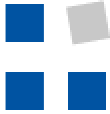


HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

Semesterarbeit Wintersemester 2003

Digitale Signalverarbeitung

Multieffektgerät

Autoren:

Thomas Kuch
Martin Stierli

Betreuer:
Prof. Dr. Guido Martin Schuster

13. Februar 2004

Hochschule für Technik Rapperswil
Oberseestrasse 10
Postfach 1475
CH-8640 Rapperswil

Abteilung *Elektrotechnik*

Vertiefungsfach *Digitale Signalverarbeitung*

Semesterarbeit Wintersemester 2003

Multieffektgerät

© 2004

Abstract

In dieser Semesterarbeit wurde auf der Basis eines Texas-Instruments-Entwicklungsboardes für digitale Signalverarbeitung ein Gitarren-Multieffektgerät implementiert, welches über einzelne Regler am PC gesteuert werden kann. Durch die verschiedenen Effekte erhält ein Gitarrist eine zusätzliche interessante Möglichkeit, seine Kreativität auszuleben.

Das analoge Gitarrensignal wird von einem Analog/Digital-Wandler auf dem Board eingelesen und abgetastet. Der digitale Signalprozessor verarbeitet die Daten je nach Auswahl der Effekte und gibt sie via Digital/Analog-Wandler wieder analog aus. Von einem PC aus können die Effekte ein- und ausgeschaltet, sowie gewisse Parameter verändert werden.

Diese Dokumentation beschreibt ausführlich, wie die verschiedenen Effekte prinzipiell funktionieren und wie sie auf dem verwendeten Board implementiert wurden.

Inhaltsverzeichnis

1 Einleitung **6**
1.1 Projektidee 6
1.2 Zusammenfassung der Realisierung 6
1.3 Ausblick 6
1.4 Struktur dieses Berichts 6
2 Grundlagen **6**
2.1 Chorus 6
2.2 Delay 6
2.3 Tremolo 6
2.4 Overdrive, Distortion 6
2.5 Reverb 6
2.6 Wah 6
2.7 Phaser (Phasenschieber) 6
2.8 Looper 6
3 Experimente in MatLab **6**
3.1 Schwierigkeiten 6
3.2 MatLab-Programme 6
3.2.1 Delay 6
3.2.2 Tremolo 6
3.2.3 Chorus 6
3.2.4 Distortion 6
3.2.5 Phaser 6
3.2.6 Wah 6
4 Hardware **6**
4.1 DSP Starter Kit 6
4.2 Audio Daughter Card 6
4.3 Erforderliche Samplefrequenz 6
4.4 Geplante Zusatzhardware 6
5 Datentransfer **6**
5.1 McBSP 6
5.2 EDMA 6
6 Code Composer Studio **6**
6.1 RTDX 6
6.2 Parametersteuerung mit GEL 6
6.2.1 Was ist GEL 6
6.2.2 Verwendung eines GEL-Files 6
6.2.3 GEL Slider 6
6.3 Profiler 6
7 Implementierung auf DSP-Board **6**
7.1 Sample- oder blockweises Einlesen der Audiodaten 6
7.2 Hilfsfunktionen für Effekte 6
7.2.1 inHistory 6
7.2.2 delHistory 6
7.2.3 Kontinuierlicher Raised Cosine 6
7.3 Implementierte Effekte 6
7.3.1 Delay 6
7.3.2 Tremolo 6

7.3.3 Chorus 6
7.3.4 Reverb 6
7.3.5 Distortion 6
7.3.6 Wah 6
7.4 Stabilitätsproblem 6
8 Benutzung 6
8.1 Voraussetzungen 6
8.2 Inbetriebnahme 6
9 Schlussfolgerungen 6
9.1 Rückblick auf Arbeitsverlauf 6
9.2 Persönlicher Rückblick 6
9.3 Dank 6
10 Quellen 6
11 Anhang 6
11.1 Abbildungsverzeichnis 6
11.2 Zusätzliche Dokumente/Medien 6

1 Einleitung

1.1 Projektidee

Auf dem Markt sind grundsätzlich zwei Arten von Gitarreneffektgeräten erhältlich: teure, aber gut zu bedienende Einzeleffektpedale und aufwändig zu bedienende Multieffektgeräte. Beide werden zwischen Elektrogitarre und Verstärker geschaltet und verändern den ursprünglichen Gitarrenklang unterschiedlich. Der Musiker hat dadurch mehr kreative Möglichkeiten, da er nicht nur durch die Art seines Spiels, sondern auch noch durch gezielten Einsatz von Effekten seinen Klang beeinflussen kann.
Das Ziel dieser Arbeit war es, den Komfort der Einzelpedale mit der günstigeren Bauweise eines Multieffektgerätes zu kombinieren, indem zwar mehrere Effekte in einem Gehäuse untergebracht und von einem digitalen Signalprozessor generiert werden, diese jedoch einzeln über eigene Parameter-Regler und nicht über komplizierte Menuführung steuerbar sein sollen.

1.2 Zusammenfassung der Realisierung

Dem eigentlichen Programmieren des digitalen Signalprozessors ging einerseits eine intensive Phase der Effekt-Recherche, sowie einige Zeit des Experimentierens in MatLab voraus. Es stellte sich heraus, dass die Materie „Klang" ziemlich schwierig zu handhaben ist, da sie nicht präzis beschrieben werden kann. Was klingt gut, was klingt schlecht? Nichts desto trotz ist es uns gelungen, einige Effekte zu implementieren, die teilweise erstaunlich ansprechend klingen.

Das Ein- und Ausschalten, sowie das Einstellen gewisser Effektparameter realisierten wir aus Zeitgründen nicht mit einer Zusatzhardware, sondern mittels einzelner Sliders am PC, die die Entwicklungsumgebung zur Verfügung stellt. Gerne hätten wir ein Visual C++ Programm geschrieben, welches in einem Fenster alle Schalter und Regler vereint und das DSP-Board ansteuert. Leider mussten wir nach intensiven Nachforschungen feststellen, dass es nicht möglich ist, mit dem Board zu kommunizieren, ohne die Entwicklungsumgebung im Hintergrund laufen zu lassen.

Betreffend Stabilität existieren leider noch ein paar Schwierigkeiten, die einerseits auf die Entwicklungsumgebung, andererseits beim Zusammenschalten vieler Effekte auf die Rechenintensität des Effekt-Codes zurück zu führen sind.

1.3 Ausblick

Würde man an unserem Projekt weiter arbeiten, müsste man den bestehenden Code genau analysieren und auf Ausführungsgeschwindigkeit optimieren, damit er stabiler wird.
Die von uns einst geplante Zusatzhardware, die es erst ermöglicht, das Effektgerät als „Stand Alone-Anwendung" zu nutzen, wäre ebenfalls eine sinnvolle Erweiterung, genauso wie weitere gut klingende Effekte.

1.4 Struktur dieses Berichts

Zu Beginn möchten wir dem Leser einen Überblick über die verschiedenen Effekte verschaffen, die sich in der Welt der Gitarristen über die Jahre hinweg etablieren konnten. Unmittelbar danach zeigen wir auf, wie wir einige dieser Effekte in MatLab programmiert haben.
In Abschnitt 4 und 5 beschreiben wir relativ detailliert unsere verwendete Hardware und wie wir die digitalisierten Daten auf dem Board managen. Ebenso ist da unsere zu Beginn geplante Zusatzhardware erklärt.
Bevor wir in Abschnitt 7 unsere C-Implementierungen der einzelnen Effekte erläutern, beschreiben wir in Abschnitt 6 kurz die für die Programmierung verwendete Entwicklungsumgebung, das Code Composer Studio.

Wer es besonders eilig hat und die Effekte ausprobieren möchte, findet eine detaillierte Schritt-für-Schritt-Anleitung zur Ausführung in Abschnitt 8.

Für ein optimales Verständnis dieser Dokumentation ist es von Vorteil, wenn der Leser über die Grundlagen der C-Programmierung und der Elektrotechnik verfügt, sowie über die Funktionsweise eines Prozessorsystems und digitaler Filter Bescheid weiss.

2 Grundlagen

Zu Projektbeginn musste zuerst recherchiert werden, welche Effekte existieren und wie sie funktionieren. Danach mussten wir uns entscheiden, welche wir in Angriff nehmen wollten. Nachfolgend werden alle Effekte erläutert, welche wir ursprünglich für eine Implementierung in Betracht gezogen haben.

2.1 Chorus

Der Chorus-Effekt vervielfacht einen einzelnen Ton so, als ob mehrere Gitarren gleichzeitig das gleiche spielen würden. Da in der Realität niemals mehrere Musiker absolut unisono spielen können, ergibt sich eine ganz kleine zeitliche Differenz zwischen den Tönen und man empfindet einen volleren Klang, analog zu einem Chor, bei dem ja auch mehrere Sänger das gleiche singen. Zudem ist die Tonhöhe auch nicht absolut identisch, wenn die Instrumente auch noch so gut gestimmt wurden.
Der klassische Chorus verzögert das Signal um ca. 20 bis 30ms und mischt es dem Originalsignal hinzu. Je mehr verschieden verzögerte Signale hinzugefügt werden, desto voller klingt der Sound (desto „grösser wird der Chor“). Die etwas modernere Realisierung des Chorus-Effektes geschieht nicht mit fixen, sondern mit variablen Delay-Zeiten. So ändert auch die Tonhöhe leicht, was wie oben erläutert den Effekt von mehreren Spielern noch realistischer macht. Die Art und Weise, wie die ändernde Verzögerung realisiert wird, hat natürlich einen Einfluss auf den Klang. Analog werden die Verzögerungsglieder via langsame Oszillatoren (LFO) gesteuert. Der Klang ist je nach LFO-Form (Sinus, Dreieck,...) etwas anders.

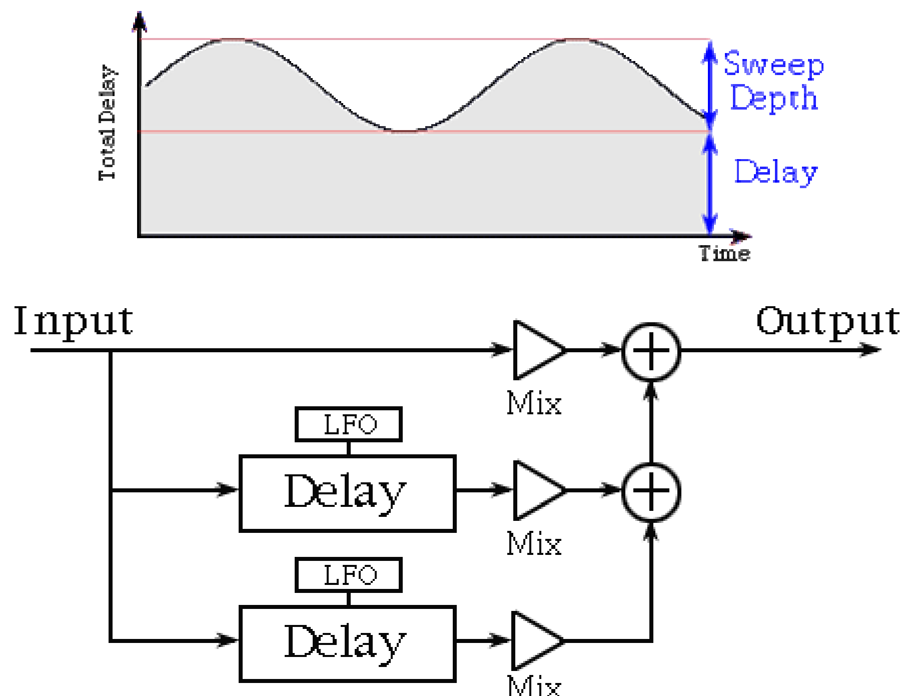


Abbildung 2-1, Chorus-Prinzip, Quelle [1]

Digital ist der Chorus mit kreisförmigen Puffern zu realisieren, aus denen unterschiedlich schnell ausgelesen wird. Wenn das Signal langsamer ausgelesen werden soll, müssen allerdings Werte interpoliert werden, was unter Umständen zu Rauschen führen kann.

Parameter:
Delaytiefe, Änderungsgeschwindigkeit, Anzahl Stimmen, Effektlevel

2.2 Delay

Der Delay fügt dem Eingangssignal ein bzw. mehrere verzögerte Ebenbilder des Signales hinzu, wobei die Lautstärke abnimmt.

