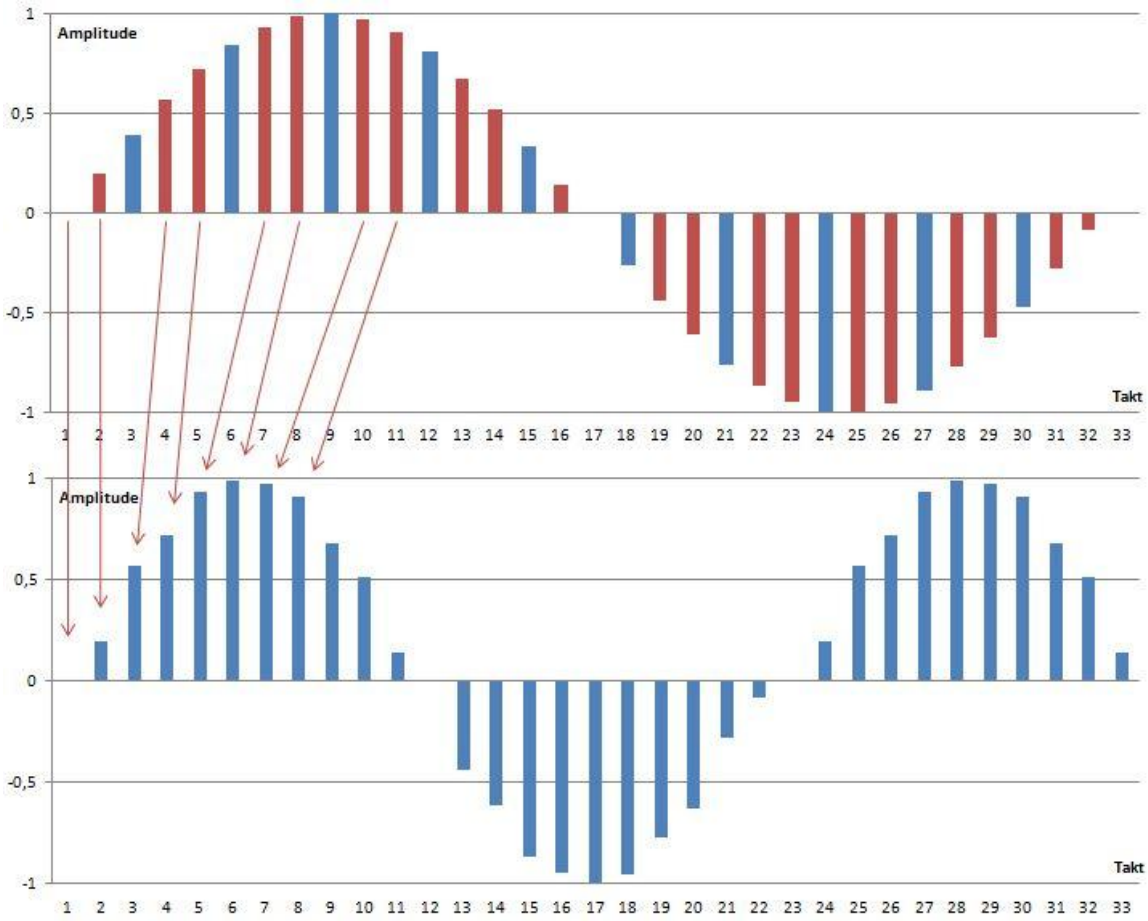

Abbildung 3: Direkte Digitale Synthese

Ein Nachteil der DDS ist die Tatsache, dass bei zu großem Abstand zwischen Ausgabe-Frequenz und der des Referenz-Signals, die Wellenform sehr stark verfälscht wird. Oder anders, um einen im Vergleich zum Ausgangssignal hohen Ton zu erzeugen müssen sehr viele Samplewerte ausgelassen werden, welche aber unter Umständen wichtig für die Klangcharakteristik des Tons sind.

Genau dieses Verfahren kommt bei diesem Projekt für die Erzeugung verschiedener Frequenzen zum Einsatz.
Hierbei füllt eine Funktion eine Lookup Table (LUT) mit dem Sinus Signal, welches 44000 Samples eines einzelnen Sinus beinhaltet.

