Signalprozessor

Equalizer

alternative Prüfungsleistung

Jürgen Döffinger (631551)

03. Januar 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Entwurf

1.1 Konzept

In dieser Dokumentation werden die einzelnen Phasen vom Entwurf bis zur Implementierung des Equalizer in das Evaluation-Board ADSP-21369 SHARC EZ-KIT Lite Evaluation Kit erläutert. Zunächst möchte ich auf den Entwurf und die damit verbundene Idee eingehen.

Die Idee zu einem Equalizer war bei mir schon länger vorhanden und durch das Fach Signal- und Systemtheorie verstärkt wurden. Wir behandelten dort die Faltung im Zeitbereich und die entsprechende Multiplikation im Bildbereich. Da im Bildbereich die Amplitude über der Frequenz abgebildet ist und durch eine Multiplikation die Verstärkung bzw. Dämpfung eingestellt werden kann, war die Idee so einen Filter zu bauen. Zunächst dachte ich daran, dass Eingangssignal mittels DFT in den Bildbereich zu transformieren, dort mit einem vorgegebenen Filter zu multiplizieren und dann mittels inverser DFT in den Zeitbereich zu transformieren, womit sich eine Verstärkung bzw. Dämpfung bestimmter Frequenzen einstellen lässt.

Bei weiterer Überlegung ergab sich, dass dies nicht notwendig ist, da die Multiplikation im Bildbereich der Faltung im Zeitbereich entspricht. Es müsste nur, bei Veränderung der Filterwerte, das Filtersignal einer inversen DFT unterzogen werden, damit es dann mit dem Audiosignal gefaltet werden kann.

Bei weiterer Beschäftigung mit dem Thema Filter, ergab sich, dass der FIR (Finite Impulse Response) Filter das gleiche Konzept verfolgt. Das Konzept sieht vor, dass mehrere Filter kombiniert werden. Dies wird durch die Addition der Filterkoeffizienten erreicht. Des Weiteren müssen die Koeffizienten des jeweiligen Filters mit einem Faktor multipliziert werden, um eine Verstärkung bzw. Dämpfung des vom Filter gefilterten Frequenzbereiches zu erreichen.

Des Weiteren muss eine Lösung für ein MMI (Man-Machine Interface / Mensch-Maschine-Schnittstelle) gefunden werden. Das Board verfügt über acht LEDs und vier Taster. Relativ schnell stand für mich ein Konzept fest, welches vorsieht, das zwei Taster den Filter auswählen (Vor- und Zurück) und mit den anderen Beiden der Wert verändert werden soll. Die LEDs sollen dabei jeweils den gewählten Filter bzw. den eingestellten Wert anzeigen. Da nur acht LEDs zur Verfügung stehen, soll der jeweilige Zahlenwert als Binärwert dargestellt werden.

Ein anderer Aspekt ist die Frage, welcher Eingang soll genutzt werden und welcher Ausgang. Womit sich auch die Frage stellt, ob analoge oder digitale Signale verarbeitet werden sol-

len. Da ich erstmal nur analoge Audiosignale verarbeiten möchte, bleibt nur ein Anschluss übrig. Es ist der einzige analoge Eingang. Da ich kein konkretes Konzept für eine bestimmte Anordnung von Lautsprechern (z.B. 7.1 System usw.) oder Kodierungen verfolge, wird das bearbeitete Signal auf allen analogen Ausgängen gleich ausgegeben. Das Einzige das zu Beachten ist, dass Stereo-Signale verarbeitet werden sollen.

1.2 Machbarkeit

Bei der Machbarkeit, stellen sich folgende Fragen:

- Mit welcher Anzahl an Koeffizienten ist die Faltung innerhalb der Abtastzeit durchführbar?
- Wie hoch kann mit den berechneten Koeffizienten aufgelöst werden?
- Wie viele Koeffizienten können mittels des Speichers effektiv verarbeitet werden?
- Wie viele Filterbereiche können angelegt werden?