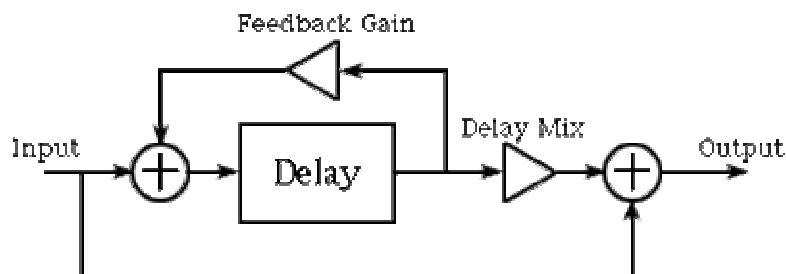


Abbildung 2-2, Delay-Prinzip, Quelle [1]

Digital wird der Delay über kreisförmige Puffer realisiert, d.h. das Eingangssignal wird ständig gespeichert und um die gewünschte Zeit verzögert dem Eingang teilweise wieder beigemischt und wieder abgespeichert usw.

Parameter:
Verzögerungszeit, Abklinggeschwindigkeit, Effektlevel

2.3 Tremolo

Tremolo ist ein sehr einfacher, aber teilweise nützlicher und angenehmer Effekt. Er verändert periodisch die Ausgangslautstärke. Durch das schnelle Ändern der Lautstärke erscheint der Klang harmonisch „ratternd". Der klassische Tremolo-Effekt ändert die Lautstärke natürlich sinusförmig. Andere Signalformen ergeben ebenfalls interessante Klangcharakteristiken.

Parameter:
Änderungsgeschwindigkeit, Änderungstiefe, Signalform

2.4 Overdrive, Distortion

Overdrive und Distortion waren ursprünglich die Wirkungen der Röhren und Transistoren in den Verstärkern, die zu weit aufgedreht in Sättigung gerieten. Mit Overdrive wird in der Regel ein sanftes Beschneiden des Signals gemeint (soft clipping), was einen dezent angezerrten Klang ergibt, während Distortion ein hartes Abschneiden des Signales bedeutet (hard clipping), das dann auch ziemlich hart klingt.

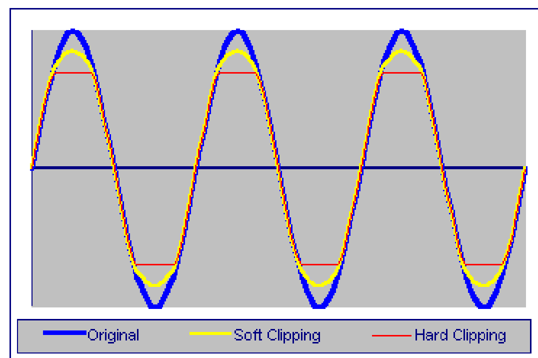


Abbildung 2-3, Distortion-Prinzip, Quelle [2]

Parameter:
Beschneidungsgrad, Höhen/Tiefen-Regler, Effektlevel

2.5 Reverb

Der Effekt Reverb, zu Deutsch Hall, simuliert die kontinuierliche Reflexion von Schallwellen in einem Raum. Je glatter und härter die Wandflächen sind, desto besser werden die Schallwellen reflektiert. Je nach Beschaffenheit eines Raumes ergeben sich somit völlig unterschiedliche Klangcharakteristiken. Was der Mensch als Hall empfindet, ist das unterschiedlich starke und zeitlich unterschiedlich verzögerte Eintreffen dieser Reflexionen beim Hörer.

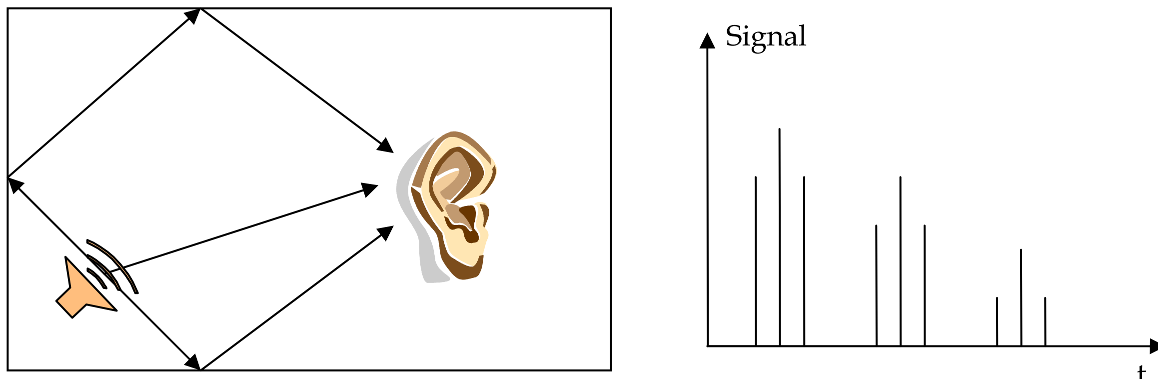


Abbildung 2-4, Darstellung des Halleffektes

Analog werden Hall-Effekte mit Metall-Federn erzeugt, die unter Strom eine Schwingung erzeugen. Digital werden dem Ursprungssignal unterschiedlich stark verzögerte Signalkopien hinzugefügt, wobei bei der Wahl der verschiedenen Verzögerungen und Intensitäten meist versucht wird, eine gewünschte Raumcharakteristik zu simulieren.

Parameter:
Intensität

2.6 Wah

Der Wah-Effekt, oder auch Wahwah genannt, besteht aus einem Bandpass, dessen Frequenzantwort sich im Frequenzspektrum nach oben und unten bewegt.
Grundsätzlich gibt es drei Möglichkeiten, ein Wah zu bauen: Man hat ein Pedal, mit dem man den Bandpass (logarithmisch) verschiebt oder man lässt den Bandpass mit einer konstanten Schwingung hin und her fahren (Autowah) oder man bewegt den Bandpass umso schneller und weiter, je lauter das Signal ist (Dynamic Wah).
Da wir nicht sicher wussten, in welchem Bereich sich ein Wah-Bandpass bewegt, haben wir ein CryBaby-Pedal von Dunlop in verschiedenen Pedalstellungen ausgemessen. Es hat sich gezeigt, dass sich die Spitze in den Extrempositionen bei 400 Hz respektive bei 1.24 kHz befindet (siehe Abbildung 2-6).

Abbildung 2-5, Dunlop CryBaby Pedal, Quelle [3]

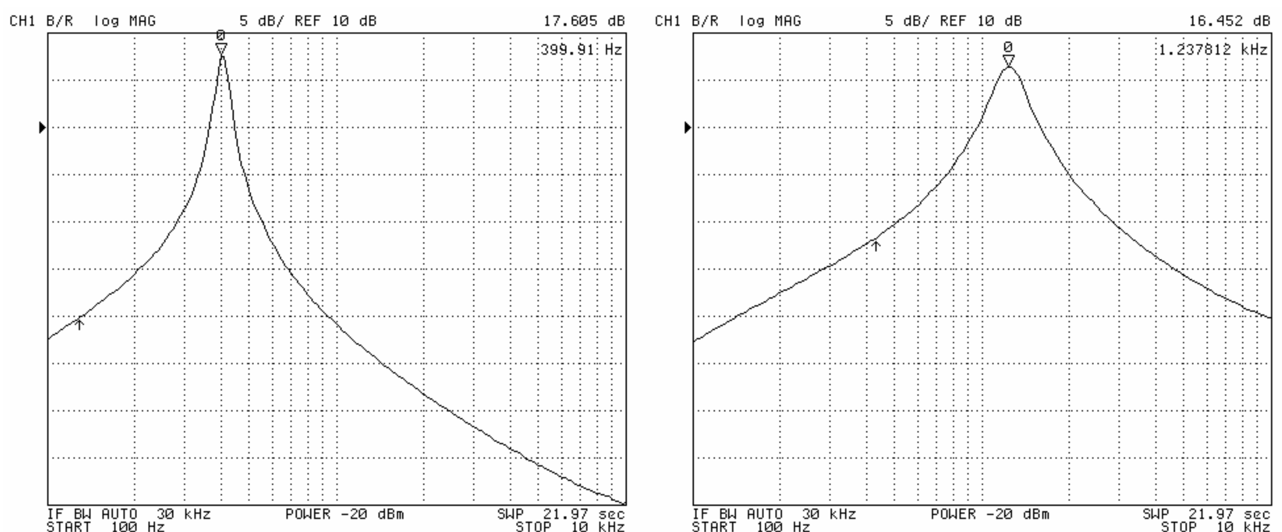


Abbildung 2-6, Frequenzgang des Dunlop CryBaby in beiden Extremstellungen

Parameter:
Vom Wah-Typ abhängig

2.7 Phaser (Phasenschieber)

Der Phasen-Schieber erreicht seinen besonderen Klang, in dem er eine oder mehrere Kerben (notches) in seinem Frequenzgang erzeugt. Die Position dieser Kerben ändert dauernd. Der Frequenz-Anteil des Audiosignales, der an diesen Kerben liegt, wird herausgefiltert.

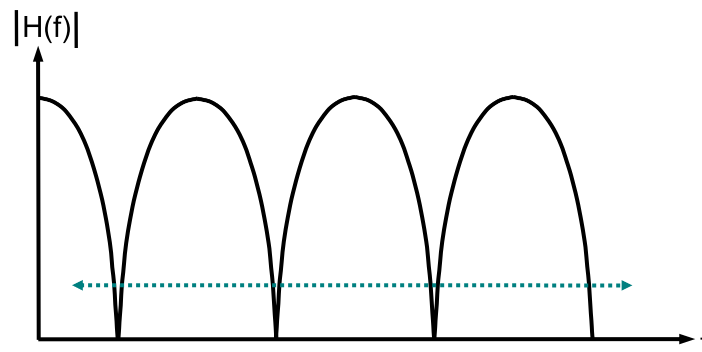


Abbildung 2-7, Phaser-Prinzip

Parameter:
Änderungsgeschwindigkeit, Verschiebungsbereich, Effektlevel

2.8 Looper

Der Looper ist eigentlich kein eigentlicher Effekt, doch ist er ein äusserst interessantes Hilfsmittel für ein kreatives Gitarrenspiel. Der Looper zeichnet bei getretenem Taster bis zum Loslassen das Signal auf und gibt es gleichzeitig unverändert aus. Wie der Name Looper bereits vermuten lässt, wird diese aufgezeichnete Passage nun immer wieder abgespielt. Dabei kann bei erneutem Treten des Tasters wiederum etwas hinzugefügt werden.
Ein wesentliches Qualitätsmerkmal eines Loopers ist die maximale Aufzeichnungsdauer.

Parameter:
In der Regel keine

3 Experimente in MatLab

3.1 Schwierigkeiten

Nachdem wir die verschiedenen Effekte theoretisch angeschaut hatten, haben wir versucht, mittels MatLab die Effekte auszuprobieren, da die DSP-Boards noch nicht verfügbar waren.

Da keine Spezifikationen und kaum detaillierte Erklärungen über die verschiedenen Effekte existieren, gestaltete sich die Nachbildung relativ mühsam. Andererseits war die „Forscheraufgabe“ auch ein interessanter Aspekt unserer Arbeit. Ein Programm konnte absolut korrekt sein, aber aufgrund falscher Parametereinstellungen völlig anders klingen. Durch Abändern von jeweils immer nur einem Parameter tastete man sich langsam an den gewünschten Effekt heran. Bei einigen Effekten funktionierte dies relativ gut, andere wiederum erreichten nie den Klang, den man sich eigentlich wünschte, woraus man dann schliessen konnte, dass die Lösung noch nicht optimal war.