4.4 Von der Notennummer zum Ton

Der Kammerton A ist ein Sinuston mit der Frequenz von 440 Hz und besitzt die Notennummer 69.
Dieser Ton wird verwendet um z.B. eine Gitarre zu stimmen. Dazu nutzt man eine Stimmgabel die nach dem Anschlagen eine Frequenz von 440 Hz erzeugt.

Oktave	C	C#	D	D#	E	F	F#	G	G#	A	A#	H
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

Tabelle 6: Töne in Notennummern

Das MIDI Protokoll umfasst 128 Noten. Dazu muss die jeweilige Nummer in einen Ton umgewandelt werden. Das Verhältnis zweier benachbarter Halbtöne beträgt beim 12-Tonsystem $\sqrt[12]{2}$.

4.5 Berechnung der Frequenz

Ausgehend davon, dass der Ton A mit seiner Frequenz bekannt ist, lassen sich alle anderen Frequenzen berechnen. $[n]$ ist dabei die gegebene Notennummer.

$$f = c \cdot 2^{\frac{n}{12}} Hz$$

$$400Hz = c \cdot 2^{\frac{69}{12}} Hz$$

$$c = \frac{400Hz}{2^{\frac{23}{4}}} \approx \underline{8{,}1757989156437}$$

Nun kann man die Konstante *c* dafür benutzen, um die anderen Frequenzen der einzelnen Töne auszurechnen.
Beispielsweise wird der Ton F mit der Nummer 40 folgendermaßen berechnet.

$$f = 8{,}1757989156437 \cdot 2^{\frac{40}{12}} Hz = \underline{82{,}406Hz}$$

Die Formel besagt auch, dass sich die Frequenz eines Tons in der nächst höheren Oktave verdoppelt und in der nächst kleineren Oktave halbiert. So hat A3 in der 4ten Oktave 880 Hz und in der 2ten Oktave nur 220 Hz.

Um die Ressourcen des DSP's zu sparen, werden die Notennummern vor Beginn der Wiedergabe in entsprechende Frequenzen umgewandelt. Diese Aufgabe übernimmt die Funktion `convert_note_number (midiTracks* midiSample)`.