1.2.1 Mit welcher Anzahl an Koeffizienten ist die Faltung innerhalb der Abtastzeit durchführbar?

Vernachlässigt man das Abholen der Eingangssignalwerte für den linken bzw. rechten Kanal und die Ausgabe der berechneten Werte (sind nur wenige Taktzyklen) und konzentriert man sich auf die Berechnung der Faltung, so ergeben sich folgende Berechnungen:

$$adders = N \tag{1}$$

$$multipliers = N + 1 \tag{2}$$

Geht man weiterhin von der Gleichung

$$B_a = \frac{f_{clock}}{f_a} \tag{3}$$

aus, wobei B_a für die Anzahl der Berechnungen zwischen den Abtastungen steht, f_{clock} für die Taktfrequenz des Prozessors und f_a für die Abtastfrequenz, so ergeben sich

$$B_a = \frac{400\ MHz}{48\ kHz} \tag{4}$$

$$B_a = 8333\frac{1}{3}\ Berechnungen\ pro\ Abtastung \tag{5}$$

rund 8333 Berechnungen pro Abtastung.

Für die Anzahl der Berechnungen pro Faltung, kann folgende Gleichung herangezogen werden.

$$B_F = adders + multipliers \tag{6}$$
$$B_F = N + N + 1 \tag{7}$$

Wobei N die Anzahl der Koeffizienten und B_F die Anzahl Berechnungen pro Faltung ist. Mit (3) und (7) ergibt sich eine Gleichung zur Berechnung der Anzahl Faltungen pro Abtastung.

$$n_F = \frac{B_a}{B_F} \tag{8}$$
$$n_F = \frac{\frac{f_{clock}}{f_a}}{N + N + 1} \tag{9}$$
$$n_F = \frac{f_{clock}}{f_a(2N + 1)} \tag{10}$$

Setzt man nun die Anzahl Faltungen pro Abtastung $n_F = 1$ ein und stellt nach N um, so ergibt sich die Berechnung der Anzahl Koeffizienten pro Abtastung.

$$N = \frac{\left(\frac{f_{clock}}{f_a} - 1\right)}{2} \tag{11}$$

Setzt man weiterhin die Werte für das Board ein, so kommt man auf folgende Gleichung.

$$N = \frac{\left(\frac{400\ MHz}{48\ kHz \cdot 1} - 1\right)}{2} \tag{12}$$

$$N = 4166\frac{1}{6}\ \ \frac{Koeffizienten}{Abtastung} \tag{13}$$

Geht man noch von ein paar Taktzyklen für die Abholung des Eingangswertes und der Ausgabe des Ausgangswertes, sowie der Einstellung von Adressgeneratoren usw. aus, so lässt sich auf $N = 4096$ Koeffizienten runden.

1.2.2 Wie hoch kann mit den berechneten Koeffizienten aufgelöst werden?

Im vorherigen Abschnitt habe ich die Anzahl der möglichen Koeffizienten für den FIR Filter berechnet. Es waren N = 4096. Im folgenden soll nun geklärt werden, wie hoch der Filter den abgetasteten Frequenzbereich auflösen kann.

Dabei ist von einer Abtastfrequenz $f_a = 48\ kHz$ auszugehen. Mit dieser Abtastfrequenz lässt sich die Auflösung wie folgt berechnen.

$$A_f = \frac{f_a}{N} \tag{14}$$

$$A_f = \frac{48000\ Hz}{4096} \tag{15}$$

$$A_f = 11,71875\ Hz \tag{16}$$

Der Frequenzbereich mit der Abtastfrequenz $f_a = 48$ kHz lässt sich also auf rund $A_f = 12$ Hz auflösen.