3.2 MatLab-Programme

3.2.1 Delay

Zuerst wird das ursprüngliche Signal x in die Ausgangsvariable geschrieben. Danach werden zu jeder Stelle die mit Faktoren gewichteten Werte von vorherigen Zeitpunkten hinzu addiert.
Die For-Schleife beginnt erst bei 9001, da die längste Verzögerung 9000 Samples beträgt, was bedeutet, dass bereits 9000 Samples „vorbei“ sein müssen, ehe das verzögerte Signal hinzu addiert werden kann (vor Null existiert nichts!). Die 1 rührt daher, dass der Vektorindex in MatLab von 1 ausgeht und nicht von 0 wie z.B. in C.

```
x = wavread('guit1');
y=x;
dim=length(x);
for i = 9001:dim
    y(i) = 0.7*x(i) + 0.4*x(i-3000) + 0.3*x(i-6000) + 0.2*x(i-9000);
end;
soundsc(y,16000);
```

3.2.2 Tremolo

Der Signalpegel des Eingangssignals wird mittels Kosinus variiert, wobei die Gesamtlautstärkeänderung zwischen 0.5 und 1 liegt.

```
x = wavread('guit1');
y=x;
dim=length(x);
for j = 1:dim
    y(j) = (0.75+0.25*cos(j*2*pi/2000))*x(j);
end;
soundsc(y,16000);
```

3.2.3 Chorus

Der abgebildete Code implementiert einen zweistimmigen Chorus. Das bedeutet, dass dem Originalsignal zwei unterschiedlich stark verzögerte Signale hinzugefügt werden, wobei die beiden Verzögerungen sich über die Zeit ebenfalls noch ändern.
Wir erzeugen in unserem Code zwei unterschiedlich schnelle, um 1 angehobene Kosinusschwingungen. Diese setzen wir in der Haupt-For-Schleife an den Stellen ein, wo beim Delay Konstanten stehen. Statt z.B. konstant 30 Samples Verzögerung steht m(j)*30, wobei m ein Kosinus-Vektor ist. Natürlich muss dieser Wert mit round(...) noch auf eine ganze Zahl gerundet werden, da es keine gebrochenen Samples gibt.

```
x = wavread('guit1');
y=x;
dim=length(x);
m=zeros(dim,1);
n=zeros(dim,1);
for i = 1:dim
    m(i)=1+cos(i*2*pi/16000);
    n(i)=1+cos(i*2*pi/12000);
end;
for j = 61:dim
    y(j) = 0.7*x(j) + 0.3*x(j-round(m(j)*30)) + 0.3*x(j-round(n(j)*20));
end;
soundsc(y,16000);
```

3.2.4 Distortion

Eine einfache Möglichkeit einen Verzerrer zu implementieren, ist das Signal ab einem gewissen Schwellwert einfach auf diesen Wert „hart“ zu begrenzen (siehe 2.4). Eine Lösung, die sich als wesentlich besser erwiesen hat, ist die Multiplikation des Eingangssignals mit einem vom Eingangspegel abhängigen Verstärkungsfaktor. Die Verstärkung in Abhängigkeit des Einganspegels ist bei uns eine gauss’sche Glockenkurve. Diese Form hat sich als einigermassen gut erwiesen.

```
x = wavread('guit1');
y=x;
dim=length(x);
sigma=.3;
for j = 1:dim

  y(j) = 2.5*x(j)*( 1/sqrt(2*pi*sigma))*exp((-1/2*(x(j)/sigma)^2));   %Gauss

end;
soundsc(y,16000);
```

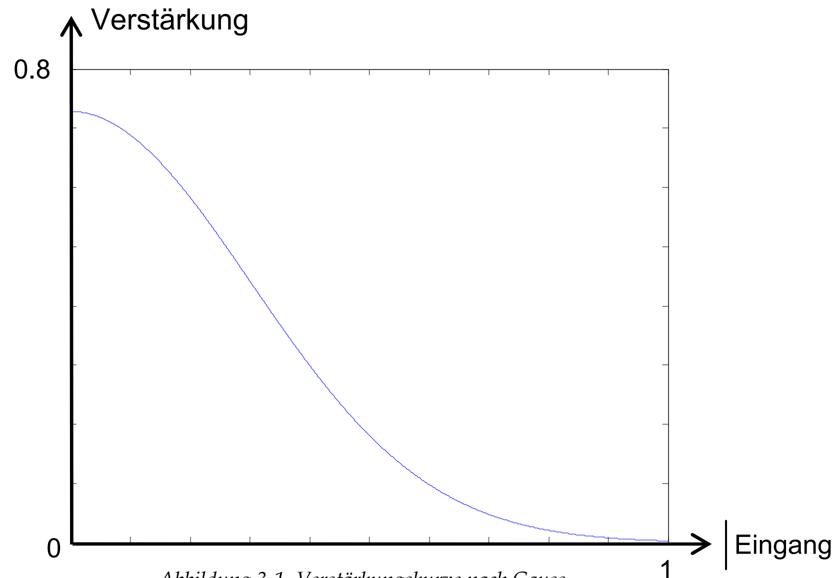


Abbildung 3-1, Verstärkungskurve nach Gauss

3.2.5 Phaser

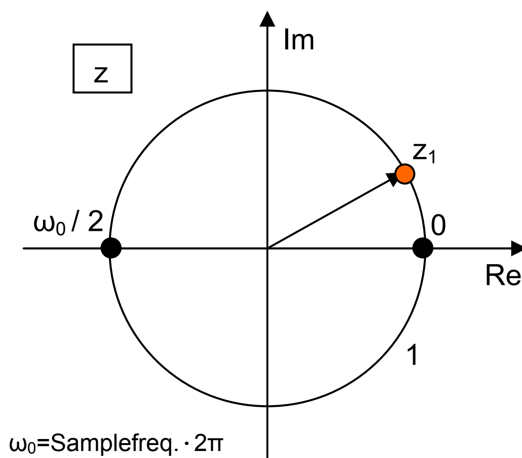


Abbildung 3-2, Pol/Nullstellen-Diagramm

Um im Frequenzspektrum sehr selektiv einzelne Frequenzbereiche herausfiltern zu können, wird wie in 2.7 beschrieben ein sogenanntes Notchfilter benötigt.
Ein solches zu designen hat uns anfänglich ziemlich Mühe bereitet, da wir absolut keine Erfahrung in diesem Gebiet hatten. Erschwerend kam hinzu, dass wir nicht einfach ein fixes Filter brauchten, sondern eines, dessen Notches sich zeitlich hin und her bewegen.
Für feste Filter existieren in MatLab gute Toolboxen, welche einem fertig berechnete Filter-Koeffizienten liefern. Das Problem dabei ist jedoch, dass bei einer zeitlichen Änderung der Notchfrequenzen diese stets neu berechnet werden müssen. Also mussten wir eine relativ einfache Möglichkeit finden, um die Koeffizienten eines Notchfilters selber berechnen zu können.
Herr Schuster riet uns, in der Z-Ebene auf dem Einheitskreis bei unseren gewünschten Frequenzen Nullstellen zu setzen (→ Notches) und daraus die benötigten Koeffizienten zu berechnen. Um den mathematischen Aufwand gering zu halten, versuchten wir zuerst, ein Filter mit nur einem Notch zu rechnen:

$z_1=e^{j\omega_1 T}$
$\omega_1=(1/80) \bullet \omega_0$ (Notchfrequenz bei 200 Hz, wenn die Samplefrequenz 16 kHz beträgt)

$H(\omega) = (z - z_1)(z - z_1^*)$
$H(\omega) = z^2 - (z + z_1^*)z + z_1 z_1^*$
$H(\omega) = z^2 - \cos(\omega_1 T)z + 1$
$H(\omega) = 1 + b_1 z + z^2 \rightarrow \mathbf{b_1 = -\cos(\omega_1 T)}$

Das einfache Filter haben wir danach in MatLab programmiert, wobei wir die Koeffizienten mittels eines Dreieckvektors variiert haben. Nach einigen Parameteranpassungen klang das Filter in etwa nach einem Phaser!

```
x = wavread('guit1');
dim=length(x);
y=zeros(dim,1);
m=zeros(dim,1);
len=20000;
cnt=0;

%Dreieckform generieren:
for i=1:dim
    if cnt==len
        cnt=0;
    else
        cnt=cnt+1;
    end;

    if cnt<(len/2)
        m(i)=(2/len)*(cnt-1);
    else
        m(i)=(2/len)*(len-cnt);
    end;
end;

for n = 3:dim
    y(n)= x(n) -1.2*cos((1+70*m(n))*400*1/16000)*x(n-1) + x(n-2);
    y(n)= 0.8*y(n) + 0.2*x(n); %Ausgangsmix
end;

soundsc(y,16000);
```

3.2.6 Wah

3.2.6.1 Erste Wah-Implementation

Unser erster Versuch, ein Wah-Effekt mit MatLab zu erzeugen, ist nachfolgend zu sehen. Wir haben uns in der Filterdesign-Toolbox von MatLab die Filterkoeffizienten eines FIR-Tiefpasses 50. Ordnung berechnen lassen. Dann haben wir diesen Tiefpass durch eine Multiplikation mit einem Kosinus zu einem Bandpass verschoben. Die Mittenfrequenz dieses Bandpasses bewegt sich zeitabhängig im Frequenzbereich hin und her. Diese Bewegung im Frequenzbereich wird durch eine kosinusförmige Änderung der Bandpass-Mittenfrequenz erreicht.
Wir haben Bandpässe mit unterschiedlichen Bandbreiten versucht, auch haben wir die zeitabhängige Änderung der Mittenfrequenz langsamer oder schneller gemacht, doch all diese Änderungen brachten einfach nicht den gewünschten Klang des Effektes.

```
x = wavread('guit1');
dim=length(x);
y=zeros(dim,1);
fc=1600;
b = [...
     0.0008469279925034,0.0008973136625483, 0.001026543226988, 0.001235559345934,...
     0.001523764033247, 0.001889000733577, 0.002327568277355, 0.002834266836469,...
     0.003402475308575, 0.004024258869593, 0.004690504767488, 0.005391083801075,...
     0.006115034349852, 0.006850765307968,  0.00758627383943, 0.008309373522677,...
     0.009007928199055, 0.009670086687645,  0.01028451348229,  0.01084061060685,...
     0.01132872597101,  0.01174034383778,  0.01206825337958,   0.0123066917549,...
     0.01245145867098,            0.0125,  0.01245145867098,   0.0123066917549,...
     0.01206825337958,  0.01174034383778,  0.01132872597101,  0.01084061060685,...
     0.01028451348229, 0.009670086687645, 0.009007928199055, 0.008309373522677,...
     0.00758627383943, 0.006850765307968, 0.006115034349852, 0.005391083801075,...
     0.004690504767488, 0.004024258869593, 0.003402475308575, 0.002834266836469,...
     0.002327568277355, 0.001889000733577, 0.001523764033247, 0.001235559345934,...
     0.001026543226988,0.0008973136625483,0.0008469279925034...
     ];

fm=zeros(dim,1);
for j = 1:dim
    fm(j)=(1+cos(j*2*pi/12000))/2 * 100;
end;
for n = 51:dim
    for i = 0:50
         y(n) = y(n) + (2 * b(i+1) * cos(2*pi*fm(n)*i/fc)) * x(n-i);
    end;
end;
soundsc(y,16000);
```

3.2.6.2 Zweite Wah-Implementation

Die folgende MatLab-Implementation des Wah-Effektes ist eine Portierung eines C-Codes, den wir im Internet gefunden haben (Quelle [4]). Laut der Dokumentation dieses Codes wird hier nicht ein sich verschiebender Bandpass, sondern ein sich verschiebendes sogenanntes "Peaking Filter", welches Frequenzen in der Nähe seiner Mittenfrequenz anhebt, die restlichen Frequenzanteile aber nicht beeinflusst, verwendet. Es ist hier als IIR-Filter programmiert.

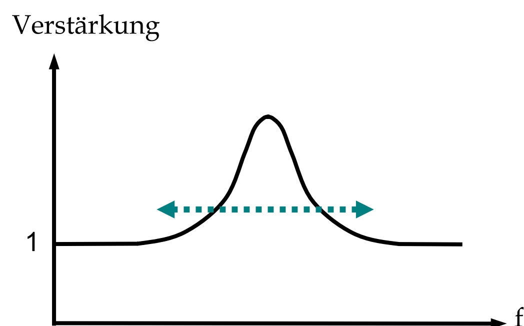


Abbildung 3-3, Peaking-Filter

Die Filterkoeffizienten müssen für jeden Wert, der sich zeitlich ändernde Filter-Mittenfrequenz, berechnet werden. Das ergibt für jede Mittenfrequenz einen neuen Koeffizientensatz. In diesem Programm gibt es 3000 Filtersätze, die in einer 3000x5-Matrix abgespeichert sind. Nach welchen Gesetzmässigkeiten diese Koeffizienten genau berechnet werden, haben wir leider nicht herausgefunden, da uns die benötigten Grundlagen fehlten. Die Geschwindigkeit, mit welcher sich der Peak im Frequenzbereich bewegt, hängt von der Amplitude des Eingangssignals ab, das heisst wie stark eine Gitarrensaite angeschlagen

wird. Je grösser die Amplitude, je weiter wird in der Koeffizienten-Matrix gesprungen, das heisst, es werden in der Matrix mehr Zeilen übersprungen.
Diese Implementation des Wah-Effektes klingt doch recht ansprechend.