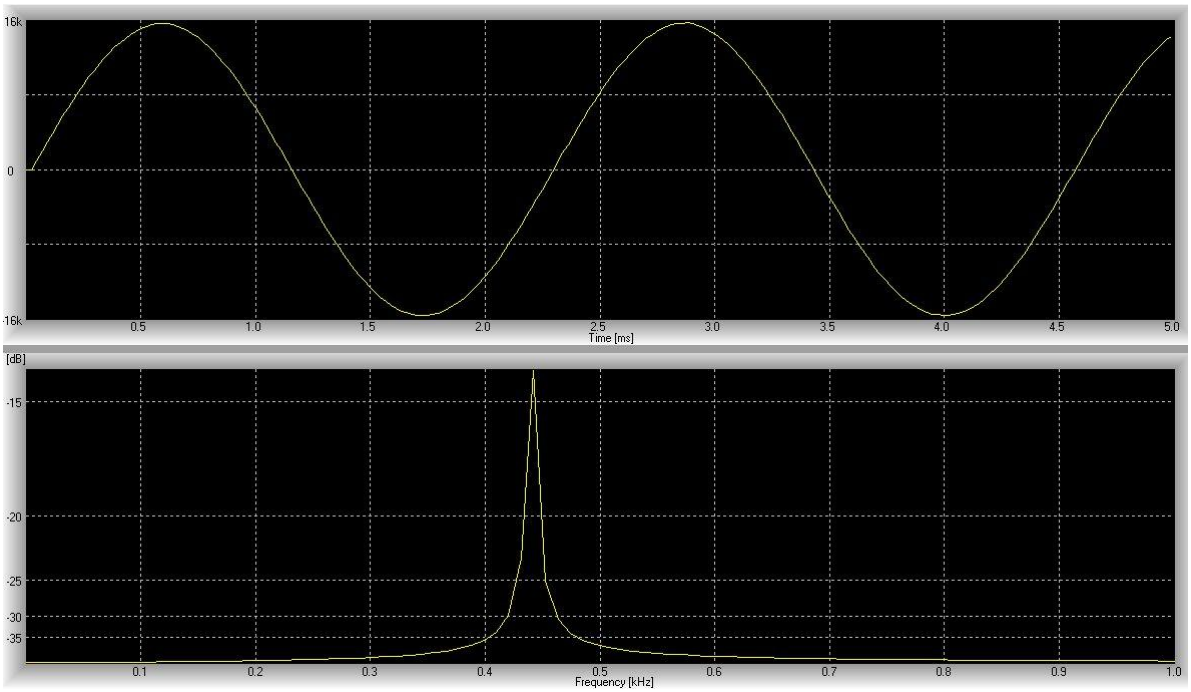

Abbildung 4: Sinusschwingung mit 440Hz

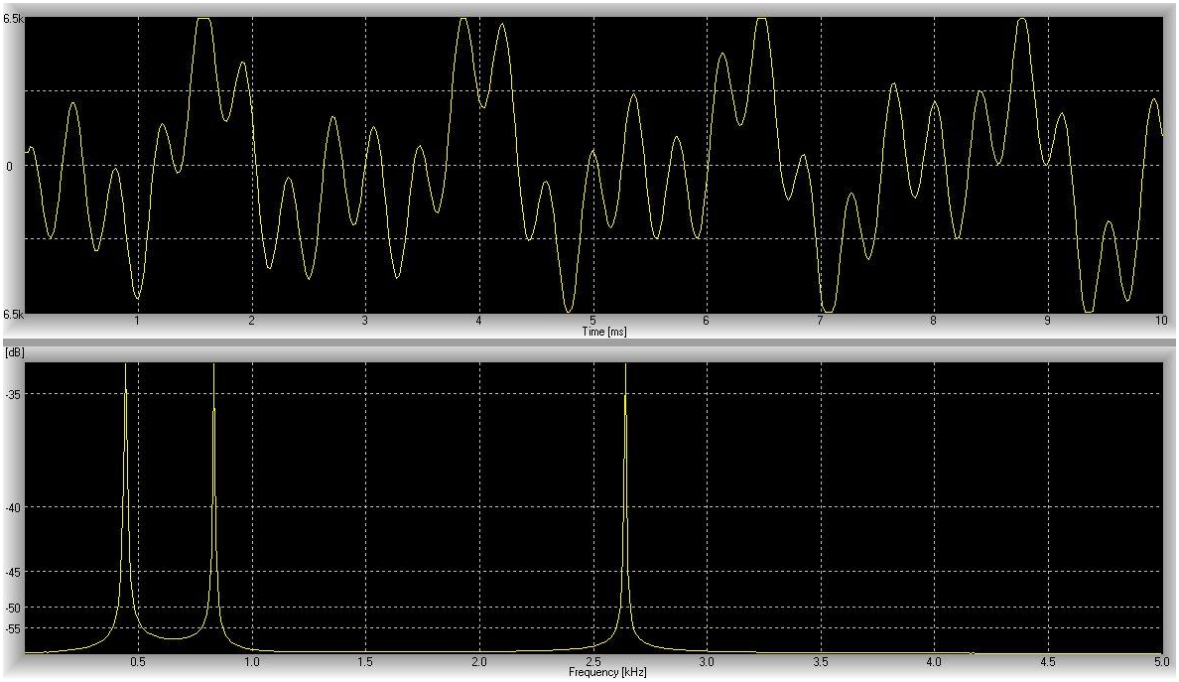

Abbildung 5: Klang mit 3 verschiedenen Frequenzen

In Abbildung 4 und Abbildung 5 ist das Oszillogramm und Frequenzspektrum eines reinen Tons und eines Klangs dargestellt. Diese Signale wurden für Testzwecke mit dem DSP Board erzeugt. Beide Male wird der Stimmton a' gespielt. Wenn man mit einem Musikinstrument einen "Ton" spielt, entstehen automatisch zum Grundton auch die Obertöne - man erzeugt ungewollt einen "Klang". Nur elektronische Instrumente können reine Töne erzeugen. Bei einem Klang sind noch mehrere Frequenzen vertreten, nämlich

die des Grundtons und die der Obertöne. Im Spektrum sieht man sofort, welche Frequenzen dies sind und dass sie jeweils ein ganzzahliges Vielfaches der Grundfrequenz sind.
Aus dem Oszillogramm lässt sich etwas über die Reinheit eines Klanges und über die Wiederholung von gewissen Klangteilen erfahren. Je mehr die Kurve einem Sinus gleicht, desto reiner ist der Klang.