An dieser Stelle stellt sich eine weitere Frage. Ist eine Auflösung von 12 Hz notwendig? Ich meine nicht unbedingt. Hier kommt es vermutlich darauf an, für was der Equalizer verwendet werden soll. Ich habe mich für eine Auflösung von 100 Hz entschieden, da ich mit diesem Projekt erst einmal einen Equalizer umsetzen möchte, ohne eine bestimmte Anwendung zu verfolgen. Mir geht es im Moment nur um die prinzipielle Umsetzung. Es ist auch zu bedenken, dass mit einem FIR - Filter und der großen Anzahl an Koeffizienten sich eine große Verzögerung zwischen dem Eingang eines Signalwertes und dessen berechnete Ausgabe entsteht.

$$t_V = \frac{N}{f_a} \tag{17}$$

$$t_V = \frac{4096}{48000\ Hz} \tag{18}$$

$$t_V = 85\frac{1}{3}\ ms \tag{19}$$

Es ist also zu sehen, dass N = 4096 Koeffizienten eine Verzögerung von $t_V = 85\frac{1}{3}$ ms hervorrufen. Dabei kann die Verzögerung durch die Wandlung von analog zu digital und digital zu analog vernachlässigt werden, da diese bei den ADC bzw. DAC des AD1835 im Nanosekundenbereich liegen.

Daher auch von mir die Entscheidung zu einer Auflösung von 100 Hz, was mit (14) umgestellt nach N eine Anzahl von N = 480 Koeffizienten ergibt. Um bei einer Koeffizientenzahl zu bleiben, welche einer Zweierpotenz entspricht, habe ich mich für 512 Koeffizienten entschieden, welche eine Verzögerung nach (17) von rund $t_V = 11$ ms ergibt. Im Zusammenhang mit einem Videosignal, liegt dies noch in einem erträglichen Maß. Allerdings sollte man sich überlegen, ob dieser Equalizer in Verbindung mit einem Videosignal, welches nicht entsprechend verzögert wird, geeignet ist? Meiner Ansicht nach nicht, denn die Verzögerung ist zu bemerken. Mit der Koeffizientenanzahl von N = 512 ergibt sich eine Auflösung von $A_f = 93,75$ Hz.

Zusammengefasst soll der EQ folgende Eigenschaften aufweisen.

- N = 512
- $A_f = 93,75$ Hz
- $t_V = 11$ ms

1.2.3 Wie viele Koeffizienten können mittels des Speichers effektiv verarbeitet werden?

Der ADSP-21369 hat Intern einen RAM von 2 MBit. Die Koeffizienten sind 32 bit - Werte. Daraus ergibt sich eine Möglichkeit

$$n_M = \frac{2Mbit}{32bit} \tag{20}$$

$$n_M = 1048576\ Koeffizienten \tag{21}$$

1048576 Koeffizienten abzuspeichern. Dies ist aber nicht notwendig, denn geht man von 512 Koeffizienten aus, so muss nur noch die Anzahl Filter festgelegt werden. Selbst bei typischen Werten von 10 oder 32 Filtern wird dieser Wert nicht erreicht. Es bleibt also noch genügend Speicher für andere Variablen und dem Programm selber.

1.2.4 Wie viele Filterbereiche können angelegt werden?

Aufgrund der Vorausgegangenen Berechnung habe ich mich für 11 Filter entschieden. Dies ergibt einen Speicherbedarf von

$$512\ Koeffizienten \cdot 32\frac{bit}{Koeffizienten} \cdot (11\ Filter + 1\ Gesamtfilter) = 196608\ bit$$

196608 bit. Dies ist wesentlich weniger als der zur Verfügung stehende Speicher von 2 Mbit. 11 Filter und 1 Gesamtfilter sind also realisierbar.

Warum aber 11 Filter und 1 Gesamtfilter?

Dahinter steht die Idee 10 einstellbare Filter zu verwenden und 1 Filter für den oberen Frequenzbereich, welcher im Grunde nicht mehr zu hören ist, zu nutzen. Der Gesamtfilter ist der Filter welcher die Addition aller Koeffizienten der Einzelfilter enthält und ist derjenige der mit dem Eingangssignal gefaltet wird. In den 11 Filtern werden also nur die Grundkoeffizienten vorrätig gehalten, welche eine Verstärkung von 1 für den jeweiligen Frequenzbereich aufweisen.

2 Implementierung

Zunächst möchte ich an dieser Stelle noch mal zusammenfassen, welche Werte ich für den Equalizer im vorangegangenen Abschnitt festgelegt habe.