In Abbildung Abbildung 3-4 ist mit MatLab der Frequenzgang des Filters mit dem 400. und dem 1300. Koeffizienten-Satz aus der Koeffizienten-Matrix dargestellt worden. Wie man sieht, ist der Peak des ersten Filters bei 400Hz und der Peak des zweiten ungefähr bei 1300Hz.

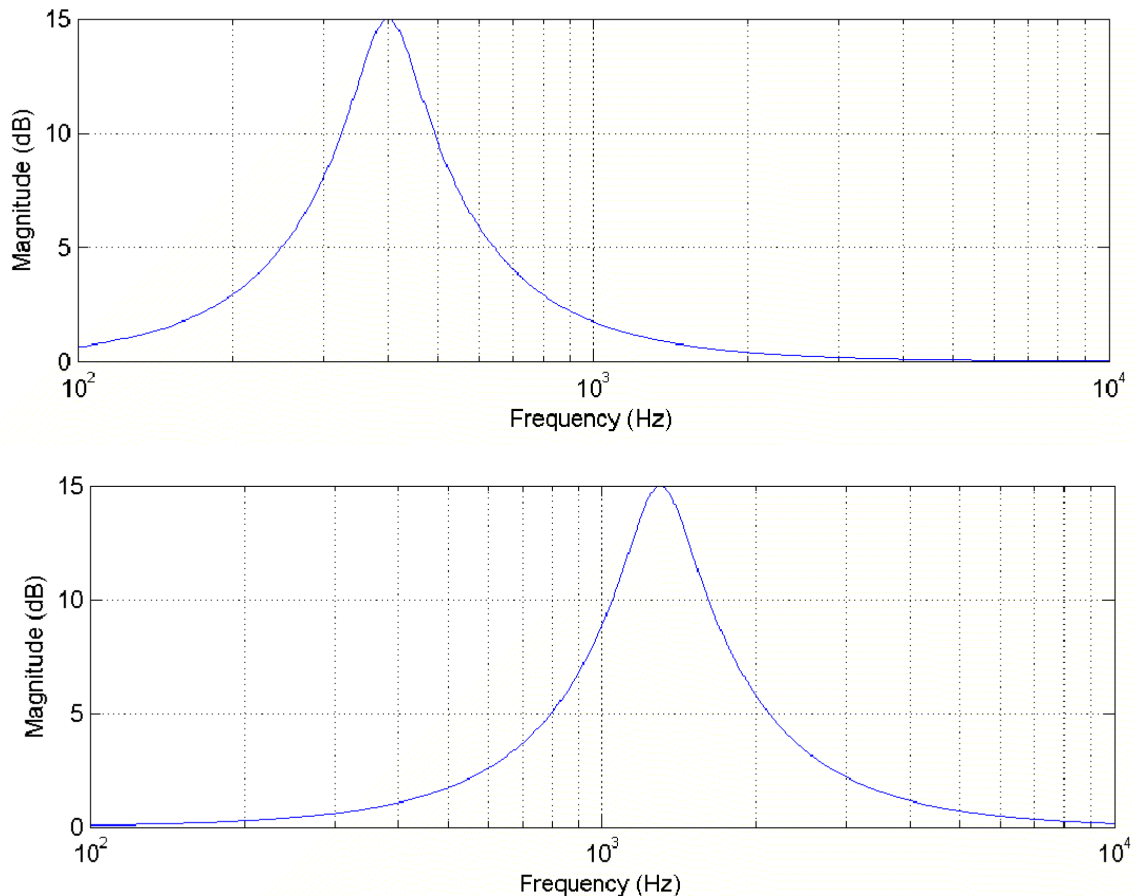


Abbildung 3-4, Amplitudengänge des Peaking-Filters

Diese Amplitudengänge weisen doch eine gewisse Ähnlichkeit mit den Amplitudengängen der beiden Extremstellungen des Dunlop Wah-Pedals von Abbildung 2-6 auf.

```
x = wavread('guit1');
dim=length(x);
y=zeros(dim,1);
dBgain = 5;
RATE = 16000;
rate = 5000;
bandwidth = 5;
minfc = 1;
maxfc = 2000;
filter_coefs = zeros(3000,5);  % Filterkoeffizienten-Matrix
A = exp((dBgain/20)*log(10));  % Verstärkungsumrechnung von dB nach linear
for i=1:3001                   % Berechnung der Filterkoeffizienten-Matrix
      omega = 2*pi/RATE * i;
      sn    = sin(omega);
      cs    = cos(omega);

      if(sn==0)
          sn=0.0000001;
      elseif(cs==0)
          cs=0.0000001;
      end;

      alpha = sn*sin(log(2)/2*bandwidth*omega/sn);

      b0 =   1.0 + alpha*A;
      b1 =  -2.0*cs        ;
      b2 =   1.0 - alpha*A;
      a0 =   1.0 + alpha/A;
      a1 =  -2.0*cs        ;
      a2 =   1.0 - alpha/A;
      filter_coefs(i, 1) = (b0/a0);
      filter_coefs(i, 2) = (b1/a0);
      filter_coefs(i, 3) = (b2/a0);
      filter_coefs(i, 4) = (a1/a0);
      filter_coefs(i, 5) = (a2/a0);
  end;

   min_coef = minfc;
   max_coef = maxfc;
   cur_coef = min_coef;
   step = rate/RATE;

 for n = 3:dim
        cur_coef2 = int32(cur_coef);
        y(n) = filter_coefs(cur_coef2, 1) * x(n)...
                   + filter_coefs(cur_coef2, 2) * x(n-1)...
                   + filter_coefs(cur_coef2, 3) * x(n-2)...
                   - filter_coefs(cur_coef2, 4) * y(n-1)...
                   - filter_coefs(cur_coef2, 5) * y(n-2);


        cur_coef = cur_coef +  step* 5*abs(x(n));  % Je nachdem, wie hoch die
                                                   % Amplitude des
                                                   % Eingangs x ist, wird in der
                                                   % Koeff.-Matrix
       if(cur_coef > max_coef)                     % mehr oder weniger gesprungen.
           cur_coef = max_coef;
           step = -step;
       elseif(cur_coef < min_coef)
           cur_coef = min_coef;
           step = -step;
       end;

end;
soundsc(y,16000);
```

Nachfolgend sollen noch die wichtigsten Parameter des obigen Codes erklärt werden:

`RATE`: Samplefrequenz in Hz

`minfc`: minimale Frequenz in Hz (0 bis 3000)

`maxfc`: maximale Frequenz in Hz (0 bis 3000)

`rate`: die Frequenz der Änderung des Filter-Peaks im Frequenzbereich (0 bis `RATE`)

`dBgain`: Verstärkung des Peaking-Filter in dB (-15 bis 15)

`bandwidth`: Bandbreite in Oktaven (zwischen Filter-Mittenfrequenz und `dBgain`/2 (0 bis 10)

Mit `minfc` und `maxfc` wird der Frequentbereich festgelegt, in dem sich das Peaking-Filter bewegen soll. Mit `rate` kann die Geschwindigkeit angegeben werden, mit der sich das Peaking-Filter im Frequenzbereich hin- und herbewegt.

4 Hardware

4.1 DSP Starter Kit

Für die Realisierung der Effekte steht uns ein TMS320C6711 DSP Starter Kit (DSK) von Texas Instruments zur Verfügung.
Hier einige Daten des im DSK enthaltenen Entwicklungs-Boards:

Die CPU ist ein TMS320C6711-DSP mit einer Taktrate von 150MHz, der 900 MFLOPS ausführen kann.

Zur Speicherung von Programmcode und Daten stehen dem Programmierer, nebst den 64KB internem RAM des DSPs, 16MB SDRAM und 128KB Flash-EPROM zur Verfügung.

Die Karte enthält ausserdem einen 16Bit AD/DA-Wandler mit einer maximalen Sample-Rate von 8kHz und eine Schnittstelle für Erweiterungskarten (daughter cards).

Nebst der Entwicklungskarte ist im DSK auch das Code Composer Studio enthalten. Mit dieser Programmierumgebung kann das Entwicklungs-Board über die parallele Schnittstelle des PCs entweder in Assembler, oder vor allem in C programmiert werden. Mehr zum Code Composer in Abschnitt 6.

4.2 Audio Daughter Card

Weil der AD-Wandler auf der Karte nicht die benötigte Samplefrequenz aufbringt (siehe auch 4.3), mussten wir noch eine Zusatzkarte, die Audio Daughter Card TMDX326040A, zur Hilfe nehmen.
Diese Zusatzkarte hat einen Stereo-AD/DA-Wandler, der wahlweise digitale Werte von 16- oder 20Bit liefert, ausserdem hat man mit dieser Card eine Samplefrequenz von 48kHz zur Verfügung. Der AD/DA-Wandler ist über den McBSP1 (siehe 5.1) mit dem DSP verbunden.

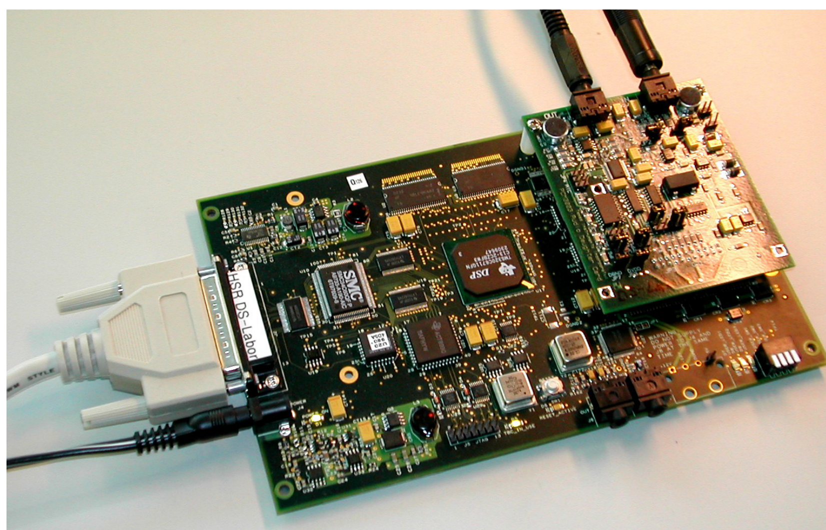


Abbildung 4-1, DSP-Board mit Daughtercard

4.3 Erforderliche Samplefrequenz

Um uns entscheiden zu können, ob wir die zusätzliche Audio-Daughtercard benötigen, oder ob die Abtastfrequenz des A/D-Wandlers auf dem Grundboard mit 8 kHz ausreicht, mussten wir in Erfahrung bringen, in welchem Frequenzspektrum sich die Töne einer Gitarre bewegen. Dazu führten wir zwei Messung durch, die uns Aufschluss darüber gaben, wie das Frequenzspektrum einer Gitarre aussieht und wo etwa die Frequenzgrenze ist, ab der man klar wahrnimmt, dass Oberwellen fehlen.

Bei der ersten Messung (Messung a in Abbildung 4-2) massen wir das Spektrum einer E-Gitarre. Der Spektrumanalyzer zeigte bei hohen Tönen, vor allem beim Anschlagen einer Saite, Oberwellen bis zu 20 kHz, während die tiefst mögliche Frequenz bei 82.5 Hz liegt (tiefe E-Saite)!
Beim Hörtest (Messung b in Abbildung 2-7), bei dem wir das verstärkte Gitarrensignal durch ein digitales Tiefpassfilter auf einen Kopfhörer ausgaben, stellten wir fest, dass man die Grenzfrequenz des Tiefpasses bis auf 8 kHz hinunterdrehen kann, ohne dass der Klang wesentlich beeinflusst wird.