4.6 Timing

Notenwerte geben die relative Tondauer an. Wie lange sie tatsächlich gespielt wird,
wird vom Dirigenten, bzw. im Falle der künstlichen Signalerzeugung, von der Samplerate bestimmt.
Da dieser Wert nicht direkt aus dem Midi Dokument ausgelesen werden kann, muss die Samplerate aus der DeltaTime (dt. Deltazeit) ausgelesen werden.
Diese Umrechnung übernimmt die Funktion `convert_delta_time (midiTracks* midiSample)` für die gesamten Track-Events.
Dabei wird zunächst das Verhältnis zum Deltawert für die Länge einer Viertelnote(PPQN) berechnet. Dies wird dann mit dem Tempowert multipliziert und nochmal durch 1000 dividiert, da der Wert in Mykrosekunden angegeben ist.

Wird zum Beispiel aus der Midi Datei ein PPQN-Wert von 480 ausgelesen, ergibt sich folgende Tabelle.

Notenwert	Ticks	Notenwert	Ticks
$^1/_1$	1920	$^3/_2$	2880
$^1/_2$	960	$^3/_4$	1440
$^1/_4$	480	$^3/_8$	720
$^1/_8$	240	$^3/_{16}$	360
$^1/_{16}$	120	$^3/_{32}$	180
$^1/_{32}$	60	$^3/_{64}$	90
$^1/_{64}$	30	$^3/_{128}$	45
$^1/_{128}$	15	$^3/_{256}$	--
$^1/_{256}$	--	$^3/_{512}$	--
$^1/_{512}$	--	$^3/_{1024}$	--

Tabelle 7: PPQN Wertetabelle

4.7 Interrupt

Mit den berechneten Notenwerten und den Kommandos NOTE_ON bzw. NOTE_OFF aus der Midi Datei, können nun die einzelnen Tracks wiedergegeben werden.
Diese Aufgabe übernimmt eine interruptgesteuerte Funktion, welches den kleinsten vorhandenen Notenwert bestimmt. In dem Beispiel aus Tabelle 7 ist dies die $1/128$ Note. Die anderen Notenwerte sind immer ein Vielfaches von diesem Wert.

Die Interrupt-Routine schaut nun jede $1/128$ Note in die Tackliste und prüft, ob ein neues Event ansteht und führt dieses aus. Ist dies nicht der Fall, wird die Deltazeit verringert.

5 Optimierung

5.1 MIDI Dokument auslesen

TODO: …

6 Fazit und Ausblick

Da der MIDI Synthesizer, mit dem DSP-Board, nur auf die wesentlichsten Funktionalitäten für dieses Projekt beschränkt wurde, könnte für die Zukunft die Erweiterung der Funktionalität noch anstehen. Diesbezüglich könnten die Meta-Events ausgelesen werden, um Beispielsweise Informationen über die Instrumente oder Lyrics zu bekommen.

Desweiteren kann bei der künstlichen Signalerzeugung ein anders Verfahren eingesetzt werden, um Klänge für ein Instrument zu erzeugen.
Hierbei handelt es sich um die WaveTable Synthese. Sie funktioniert nach dem gleichen Prinzip wie die DDS. Auch hier wird ein Referenz-Signal im Speicher abgelegt, im Unterschied zur DDS wird im Regelfall nicht nur eine Periode, sondern der gesamte Ton aufgezeichnet, und dieser mit variabler Wiedergabegeschwindigkeit unter Berücksichtigung jeden Stützwertes ausgegeben. Ein Sample beinhaltet somit schon einen Lautstärkeverlauf und es muss dementsprechend keine Hüllkurve mehr generiert werden. Dadurch dass bei der Wavetable-Synthese jeder gespeicherte Wert ausgegeben wird, bleibt die Klangcharakteristik auch bei höheren Frequenzen weitestgehend erhalten.