- 512 Koeffizienten je Filter
- 11 Einzelfilter
- 1 Gesamtfilter
- 48 kHz Abtastfrequenz
- Stereo - Eingangs- und Ausgangssignal
- 1 x analog Eingang und 4 x analoge Ausgänge
- LEDs zeigen binärcodiert den ausgewählten Filter bzw. den Filterwert in dB an (LSB = LED1; MSB = LED6; Vorzeichen = LED7)
- Taster PB3 und PB4 dienen der Auswahl des Filters
- Taster PB1 und PB2 dienen der Einstellung des Verstärkungsfaktors

Zunächst sei erwähnt, dass es mir darum ging die Filter und damit prinzipiell den Equalizer und das MMI umzusetzen. Daher habe ich ein Programmierbeispiel von Analog Devices und die Programmierbeispiele auf der Website von Prof. Wagner als Grundlage genommen. Aus Zeitgründen habe ich mich nicht mit der Initialisierung des ADSP-21369 SHARC EZ-KIT Lite Evaluation Kit beschäftigt und diese nur übernommen und an den notwendigen Stellen entsprechend abgeändert. Mehr dazu werde ich in den folgenden Abschnitten näher erläutern.

Das grundsätzliche Prinzip soll sein, dass mittels Interrupt dem Programm mitgeteilt wird, dass ein neuer Abtastwert vorliegt. Dieser wird der Berechnungsroutine übergeben, welche einen neuen Ausgangswert berechnet und diesen an den DAC übergibt. All dies muss natürlich innerhalb eines Abtastintervalls von rund 21 μs (Abtastfrequenz = 48 kHz) durchgeführt werden. Dabei soll die Möglichkeit des Parallelbetriebs ausgenutzt werden, womit ein Stereo-Signal gleichzeitig verarbeitet werden kann. Es sollen also die Y-Register mitverwendet werden. Daher sind auch die Koeffizienten doppelt auszulegen, womit sich der Speicherbedarf für die Koeffizienten verdoppelt und somit 393216 bit Speicher notwendig

ist. Was aber immer noch kein Problem darstellt, da für das Programm und die anderen Variablen noch genügend Speicher zur Verfügung steht.

Zur Umsetzung selber ist in einem Hauptprogramm (main) die Initialisierung des Boards vorzunehmen und in einer Endlosschleife abzuschließen, um entsprechend auf Interrupts reagieren zu können.

Auch bei Betätigung der Taster sind Interrupts auszulösen und entsprechende Interrupt Service Routinen auszuführen. Diese sollten entsprechend des betätigten Tasters den Filter auswählen bzw. den Filterwert verändern. Anschließend ist der Gesamtfilter neu zu berechnen und die Anzeige entsprechend zu verändern.

2.1 Hauptprogramm (main)

Das Hauptprogramm initialisiert, wie im vorherigen Abschnitt erwähnt, zunächst das Board und dann den Filter. Am Ende werden die entsprechenden Interrupts freigegeben und das Hauptprogramm wird nicht beendet, sondern endet in einer Endlosschleife.

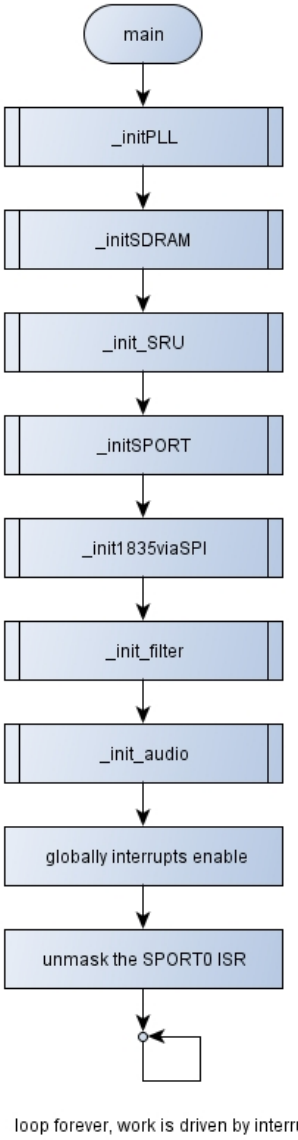


Abbildung 1: Programmablaufplan main

2.1.1 _initPLL und _initSDRAM

Diese Unterprogramme initialisieren die PLL und den SDRAM und wurden direkt aus den Beispielprogrammen übernommen, daher soll hier nur der Kommentar aus den Beispielprogrammen wiedergegeben werden.