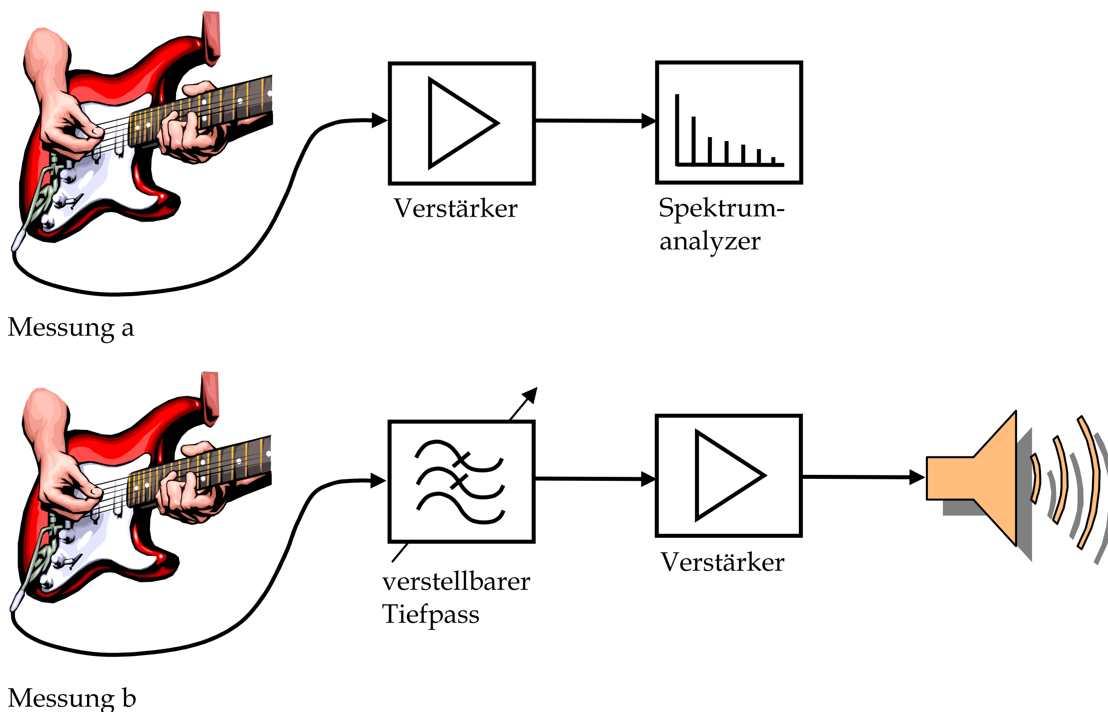


Abbildung 4-2, Gitarrenmessungen

Aufgrund dieses Hörtestes haben wir uns dann für den Einsatz der Audio-Daughtercard entschieden, da ja theoretisch mindestens mit $2 \cdot 8$kHz = 16KHz abgetastet werden müsste, um das Signal dann auch wieder rekonstruieren zu können.

4.4 Geplante Zusatzhardware

Anfangs war für unser Effektgerät eine Zusatz-Hardware geplant, die wir aber aus Zeitgründen nicht realisieren konnten, da wir auf die Qualität der Effekte Wert legten. Hier

soll nun aber trotzdem erklärt werden, wofür diese Hardware-Erweiterung gedacht war und was wir uns zu deren Realisierung überlegt haben.

Mittels Schaltern kann man die einzelnen Effekte auswählen und über die Regel-Potentiometer können die verschiedenen Parameter der Effekte eingestellt werden. Die Poti-Stellungen werden dann über einen 16-zu-1-Multiplexer (z.B. 74HC4067) zum AD-Wandler durchgeschalten. Wenn mehr als 16 Effekt-Parameter geändert werden müssen, bräuchte man dann einfach zwei oder drei dieser Multiplexer. Diese Multiplexer sollten entweder über einen möglichst einfachen Mikrokontroller oder vom DSP direkt angesteuert werden. Im ersten Fall hätten wir dann gerade einen Kontroller verwendet, der auch einen AD-Wandler integriert hat. Wir haben uns auch überlegt, ob man nicht gerade den zweiten, freien Kanal der Audio-Daughtercard für das Einlesen der Regler-Stellungen benutzen könnte.

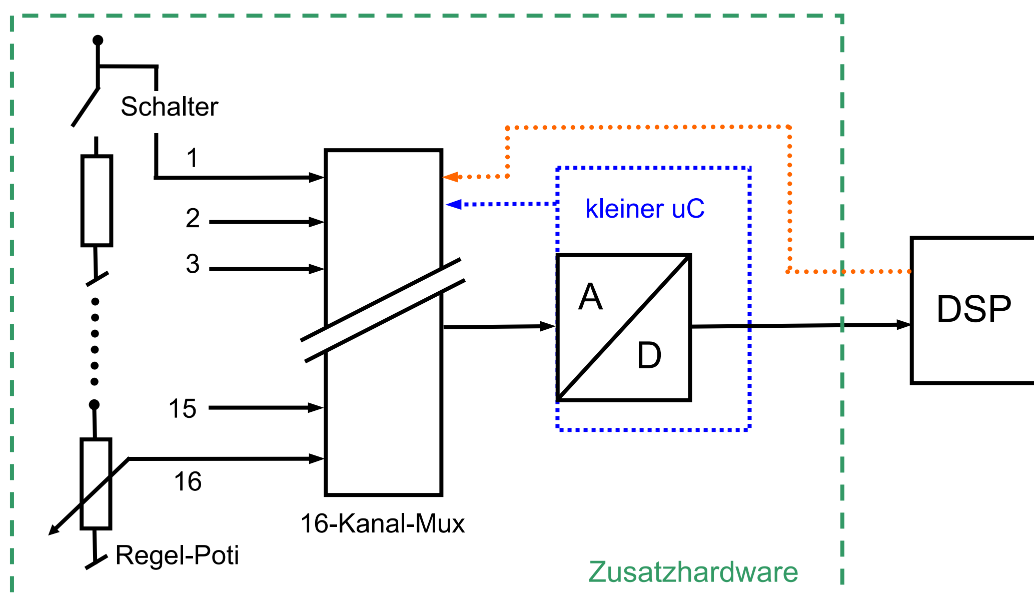


Abbildung 4-3, Zusatzhardware

5 Datentransfer

Nachfolgend möchten wir erklären, welchen Weg das Musiksignal von der Gitarre, über das DSK-Board (mit Audio-Erweiterungskarte), bis zum Verstärker zurücklegt. Dabei ist natürlich vor allem interessant, was auf dem DSK-Board mit dem Signal passiert, da die restliche Wegstrecke ja nur aus Kabeln besteht.

5.1 McBSP

McBSP bedeutet Multi Channel Buffered Serial Port. Er ist eine serielle Schnittstelle zwischen dem DSP und der Peripherie. Der McBSP ist vor allem dafür konzipiert mit AD- und DA-Wandlern zu kommunizieren. Der TMS320C6711 hat zwei solche Ports, den McBSP0 und den McBSP1.
Der McBSP0 ist im DSK mit dem auf dem Board vorhandenen AD/DA-Wandler verbunden.
Der McBSP1 ist auf die Schnittstelle für Erweiterungskarten geführt, wo er vom AD/DA-Wandler der Audio-Erweiterungskarte für den Datenaustausch mit dem DSP gebraucht wird.

Für den McBSP kann entweder ein interner oder ein externer Takt verwendet werden. In unserem Fall wird die serielle Taktrate von der Audio-Erweiterungskarte geliefert, so dass 48'000 32Bit-Samples pro Sekunde übertragen werden. Wenn man nun nicht 32Bit lange Datenworte übertragen möchte, könnte die Datenwort-Breite auch anders eingestellt werden.

5.2 EDMA

EDMA steht für Enhanced Direct Memory Access. Mit Direct Memory Access meint man, wenn der Hauptprozessor einem DMA-Kontroller den Auftrag geben kann, selbständig Daten z.B. zwischen zwei Speicherbausteinen zu verschieben. Der Hauptprozessor muss den Bus während der Zeit, in der der DMA-Kontroller Daten verschiebt, an diesen abgeben. Der DMA-Kontroller kann also immer dann im Hintergrund Daten transferieren, wenn der Hauptprozessor den Bus gerade nicht braucht. Der DSP TMS320C6711 hat einen solchen DMA-Kontroller integriert.

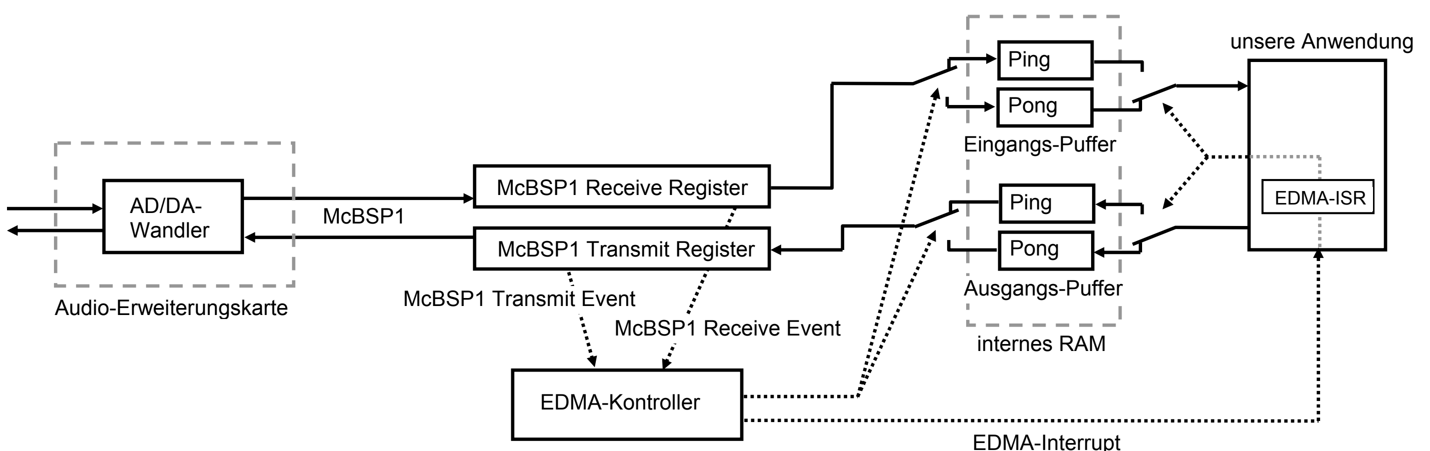


Abbildung 5-1, Signalweg

Nachfolgend soll kurz erläutert werden, was der EDMA in unserem Programm für eine Funktion einnimmt.

Wenn der *AD-Wandler* der Audio-Erweiterungskarte ein Wert ins *McBSP1 Receive Register* geschrieben hat, so wird der *McBSP1 Receive Event* ausgelöst.

Im EDMA-Register *ESEL* (Event Selector Register) kann angegeben werden, auf welchen Event welcher der 16 EDMA-Kanäle reagieren soll. Per default reagiert Kanal 15 auf den *McBSP1 Receive Event* und Kanal 14 auf den *McBSP1 Transmit Event*. Im EDMA-Register *EER* (Event Enable Register) muss dann noch die Freigabe erfolgen, damit EDMA-Kanal 14 und 15 auch wirklich anfangen, auf diese Events zu reagiern.

Hat der EDMA-Kontroller den *McBSP1 Receive Event* erhalten, schaut er in seiner Konfigurationstabelle (im Parameter RAM) im Eintrag für den zu diesem Event gehörenden Kanal nach, was er jetzt mit den Daten im *McBSP1 Receive Register* tun soll. Solch ein Eventparameter-Eintrag enthält unter anderem dies:

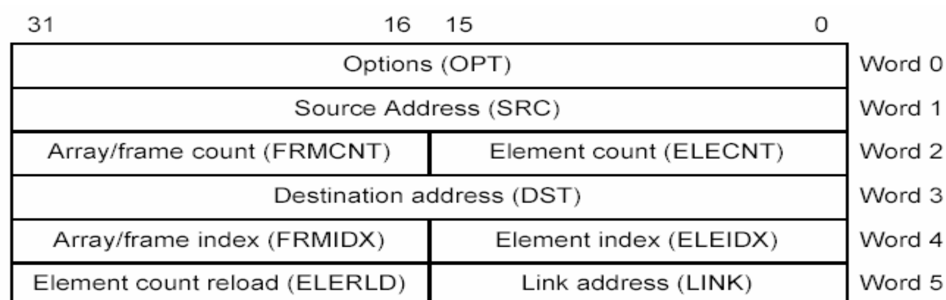
OPT (Options Parameter)
Hier können einige Optionen angegeben werden. Es werden hier nur die drei für uns wichtigsten erläutert. Man kann angeben, dass die Source- oder die Destination-Adresse nach jedem Datenwort, das innerhalb eines EDMA-Transfers gesendet wird, um eins erhöht wird. Ausserdem welches Bit im *CIPR* (Channel Interrupt Pending Register) gesetzt werden soll, wenn der EDMA-Transfer komplett ist. Auch kann man angeben, dass der EDMA-Kontroller nach Beendigung eines EDMA-Transfers einen anderen Eventparameter-Eintrag für diesen Event benutzen soll. Die Adresse dieses neuen Event-Eintrages steht in *LINK*.

SRC (Source Address Parameter)
Der Ort, an dem der EDMA-Kontroller die Daten abholen kann.

CNT (Count Parameter)
Wie viele Datenworte ein EDMA-Transfer beinhalten soll.

DST (Destination Address Parameter)
Der Ort, an den der EDMA-Kontroller die Daten hinschreiben soll.