7 Quellenverzeichnis

[1] *Was unterscheidet Ton, Klang und Geräusch?*
http://www.laermorama.ch/m1_akustik/tonklang_w.html

[2] *Standard MIDI File Format Dustin Caldwell*
http://www.larsricshter-online.de/lmids/midformat.htm

8 Anhang

8.1 Quelltexte

```
/**
 * DSP-Bibliotheken
 */
#include "dsk6713_aic23.h"     //support file for codec,DSK
#include "dsk6713_led.h"
#include "dsk6713_dip.h"
#include "dsk6713config.h"
#include "dsk6713.h"

/**
 * C-Bibliotheken
 */
#include "stdio.h"
#include "stdint.h"
#include "stdlib.h"
#include "stdbool.h"
#include "notes.h"
#include "convert.h"

/**
 * Globale Variabelen
 */
DSK6713_AIC23_CodecHandle hCodec;
uint32_t  fs = DSK6713_AIC23_FREQ_44KHZ;  // setzen der Sampling-Rate (8, 16,
44, 96 kHz)

midiTracks  midiSample;
trackChunk  midiTrack[16];
command  midiCommands[16*200];
command* midiCommandsPointer[16];
short   counter = 0;
short   index = 0;
float   volume_DSP = 1.0f;   // external DSP-Board Volume
uint8_t  tmp, i = 0;



/**
 * Externe Funktion
 */
extern far void vectors();

/**
 * Interrupt Service Routine
 */
interrupt void c_int11(void){
 static uint8_t  t_timeDevision=0;
 uint8_t    t_index = 0;


 if(t_timeDevision>15){
  while (t_index < midiSample.numTracks) {
```

```
    if (midiSample.tracks[t_index].delayCounter == 0) {
     if( midiSample.tracks[t_index].commandCounter <
midiSample.tracks[t_index].size.size-1){
      midiSample.tracks[t_index].commandCounter++;
     }
     if( midiSample.tracks[t_index].commandCounter+1 <
midiSample.tracks[t_index].size.size){
      midiSample.tracks[t_index].delayCounter =
midiSample.commands[t_index][midiSample.tracks[t_index].commandCounter+1].delta
Time;
     }
    }
    else{
     midiSample.tracks[t_index].delayCounter--;
    }

    t_index++;
   }
   t_timeDevision=0;
  }
  t_timeDevision++;

  return;         //return from interrupt
}

/*
 *  Accessory functions
 */
void interupt_config() {

  int *p;
  //--- Set up needed to for interrupts ---
  CSR = CSR & 0xFFFFFFFFFE; //disable interrupts without CSL
  CSR &= ~0x1; //disable interrupts without CSL

  hAIC23_handle=DSK6713_AIC23_openCodec(0, &config); // handle(pointer) to codec
  DSK6713_AIC23_setFreq(hAIC23_handle, fs); // set sample rate
  MCBSP_config(DSK6713_AIC23_DATAHANDLE,&AIC23CfgData); // interface 32 bits to
AIC23
  MCBSP_start(DSK6713_AIC23_DATAHANDLE, MCBSP_XMIT_START | MCBSP_RCV_START |
  MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220); // start data channel

  IRQ_setVecs(vectors);         //point to the IRQ vector table
  IRQ_map(CODECEventId, 11); //map McBSP1 Xmit to INT11
  p=(int*)(MUXH); //adresse von MUXH - ohne Typecasting funktionierts nicht...
  //set MUXH bits 9:5 (belonging to int11) to 01110 (specifying that it belongs
to Xint1
  *p = *p & 0xFFFFFC1F;
  *p = *p | 0x1C0;

  //IRQ_reset(CODECEventId); //reset codec INT 11
  ICR = 0x800; // clear interrupt flag at bit 11
  IER &= ~0x800; //disable interrupt 11

  //IRQ_globalEnable(); //globally enable interrupts
  CSR |= 0x1;  //set Control-and-Status-Register (CSR) bit 0 to 1

  //IRQ_nmiEnable(); //enable NMI interrupt
  IER |=0x2;
```

```
 //IRQ_enable(CODECEventId); //enable CODEC eventXmit INT11
 IER |= 0x800; //enable interrupt 11

 MCBSP_write(DSK6713_AIC23_DATAHANDLE,0); //start McBSP interrupt
}

int main(int argc, char **argv) {

 union {uint32_t combo;short channel[2];} AIC23_data;

 midiSample.tracks = midiTrack;
 uint8_t t_index=0;

 while(t_index<16){
  midiCommandsPointer[t_index]=&midiCommands[200*t_index];
  t_index++;
 }
 midiSample.commands = midiCommandsPointer;

 create_lut_sinus();  // fill the Lookup Table with the Sinus-Signal
 fill_toene(&midiSample);
 convert_delta_time(&midiSample);
 convert_note_number(&midiSample); // Umwandlung der

 DSK6713_init();    // call BSL to init DSK-EMIF,PLL)
 interupt_config(); // configuration the Interupts
 DSK6713_LED_init();  // init LED from BSL

 while (1) {
  //read next value from codec (poll until value available)
  while (!DSK6713_AIC23_read(hCodec, &AIC23_data.combo));
  AIC23_data.combo=0;
  i=0;
  while (i < midiSample.numTracks) {
   uint16_t t_data = sinusLUT[midiSample.tracks[i].counter] >> 4;
   AIC23_data.channel[LEFT] += t_data;
   AIC23_data.channel[RIGHT] += t_data;
   // wenn ... und Note ON
   if(midiSample.tracks[i].commandCounter!=-1 &&
midiSample.commands[i][midiSample.tracks[i].commandCounter].command ==
NOTE_ON){
    // signal processing

    midiSample.tracks[i].counter +=
midiSample.commands[i][midiSample.tracks[i].commandCounter].noteNumber;
    if (midiSample.tracks[i].counter > SAMPLE_RATE-1)
     midiSample.tracks[i].counter -= SAMPLE_RATE;
   }
   i++;
  }
  DSK6713_AIC23_write(hCodec, AIC23_data.combo);
 }
 //return EXIT_SUCCESS;
}
```