> Sets up the SDRAM controller to access SDRAM. In this file are two subroutines, the first to set up the SHARC's PLL, and the second to set up the SDRAM controller. CLKIN= 24.576 MHz, Multiplier= 27, Divisor= 2, CCLK_SDCLK_RATIO 2.0. Core clock = (24.576MHz * 27) /2 = 331.776 MHz

2.1.2 _init_SRU

Auch hier wurde die Routine aus den Beispielprogrammen übernommen und ein wenig abgewandelt. Dazu zunächst der Programmablaufplan.

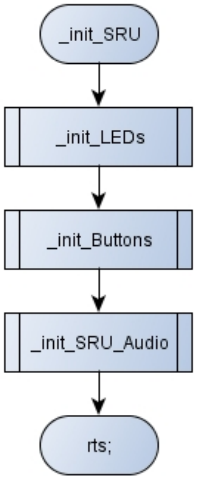


Abbildung 2: Programmablaufplan _init_SRU

2.1.2.1 _init_LEDs

Das Unterprogramm _init_LEDs initialisiert die LEDs. Dabei werden die LEDs 1 bis 5 über DPI geroutet (siehe Abbildung 3).

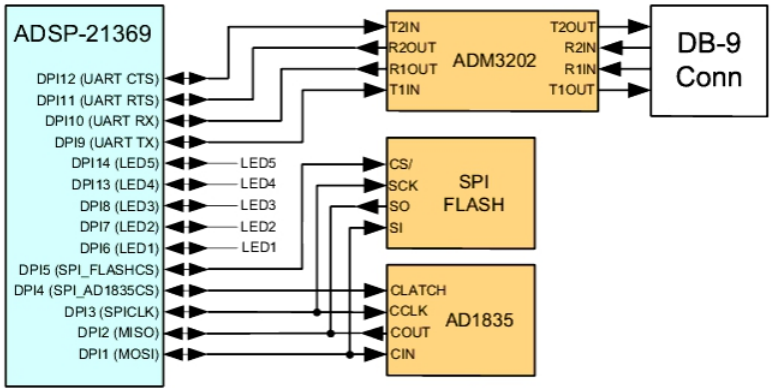


Abbildung 3: DPI Connections Block Diagram

Die LEDs 6 und 7 werden über DAI geroutet (siehe Abbildung 4).

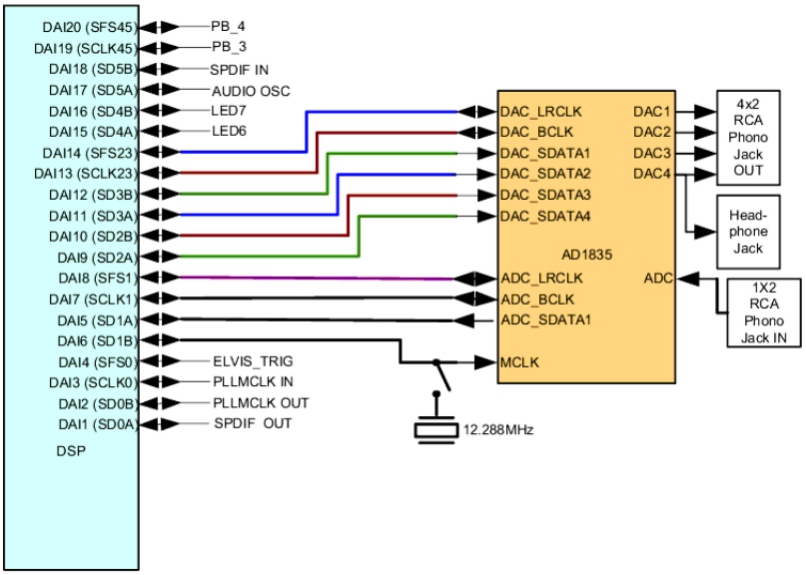


Abbildung 4: DAI Connections Block Diagram

LED8 wird nicht verwendet.