LINK (Link Address Parameter)
Die Adresse des Event-Eintrages, der bei einem erneuten Eintreffen des Events die Parameter für den EDMA-Transfer liefern soll.



31 – 16	15 – 0	
Options (OPT)		Word 0
Source Address (SRC)		Word 1
Array/frame count (FRMCNT)	Element count (ELECNT)	Word 2
Destination address (DST)		Word 3
Array/frame index (FRMIDX)	Element index (ELEIDX)	Word 4
Element count reload (ELERLD)	Link address (LINK)	Word 5

Abbildung 5-2, Eventparameter-Eintrag, Quelle [5]

Hier ist nun auch die richtige Stelle, um aufzuzeigen, wie mittels EDMA ein Ping-Pong-Puffer realisiert werden kann: im *Link Address Parameter* des Eventparameter-Eintrages X (dessen *DST* auf Ping zeigt) steht die Adresse von Eventparameter-Eintrag Y (dessen *DST* auf Pong zeigt) und im *LINK* von Y steht die Adresse von X. So werden die Eventparameter für den Event immer abwechslungsweise aus X und aus Y entnommen.

Hat der EDMA-Kontroller die in *CNT* definierte Anzahl an Datenworten (z.B. vom *McBSP1 Receive Register* ins interne RAM des DSPs) verschoben, so meldet er dies mit einem Interrupt. Damit der EDMA-Kanal diesen Interrupt auch auslöst, muss noch im *CIER* (Channel Interrupt Enable Register) die Freigabe erfolgen.
Nun gibt es aber nur einen EDMA-Interrupt für alle EDMA-Kanäle. Wenn mehr als ein EDMA-Kanal verwendet wird, weiss man nicht, welcher EDMA-Kanal den Interrupt ausgelöst hat. Um genaueres über den Grund des EDMA-Interrupts zu erfahren, muss man im *CIPR* (Channel Interrupt Pending Register) prüfen, welche Bits dort gesetzt sind. Vielleicht erinnern Sie sich noch, dass man im Eventparameter *OPT* angeben kann, welches Bit im *CIPR* gesetzt werden soll, wenn ein Event eintritt.
In der EDMA Interrupt Service Routine kann dann anhand der in *CIPR* gesetzten Bits ermittelt werden, welcher EDMA-Kanal den Interrupt ausgelöst hat.

In der EDMA Interrupt Service Routine muss nun jedes mal zwischen dem Ping- und dem Pong-Puffer hin- und her geschaltet werden. Auch müssen die in *CIPR* gesetzten Bits wieder gelöscht werden. Komischerweise kann man diese Bits nur zurücksetzen, wenn man sie nochmals mit einer 1 überschreibt.

Für detailliertere Ausführungen zu DMA wird auf den "TMS320C6000 EDMA Controller Reference Guide" [5] von Texas Instruments und entsprechende Fachliteratur verwiesen.

6 Code Composer Studio

Das Code Composer Studio (CCS) ist die Entwicklungsumgebung, die zum Umfang des TMS320C6711 DSP Starter Kits gehört. Es beinhaltet einen C-Compiler inklusive Debugging-Tools und lehnt sich optisch sehr an Visual C++ an, was einem ermöglicht, ohne grosse Einarbeitung damit zu arbeiten. Zudem bietet es verschiedenste komfortable Features an, die bei der DSP-Programmierung äusserst hilfreich sein können. Nachfolgend werden die für uns wesentlichen erläutert. Für detailiertere Informationen verweisen wir an dieser Stelle auf die TMS320C6000 Code Composer Studio Manuals.

Wir haben leider die Erfahrung gemacht, dass das CCS in der Version 2.10.0 gewisse Stabilitätsprobleme hat. Oftmals treten unerklärliche Fehler auf, welche meist nur durch Speisungsunterbruch des DSP-Boardes und Neustart des CCS beheben lassen.

6.1 RTDX

Das Code Composer Studio hat eine Funktionalität, mit deren Hilfe man vom PC aus Parameter eines gerade auf dem DSK-Board ablaufenden Programms ändern kann. Diese Funktionalität nennt sich *R*eal *T*ime *D*ata *E*xchange. RTDX verwenden wir in Zusammenhang mit den in GEL definierten Slidern (siehe 6.2.3), um Effektparameter zu verändern, während das Effekt-Programm am Laufen ist.
Wir glaubten lange Zeit, dass man sich die Funktionalität von RTDX auch ausserhalb des Code Composers zu Nutze machen kann, z.B. in Visual C++. Wir nahmen an, man könne eine RDTX-DLL in Visual C++ einbinden und dann ein GUI mit den entsprechenden Sliders und Buttons erstellen, das ohne Code Composer lauffähig ist.
Leider haben wir auch nach intensivsten Nachforschungen nicht herausgefunden, wie man RTDX mit VC++ realisieren kann. Erst nach einem längeren Email-Wechsel mit Texas Instruments bekamen wir die klare Antwort, dass RTDX nur möglich ist, wenn der Code Composer im Hintergrund läuft .

6.2 Parametersteuerung mit GEL

6.2.1 Was ist GEL

GEL steht für General Extension Language, ist eine Code Composer eigene Programmiersprache und dem C-Syntax sehr ähnlich. GEL dient dazu, die Möglichkeiten des Code Composer Studios noch besser auszunutzen. Das heisst, man kann verschiedene Hilfsmittel der Entwicklungsumgebung für eigene Überwachungs- oder Testprogramme verwenden.
Wir benutzen GEL dazu, um zu Laufzeit gewisse Variablenwerte auf dem DSP-Board zu verändern, was durch GEL wesentlich einfacher ist, als wenn man alles selbst via Real-Time Data Exchange (RTDX, siehe 6.1) programmieren muss.

6.2.2 Verwendung eines GEL-Files

Um ein geschriebenes GEL-File zu verwenden, muss man es manuell laden. Man tut dies, indem man im CCS im linken Fenster mit der rechten Maustaste auf den Ordner GEL-files klickt und anschliessend die entsprechende *.gel Datei auswählt.

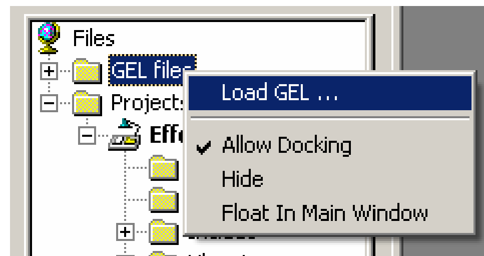


Abbildung 6-1, GEL-Datei laden

6.2.3 GEL Slider

Die GEL-Slider ermöglichen es, zu Laufzeit Variablenwerte im Programmcode zu beeinflussen. Die Abstufung und die Wertebereiche, die man einstellen kann, sind wählbar, jedoch mit der Einschränkung, dass nur ganze Zahlen verwendet werden können. Man kann also beispielsweise nicht etwas im Bereich von 0.1 bis 1 in Schritten der Grösse 0.1 einstellen, sondern nur von 1 bis 10 in Schritten der Grösse 1.

Mit dem Schlüsselwort `menuitem` wird im CCS unter dem Menüpunkt GEL ein neuer Untermenüpunkt generiert, in dem dann die ausprogrammierten Slider als Auswahl erscheinen (siehe Abbildung 6-3).

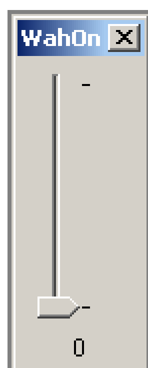


Abbildung 6-2, GEL-Slider

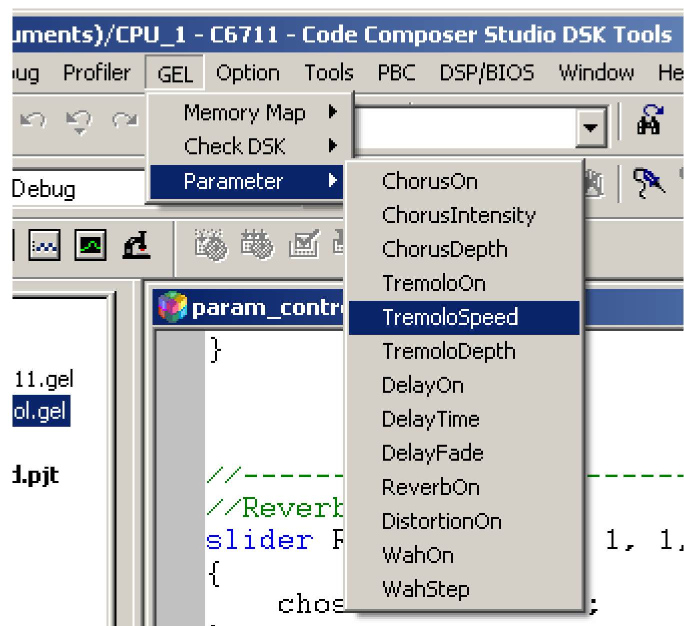


Abbildung 6-3, GEL-Auswahl

Hier ist ein Beispielprogramm für einen Slider, der Werte zwischen 10 und 1000 annehmen kann. Parameter 3 gibt die Schrittweite beim Verstellen des Sliders an, Parameter 4 die Schrittweite beim Drücken der Tasten PageUp/PageDown. Der fünfte Parameter ist die eigentliche Slidervariable, die den eingestellten Wert annimmt. In der Funktion selbst braucht der Wert dieser Variable lediglich noch der Variablen im C-File zugeordnet zu werden.

```
menuitem "Parameter"

slider WahStep(10, 1000, 10, 10, val)
{
	nStep = val;
}
```

6.3 Profiler

Der Profiler ermöglicht es, zu ermitteln, in welchen Code-Segmenten wie viel Rechenzeit benötigt wird. Dadurch kann man seinen Code gezielt optimieren. In unserem Fall funktionierte der Profiler leider nicht korrekt, da unser gesamter Effekt-Code in einer Interrupt-Serviceroutine steht, was bei der Simulation scheinbar zu Fehlern führt.
Auf die Handhabung des Profilers möchten wir hier nicht weiter eingehen, da die Bedienung ziemlich einfach und relativ gut dokumentiert ist.

7 Implementierung auf DSP-Board

7.1 Sample- oder blockweises Einlesen der Audiodaten

Nachdem das DSP-Board endlich geliefert wurde und wir von den Diplomanden M.Jurcevic und M.Gloor den Code erhalten haben, der Audiodaten einliest und wieder ausgibt, konnten wir damit beginnen, unsere Effekte auf das Board zu übertragen.
Die erste wesentliche Frage, die sich uns stellte, war, ob die zeitliche Verzögerung durch das blockweise Einlesen ein Problem für uns darstellt. Und dies tat es! Das Spielen einer Gitarre ist äusserst mühsam, wenn man nicht gleich hört, was man spielt. Durch massives Verkleinern der Ping/Pong-Blockgrösse auf 128 Samples konnte die Verzögerung dann allerdings auf ein kaum wahrnehmbares Minimum reduziert werden.
Eine andere Variante wäre gewesen, wenn wir die Daten nicht block- sondern sampleweise verarbeitet hätten. Dies würde bei einer Abtastrate von 48 kHz noch funktionieren und wäre möglicherweise auch einfacher zu implementieren gewesen. Die Wahl auf das blockweise Verarbeiten fiel aufgrund der Tatsache, dass man bei Anwendungen mit höherer Abtastrate ohnehin auf diese Methode zurückgreifen müsste.

7.2 Hilfsfunktionen für Effekte

Da praktisch alle Effekte auf vergangene Signalinformationen zugreifen und die meisten zeitlich durch einen Kosinus beeinflusst werden müssen, haben wir diese Teile als Einzelfunktionen ausgegliedert.

7.2.1 inHistory

Die `inHistory`-Funktion liefert dem Aufrufer den Wert im Eingangsspeicher zurück, der `anz` zurück liegt. Da es sich beim Speicher um einen sogenannten Circular Buffer handelt, kann man nicht einfach den aktuellen Zeiger mit gewünschtem Offset benutzen, sondern muss zuerst feststellen, ob der Wert über oder unter dem aktuellen Zeiger liegt.
Variablen-/Konstantenerklärung:

`actPtr`	Zeiger auf aktuelle Speicherposition
`startPtr`	Zeiger auf Startposition des Buffers
`BUFLEN`	Grösse des Buffers

```
int inHistory(int anz)
{
	if(actPtr-anz < startPtr)
	{
		return *( (startPtr+BUFLEN) -(anz-(actPtr-startPtr)));
	}
	else
	{
		return *(actPtr-anz);
	}
}
```

7.2.2 delHistory

Die `delHistory`-Funktion funktioniert identisch wie die Funktion `inHistory` (7.2.1), jedoch wird sie auf den Eingangsspeicher des Delay-Effekts angewendet.
Wenn man sich die verschiedenen Effekte als in Serie geschaltete Einzeleffekte vorstellt, wird klar, dass eigentlich jeder Effekt seinen eigenen Eingangsspeicher haben sollte. Bei vielen Effekten spielt dies keine enorm grosse Rolle. Beim Delay allerdings, der unter Umständen ziemlich lange Klangsequenzen verzögert wiedergeben muss, ist es merklich hörbar, wenn beispielsweise ein eingeschalteter Hall nicht auf das verzögerte Signal einwirkt.