Quelltext 1: Main-Routine

```c
#ifndef NOTES_H_
#define NOTES_H_

#include "stdint.h"
#include "definesMidi.h"

/**
 * Software defines
 */
#define MUXH                0x019C0000      // Addresse des MUXH
#define PI                  3.14159265359f  // Zahl PI
#define VOLUME              1               // max. Lautstaerke
#define QUARTER_NOTE_TIME   0.5             // Laenge einer 1/4 Note in Sekunden
#define SAMPLE_RATE         44000           // Samplerate

/**
 * deklaration der externen Variabelen
 */
extern float volume_DSP;            // Lautstärke der DSP Boards
extern short sinusLUT[SAMPLE_RATE]; // Lockuptable

/**
 * Funktion zur Umrechnung aller DeltaTime eines MIDI in eine Samplerate
 * @param midiSample
 */
void convert_delta_time (midiTracks* midiSample);

/**
 * Funktion zur Umrechnung aller Notennummern eines MIDI in Frequenzen
 * @param midiSample
 */
void convert_note_number (midiTracks* midiSample);

/**
 * Wandelt Ausgelesene Notenummer in eine Frequenz um.
 * @param num = Notenwert
 * @return (int) Frequenzwert
 */
int create_note (short num);

/**
 * Füllt Lookup Tabelle mit dem Sinus-Signal
 */
void create_lut_sinus(void);
```