7.2.3 Kontinuierlicher Raised Cosine

Die Funktion `rsdCos` liefert dem Aufrufer den um 1 erhöhten Kosinuswert des Zählerwertes, auf welcher der Zeiger `cntVal` zeigt. Bei jedem Aufruf wird der durch `cntVal` referenzierte Wert inkrementiert. Erreicht er die Grösse `t`, wird er zurückgesetzt.
In einer ersten Version errechneten wir den Kosinus bei jedem Aufruf, was jedoch ziemlich rechenaufwändig war und so zu Timing-Problemen führte. Deshalb wird nun eine zu Beginn initialisierte Raised Cosine-Tabelle mit 360 Werten verwendet.
Variablen-/Konstantenerklärung:
`M_PI` Kreiskonstante π

```
void initRsdCosTable(void)
{
	int i;
	for(i=0; i<=360; i++)
	{
		rsdCosTable[i] = 0.5 * (1 + cos(2*M_PI/360 * i));
	}
}
```

```
double rsdCos(unsigned int* cntVal, int t)
{
	unsigned int tmp;

	if(*cntVal < t)
	{
		(*cntVal)++;
		tmp = (int) ( 360.0/t * (*cntVal) );
		if(tmp > 360)
		{
		  tmp = tmp%360;
		}
		return rsdCosTable[tmp];
	}
	else
	{
		*cntVal=0;
		return rsdCosTable[0];
	}
}
```

7.3 Implementierte Effekte

Alle folgenden Programmausschnitte befinden sich in der EDMA-Interrupt-Serviceroutine, welche ausgeführt wird, wenn wieder ein neuer Block von Samples eingelesen wurde. Der gesamte Quellcode befindet sich auf der beiliegenden CD-ROM.

7.3.1 Delay

Variablenbeeinflussung durch GEL:

GEL-Slider	*Beeinflusste Variable*	*Zweck der Variable*
DelayOn	choseDelay	Aktivieren des Delay-Effekts
DelayTime	delTme	Einstellen der Wiederholzeitkonstanten
DelayFade	nFade	Beeinflussung des Abklingverhaltens

Jedem aktuellen Bufferwert werden die um 1 bis 7 mal `delTme` Blockgrössen verzögerten Vergangenheitswerte hinzugefügt, wobei die Lautstärken der Wiederholungen quadratisch abnehmen.
Da die Zählvariable i sich bei jedem Durchlauf um eins erhöht, muss sie beim Aufruf der `delHistory-Funktion` (siehe 7.2.2) im Argument subtrahiert werden, damit die Distanz zwischen den Speicherwerten stets gleich bleibt.
Da das Eingangssignal mono ist und sich somit nur im linken Kanal des Stereo-Buffers befindet, muss das aus dem Speicher geholte Sample jeweils noch um 16 Stellen nach rechts geschoben werden.

```
if (choseDelay)
{
    dFade = nFade/100.0; //Bereichsumrechnung nach 0.1 bis 0.9

    for (i=0; i<7; i++)
    {
        coefs[i] = pow(dFade, i); //Quadratisches Abklingverhalten
    }

    # pragma MUST_ITERATE(BLOCK_SIZE,BLOCK_SIZE)
    for(i = 0; i < BLOCK_SIZE; i++)
    {
        (workbuf[i]).ch[LEFT] = ( (workbuf[i]).ch[LEFT]
             + coefs[0] * (short) (delHistory(  delTme*BLOCK_SIZE-i) >>16)
             + coefs[1] * (short) (delHistory(2*delTme*BLOCK_SIZE-i) >>16)
             + coefs[2] * (short) (delHistory(3*delTme*BLOCK_SIZE-i) >>16)
             + coefs[3] * (short) (delHistory(4*delTme*BLOCK_SIZE-i) >>16)
             + coefs[4] * (short) (delHistory(5*delTme*BLOCK_SIZE-i) >>16)
             + coefs[5] * (short) (delHistory(6*delTme*BLOCK_SIZE-i) >>16)
             + coefs[6] * (short) (delHistory(7*delTme*BLOCK_SIZE-i) >>16));
    }
}
```

7.3.2 Tremolo

Variablenbeeinflussung durch GEL:

GEL-Slider	*Beeinflusste Variable*	*Zweck der Variable*
TremoloOn	choseTremolo	Aktivieren des Tremolo-Effekts

TremoloSpeed	tremSpeed	„Schwingungs-Geschwindigkeit“
TremoloDepth	tremDepth	Effektanteil am Ausgangssignal

Mit Hilfe der Funktion `rsdCos` (siehe 7.2.2) wird das aktuelle Signal in der Lautstärke verändert. Die Variable `tremSpeed` beeinflusst die Frequenz des Kosinus.

```
if (choseTremolo)
{
   dDepth = tremDepth/100.0; //Bereichsumrechnung nach 0.1 bis 1

   # pragma MUST_ITERATE(BLOCK_SIZE,BLOCK_SIZE)
   for(i = 0; i < BLOCK_SIZE; i++)
   {
      (workbuf[i]).ch[LEFT] = ((1-dDepth) +
                              dDepth*rsdCos(&cos1, (70-tremSpeed)*BLOCK_SIZE))*
                              ((workbuf[i]).ch[LEFT]);
   }
}
```

7.3.3 Chorus

Variablenbeeinflussung durch GEL:

GEL-Slider	***Beeinflusste Variable***	***Zweck der Variable***
ChorusOn	choseChorus	Aktivieren des Chorus-Effekts
ChorusIntensity	chorusIntensity	Anteil der verzögerten Stimmen am Ausgangssignal
ChorusDepth	chorusDepth	Je grösser, desto weiter ist der Zeitbereich, in dem die einzelnen Stimmen verzögert werden

Hier handelt es sich um einen dreistimmigen Chorus. Im Prinzip funktioniert er ziemlich ähnlich wie der Delay von 7.3.1. Anders ist, dass die Delay-Länge hier durch den `rsdCos` (siehe 7.2.2) zeitlich variiert, die Zeitkonstanten generell viel kleiner sind und drei verzögerte „Stimmen“ hinzugefügt werden.

```
if (choseChorus)
{
   dIntens  = chorusIntensity/10.0; //Bereichsumrechnung nach 0.1 bis 1
   dChDepth = chorusDepth/10.0;     //Bereichsumrechnung nach 0.1 bis 1

   # pragma MUST_ITERATE(BLOCK_SIZE,BLOCK_SIZE)
   for(i = 0; i < BLOCK_SIZE; i++)
   {
      (workbuf[i]).ch[LEFT] = ( (1-dIntens) * (inbuf[bufSel][i]).ch[LEFT] +
          dIntens * ((inbuf[bufSel][i]).ch[LEFT] +
          (short) (inHistory(  (int)( (BLOCK_SIZE + rsdCos(&cos2,114*BLOCK_SIZE) *
                               dChDepth * BLOCK_SIZE))  -i) >>16) +
          (short) (inHistory(  (int)( (BLOCK_SIZE + rsdCos(&cos3, 87*BLOCK_SIZE) *
                               dChDepth * BLOCK_SIZE))  -i) >>16)+
          (short) (inHistory(  (int)( (BLOCK_SIZE + rsdCos(&cos4, 44*BLOCK_SIZE) *
                               dChDepth * BLOCK_SIZE))  -i) >>16)));
   }
}
```

7.3.4 Reverb

Variablenbeeinflussung durch GEL:

GEL-Slider	*Beeinflusste Variable*	*Zweck der Variable*
ReverbOn	choseReverb	Aktivieren des Reverb-Effekts

Auch der Reverb funktioniert nach dem Prinzip des Delays, doch existieren neben den Hauptwiederholungen (im Code etwas nach vorn gerückt) sogenannte Vor- und Nachläufer, welche zeitlich etwas früher oder später auftreten und auch in der Amplitude kleiner sind. So entsteht ein relativ angenehmer Hall.

```
if (choseReverb)
{
   # pragma MUST_ITERATE(BLOCK_SIZE,BLOCK_SIZE)
   for(i = 0; i < BLOCK_SIZE; i++)
   {
      (workbuf[i]).ch[LEFT] = ( (workbuf[i]).ch[LEFT]
                        + 0.20 * (short) (inHistory(25*BLOCK_SIZE-i) >>16)
                        + 0.35 * (short) (inHistory(27*BLOCK_SIZE-i) >>16)
                      + 0.55 * (short) (inHistory(30*BLOCK_SIZE-i) >>16)
                        + 0.35 * (short) (inHistory(33*BLOCK_SIZE-i) >>16)
                        + 0.20 * (short) (inHistory(35*BLOCK_SIZE-i) >>16)

                        + 0.08 * (short) (inHistory(55*BLOCK_SIZE-i) >>16)
                        + 0.12 * (short) (inHistory(57*BLOCK_SIZE-i) >>16)
                      + 0.27 * (short) (inHistory(60*BLOCK_SIZE-i) >>16)
                        + 0.12 * (short) (inHistory(63*BLOCK_SIZE-i) >>16)
                        + 0.08 * (short) (inHistory(65*BLOCK_SIZE-i) >>16)

                        + 0.02 * (short) (inHistory(85*BLOCK_SIZE-i) >>16)
                        + 0.05 * (short) (inHistory(87*BLOCK_SIZE-i) >>16)
                      + 0.15 * (short) (inHistory(90*BLOCK_SIZE-i) >>16)
                        + 0.05 * (short) (inHistory(93*BLOCK_SIZE-i) >>16)
                        + 0.02 * (short) (inHistory(95*BLOCK_SIZE-i) >>16)

                        + 0.01 * (short) (inHistory(125*BLOCK_SIZE-i) >>16)
                        + 0.03 * (short) (inHistory(127*BLOCK_SIZE-i) >>16)
                      + 0.07 * (short) (inHistory(130*BLOCK_SIZE-i) >>16)
                        + 0.03 * (short) (inHistory(133*BLOCK_SIZE-i) >>16)
                        + 0.01 * (short) (inHistory(135*BLOCK_SIZE-i) >>16));
   }
}
```

7.3.5 Distortion

Variablenbeeinflussung durch GEL:

GEL-Slider	*Beeinflusste Variable*	*Zweck der Variable*
DistortionOn	choseDistortion	Aktivieren des Distortion-Effekts

Bei der Distortion wird jeder Samplewert mit seinem Raised-Cosine-Ergebnis multipliziert. Bei der Funktion `rsdCos` wurde als Periodendauer 2000 gewählt (siehe 7.2.3). Diesen Wert haben wir experimentell ermittelt. Wir wussten, dass sich das Maximum der Samplewerte um etwa 1000 bewegt und haben in diesem Bereich verschiedene Werte für die Periodendauer ausprobiert. Die Raised-Cosine-Verstärkungsfunktion ist der Gauss'schen

Glockenkurve sehr ähnlich, die wir für die MatLab-Implementation der Distortion verwendet haben (siehe Abbildung 3-2).

```
if (choseDistortion)
{
   unsigned int inVal;

   # pragma MUST_ITERATE(BLOCK_SIZE,BLOCK_SIZE)
   for(i = 0; i < BLOCK_SIZE; i++)
   {
       inVal = (unsigned int)(workbuf[i]).ch[LEFT];
       (workbuf[i]).ch[LEFT] = (workbuf[i]).ch[LEFT] * rsdCos(&inVal, 2000);
   }
}
```

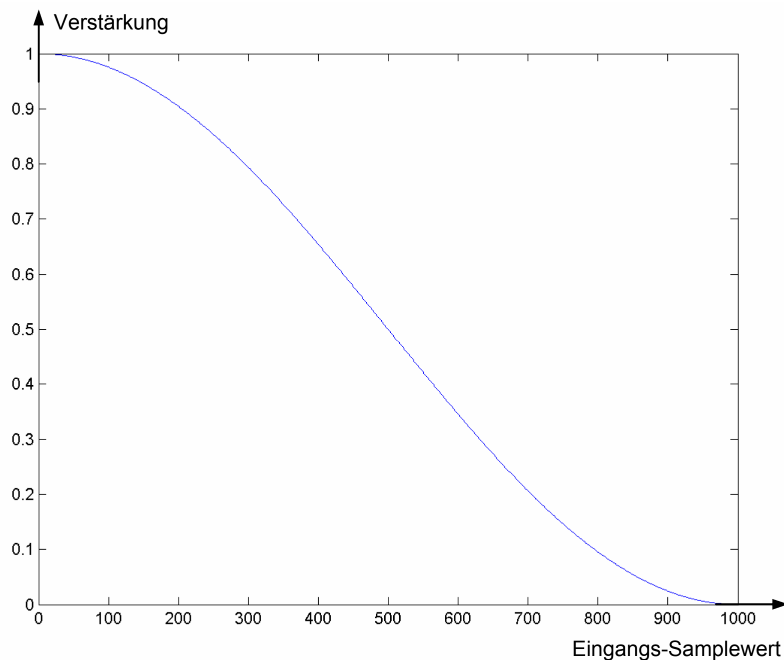


Abbildung 7-1, Raised-Cosine-Verstärkungskurve

7.3.6 Wah

Variablenbeeinflussung durch GEL:

GEL-Slider	*Beeinflusste Variable*	*Zweck der Variable*
WahOn	choseWah	Aktivieren des Wah-Effekts
WahStep	nStep	Wah-Geschwindigkeit

Die Implementation des Wah-Effektes auf dem DSP-Board ist der zweiten MatLab-Implementation (siehe 3.2.6.2) sehr ähnlich. Die Filterkoeffizienten-Matrix wird in einer separaten Funktion (`calcWahFilterCoeffs`) berechnet, auf die hier nicht eingegangen wird, da sie die Filterkoeffizienten auf die gleiche Art berechnet wie unter 3.2.6.2.
Die Geschwindigkeit, mit der sich das Peaking-Filter im Frequenzbereich hin- und herbewegt, hängt hier nicht allein von der Höhe der Eingangsamplitude ab, sondern kann vor allem durch die Variable `nStep` beeinflusst werden. Die Variable `nStep` lässt sich über einen GEL-Slider steuern.

```
if(choseWah)
{
    int i;
    step = (double)(nStep/100.0);

    for (i = 0; i < BLOCK_SIZE; i++)
    {
       /* read input */
       x0 = ((workbuf[i]).ch[LEFT]);

       // y0 = (b0/a0)*x0+(b1/a0)*x1+(b2/a0)*x2-(a1/a0)*y1-(a2/a0)*y2;
       y0 = (     filter_coefs[(int)cur_coef][0] * x0
                + filter_coefs[(int)cur_coef][1] * x1
                + filter_coefs[(int)cur_coef][2] * x2
                - filter_coefs[(int)cur_coef][3] * y1
                - filter_coefs[(int)cur_coef][4] * y2 );

       y2 = y1;
       y1 = y0;
       x2 = x1;
       x1 = x0;

       if(y0 > 32767)
       y0 = 32767;
       else if(y0 < -32768)
       y0 = -32768;

       /* write output */
       (workbuf[i]).ch[LEFT] = y0 * 0.5;

       cur_coef = cur_coef + sign * step * (abs(x0)/1000.0);

       if(cur_coef > max_coef)
       {
          cur_coef = max_coef;
          sign = -sign;
       }
       else if(cur_coef < min_coef)
       {
          cur_coef = min_coef;
          sign = -sign;
       }
   }
}
```

7.4 Stabilitätsproblem

Die einzelnen Effekte funktionieren prinzipiell einwandfrei, doch benötigt unser Code bei unglücklicher Zusammenschaltung mehrerer Effekte, z.B. Chorus und Reverb, zu lange für die Ausführung, was bei einer Realtime-Anwendung fatal ist! Das Ergebnis ist in unserem Fall ein unschönes, lautes Rauschen, welches dem Signal überlagert wird. Es verschwindet (meistens) auch nicht, wenn danach die verursachenden Effekte wieder ausgeschaltet werden.

Um herauszufinden, welcher Code-Teil die meiste Zeit benötigt, wollten wir den Profiler benutzen, der mittels Simulation errechnet, was wie lange ausgeführt wird. Mit Code in einer Interrupt-Serviceroutine scheint er allerdings Mühe zu haben. Es war uns nicht möglich, dem Profiler irgendwelche aussagekräftigen Daten zu entlocken.

Eine Idee, um die Ausführung des Codes zu beschleunigen, war, dass wir die Hilfsfunktionen, die relativ oft aufgerufen werden, inline programmieren. Da in C das C++ Schlüsselwort inline allerdings nicht existiert, konnten wir lediglich auf die Compileroptimierung o3 hoffen, die verspricht, „kleine" Funktionen selbstständig inline zu programmieren. Dies führte allerdings noch nicht zur Behebung des Problems.
Für weitere Nachforschungen hatten wir leider keine Zeit mehr.

8 Benutzung

Wie das Effektgerät schnell und ohne grosse Vorkenntnisse benutzt werden kann, wenn alle benötigte Hard- und Software vorhanden ist, wird nachfolgend beschrieben.

8.1 Voraussetzungen

- TMS320C6711 DSP Starter Kit von Texas Instruments
- PCM3003 Audio Daughter Card von Texas Instruments
- Personal Computer
- Elektrogitarre + Anschlusskabel
- Verstärker
- Ev. Übergangsstecker, Verbindungskabel zu Verstärker
- Effektgerät-Quellcode

8.2 Inbetriebnahme

Hardware vorbereiten:

- Audio Daughtercard auf DSP-Board aufstecken, sodass sich die beiden Buchsen am Rand des DSP-Boardes befinden (siehe Abbildung 4-1, DSP-Board mit Daughtercard)
- Jumper 1 und 2 auf Daughtercard entfernen (Mikrofon deaktivieren)
- Board via Parallelkabel mit PC verbinden
- Gitarre an Eingang, Verstärker am Ausgang anschliessen
- Speisung anschliessen

Software starten:

- Code Composer Studio starten
- Project ➔ Open ➔ im entsprechenden Arbeitsverzeichnis effectboard.pjt auswählen
- Im Unterfenster links mit der rechten Maustaste auf das Ordnersymbol „GEL files“ klicken und „Load GEL“ klicken
- Im Arbeitsverzeichnis das GEL-File „param_control.gel auswählen
- Project ➔ Rebuild All
- File ➔ Load Program...
- Im Debug-Ordner des Arbeitsverzeichnisses PCM_EDMA.out auswählen
- Debug ➔ Run

Effekte auswählen, Parameter einstellen:

- Unter GEL ➔ Parameter sind die verschiedenen Slider zu finden, um die Effekte einzuschalten oder die Parameter zu verändern.
- **ACHTUNG:** Beim Zusammenschalten mehrerer Effekte kann es zu einem unschönen Rauschen kommen, welches unter Umständen auch bei Ausschalten des verursachenden Effekts bleibt (siehe auch 7.4). In diesem Fall hilft nur ein erneutes Laden des Programmes: File ➔ Reload Program

9 Schlussfolgerungen

9.1 Rückblick auf Arbeitsverlauf

Zu Beginn investierten wir relativ viel Zeit in die Suche nach Informationen über die Funktionsweise der verschiedenen Effekte. Dies war sinnvoll, da dieses grundsätzliche Verständnis für die gesamte Arbeit ein wichtiges Fundament darstellte. Leider waren gute und vor allem präzise Informationen nur äusserst schwer zu finden.
Erschwerend kam hinzu, dass wir zwar wussten, wie gewisse Filter funktionieren sollten, wir aber von der Theorie noch nicht weit genug waren, um auch etwas Vernünftiges damit anstellen zu können. Vor allem die Tatsache, dass sich die Filter zeitlich ständig ändern müssen, hat uns ziemlich den Kopf zerbrochen.

Das Experimentieren in MatLab brachte uns zwar etwas praktische Erfahrung in der Programmierung der verschiedenen Effekte, doch die Tatsache, dass dort keine Real-Time-Bedingungen herrschen, reduzierte den Erfahrungswert. Da sich die Lieferung der DSKs verzögerte, waren wir jedoch dazu gezwungen, weiter in MatLab zu arbeiten, um unsere Ideen auch hören zu können. Hätten wir früher mit der Implementation auf dem DSP-Board beginnen können, so hätten wir vermutlich noch etwas mehr realisieren können.

Was uns ziemlich viel Zeit gekostet hat, waren die teilweise nicht nachvollziehbaren Fehler, die bei der Code-Ausführung auftraten. Teils haben wir diese selbst verursacht, teils spielte uns das CCS üble Streiche. Das grösste Problem, das Timing, konnten wir bis zum Schluss nicht beheben.

9.2 Persönlicher Rückblick

Grundsätzlich war die Semesterarbeit für uns eine interessante Erfahrung. Dass wir eine Arbeit machen durften, die uns persönlich interessierte, war natürlich ein erheblicher Motivationsfaktor. Als die ersten Effekte wunschgemäss funktionierten, war die Freude auch entsprechend gross. Ernüchterung kehrte jedoch ein, als wir feststellen mussten, dass wir mit unserem bescheiden anmutenden Code den hochgepriesenen DSP scheinbar überforderten.
Ebenfalls etwas frustrierend war, dass wir mit unserem theoretischen Wissen bezüglich Filterdesign noch nicht so weit waren, wie dies die Situation manchmal erfordert hätte.
Nichts desto trotz war die Arbeit äusserst lehrreich und hat uns mehrheitlich Spass bereitet.

9.3 Dank

Wir möchten uns an dieser Stelle herzlich bei Peter Roffler bedanken, der uns bei den Messungen stets freundlich unterstützt hat. Ein Merci geht auch an unsere toleranten Laborgenossen, welche unsere manchmal etwas ausgedehnten Effekt-Tests geduldet haben.

Ein ganz grosses Dankeschön geht natürlich an unseren Betreuer Guido M. Schuster, der sich stets unseren Problemen angenommen und diese Semesterarbeit überhaupt erst möglich gemacht hat.

10 Quellen

Erklärung von Effekten:

[1] http://www.harmony-central.com/Effects/effects-explained.html Stand 8.11.2003

[2] http://users.chariot.net.au/~gmarts/fx-desc.htm Stand 8.11.2003

[3] http://www.stagebeat.co.uk/product.php?product_id=1031 Stand 11.02.2004

C-Sourcecode für Effekte auf PC-Basis:

[4] http://st.karelia.ru/%7Esmlalx/ Stand 17.11.2003

Technische Dokumentationen von Texas Instruments sind unter
http://focus.ti.com/docs/prod/folders/print/tms320c6711.html
zu finden.
Für uns wichtig waren vor allem folgende Dokumente:

[5] TMS320C6000 DSP EDMA Controller Reference Guide (spru234)

[6] McBSP Reference Guide (spru580b)

[7] DSP/BIOS, RTDX and Host-Target Communications (spra895)

11 Anhang

11.1 Abbildungsverzeichnis

Abbildung 2-1, Chorus-Prinzip, Quelle [1] ... 6
Abbildung 2-2, Delay-Prinzip, Quelle [1] ... 6
Abbildung 2-3, Distortion-Prinzip, Quelle [2] ... 6
Abbildung 2-4, Darstellung des Halleffektes ... 6
Abbildung 2-5, Dunlop CryBaby Pedal, Quelle [3] ... 6
Abbildung 2-6, Frequenzgang des Dunlop CryBaby in beiden Extremstellungen ... 6
Abbildung 2-7, Phaser-Prinzip ... 6
Abbildung 3-1, Verstärkungskurve nach Gauss ... 15
Abbildung 3-2, Pol/Nullstellen-Diagramm ... 15
Abbildung 3-3, Peaking-Filter ... 6
Abbildung 3-4, Amplitudengänge des Peaking-Filters ... 6
Abbildung 4-1, DSP-Board mit Daughtercard ... 6
Abbildung 4-2, Gitarrenmessungen ... 6
Abbildung 4-3, Zusatzhardware ... 6
Abbildung 5-1, Signalweg ... 24
Abbildung 5-2, Eventparameter-Eintrag, Quelle [5] ... 6
Abbildung 6-1, GEL-Datei laden ... 6
Abbildung 7-1, Raised-Cosine-Verstärkungskurve ... 6

11.2 Zusätzliche Dokumente/Medien

- Aufgabenstellung
- CD-ROM mit Source-Code und dieser Dokumentation