Quelltext 2: Header der Signalaufbereitung (notes.h)

```c
#include "notes.h"
#include "math.h"

short sinusLUT[SAMPLE_RATE];     //lockup table



/**
 * Wandelt Ausgelesene Notenummer in eine Frequenz um.
 *
 * Okt||                      Notennummer
 * #  ||   C   | C#  | D   | D#  | E   | F   | F#  | G   | G#  | A   | A#  | H
 * ----------------------------------------------------------------------------
 *  0 ||   0   |  1  |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 | 11
 *  1 ||  12   | 13  |  14 |  15 |  16 |  17 |  18 |  19 |  20 |  21 |  22 | 23
 *  2 ||  24   | 25  |  26 |  27 |  28 |  29 |  30 |  31 |  32 |  33 |  34 | 35
 *  3 ||  36   | 37  |  38 |  39 |  40 |  41 |  42 |  43 |  44 |  45 |  46 | 47
 *  4 ||  48   | 49  |  50 |  51 |  52 |  53 |  54 |  55 |  56 |  57 |  58 | 59
 *  5 ||  60   | 61  |  62 |  63 |  64 |  65 |  66 |  67 |  68 |  69 |  70 | 71
 *  6 ||  72   | 73  |  74 |  75 |  76 |  77 |  78 |  79 |  80 |  81 |  82 | 83
 *  7 ||  84   | 85  |  86 |  87 |  88 |  89 |  90 |  91 |  92 |  93 |  94 | 95
 *  8 ||  96   | 97  |  98 |  99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107
 *  9 ||  108  | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119
 * 10 ||  120  | 121 | 122 | 123 | 124 | 125 | 126 | 127 |
 *
 * @param num = Notenwert (Siehe Tabelle)
 * @return (int) Frequenzwert
 */
int create_note (short num) {
 float div;
 float c = 8.175798916f;  // Konstante
 /* Berechnung der Frequenz
  *   f= c * 2^(num/12)
  */
 div=(float)(num)/12.0f;
 return (int)(c * powf(2.0f, div));
}

/**
 * Füllt Lookup Tabelle mit dem Sinus-Signal
 */
void create_lut_sinus(void) {
 double x = 0;
 uint16_t i = 0;

 for (i = 0; i < SAMPLE_RATE; ++i) {
  // sinus sin(2 * pi * f * t)
  sinusLUT[i] = (short)(0X7FFF*sin(x));
  x = 2*PI*i/SAMPLE_RATE;
 }
}

/**
 * Umwandlung aller DeltaTime des Midi in ein adäquate Samplerate
 * @param midiSample
 */
void convert_delta_time (midiTracks* midiSample) {
```

```
 uint16_t time_dev;
 uint8_t track_index=0;

 // durchläuft jeden Track
 while (track_index < midiSample->numTracks) {

  time_dev = midiSample->tracks[track_index].chunk.chunk;
  uint8_t event_index=0;
  // durchläuft in jedem Track die Events
  while (event_index < midiSample->tracks[track_index].size.size) {

   /**
    * wandelt die DeltaTime eines Events in eine Samplerate um.
    * TODO: schiften mit >> 4 bewirkt das max. 16s ein erneutes Event
aufgerufen werden muss.
    */
   midiSample->commands[track_index][event_index].deltaTime = (uint32_t)(
(midiSample->commands[track_index][event_index].deltaTime * QUARTER_NOTE_TIME /
time_dev) * SAMPLE_RATE ) >> 4;
   event_index++;// zum nächsten Event
  }
  track_index++;// zum nächsten Track
 }
}

/**
 * Umwandlung aller Notennummern des Midi in Frequenzen
 * @param midiSample
 */
void convert_note_number (midiTracks* midiSample) {

 uint8_t track_index=0;
 // durchläuft jeden Track
 while (track_index < midiSample->numTracks) {

  uint8_t event_index=0;
  // durchläuft in jedem Track die Events
  while (event_index < midiSample->tracks[track_index].size.size) {

   // Umwandlung der Notennummer eines Events in eine Frequenz
   midiSample->commands[track_index][event_index].noteNumber =
create_note((short)midiSample->commands[track_index][event_index].noteNumber);
   event_index++;// zum nächsten Event
  }
  track_index++;// zum nächsten Track
 }
}
```

Quelltext 3: Signalaufbereitung - notes.c

Quelltext 4: Definierung MIDI Kommands (definesMidi.h)

Quelltext 5: Header der MIDI Konvertierung (convert.h)

Quelltext 6: MIDI Konvertierung (convert.c)