Am Ende des Unterprogramms werden die LEDs 1-7 gelöscht.

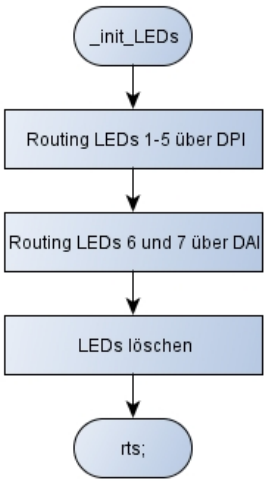


Abbildung 5: Programmablaufplan _init_LEDs

2.1.2.2 _init_Buttons

Bei der Initialisierung der Taster werden die Taster PB1 und PB2 an den IRQ0 und IRQ1 geroutet und PB3 und PB4 an den DAI-Interrupt (higher priority) geroutet. Am Ende der Routine werden die Interrupts freigegeben.

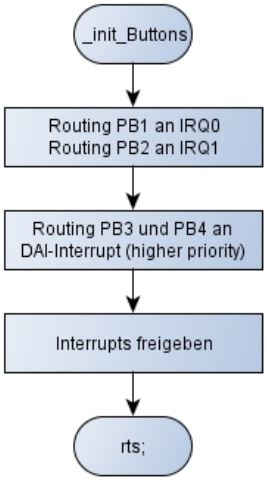


Abbildung 6: Programmablaufplan _init_Buttons

2.1.2.3 _init_SRU_Audio

Das Unterprogramm wurde aus dem Beispielprogramm von Analog Devices übernommen. Es routet die analogen Ein- und Ausgänge, sowie die Taktfrequenzen entsprechend folgendem Schema:

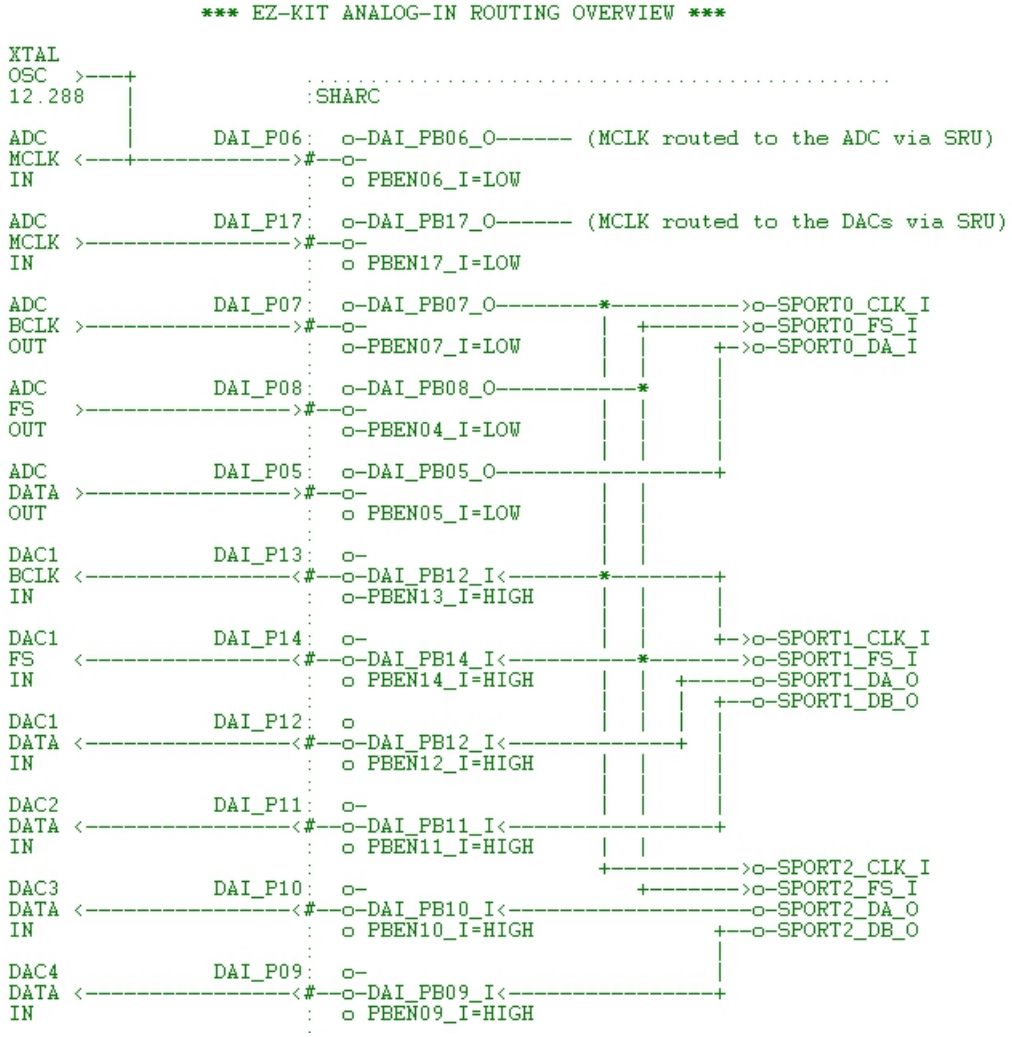


Abbildung 7: EZ-KIT ANALOG-IN ROUTING OVERVIEW

2.1.3 _initSPORT

Auch diese Routine ist direkt aus dem Beispielprogramm übernommen wurden und daher soll hier nur der Kommentar von Analog Devices zu dieser Routine wiedergegeben werden.

> This file initializes the transmit and receive serial ports (SPORTS). It uses SPORT0 to receive data from the ADC and transmits the data to the DAC's via SPORT1A, SPORT1B, SPORT2A and SPORT2B.

2.1.4 _init1835viaSPI

Wie die Routine zuvor stammt diese auch aus dem Beispielprogramm. Hier werden der ADC und die DACs des AD1835 initialisiert.

2.1.5 _init_filter

Dieses Unterprogramm initialisiert den Filter. Der Filter besteht aus 11 Einzelfiltern von denen 10 einstellbar sind. Des Weiteren wird auch der Gesamtfilter initialisiert.

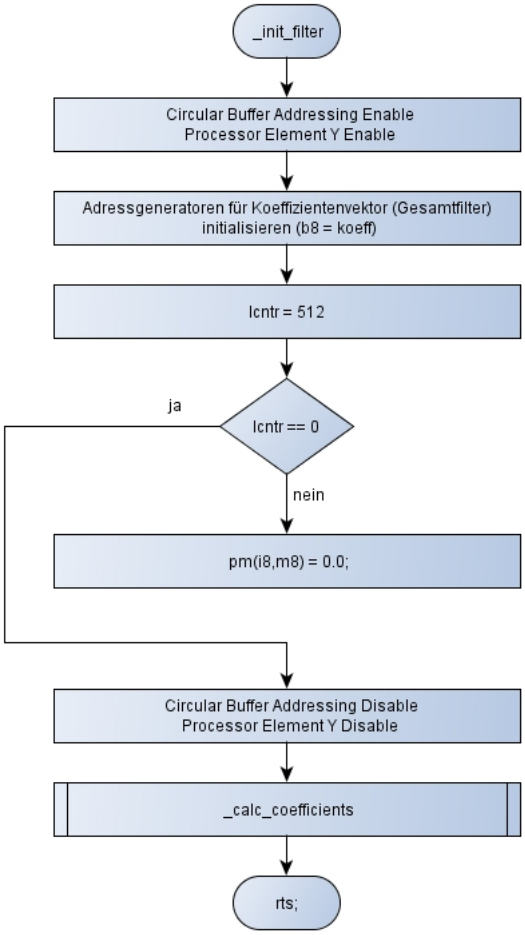


Abbildung 8: Programmablaufplan _init_filter

Zunächst wird also das Circular Buffer Addressing und der Parallelbetrieb (SIMD) eingeschaltet. Somit können die Koeffizienten für den linken und die für den rechten Kanal gleichzeitig in der folgenden Schleife gelöscht, also auf 0.0 gesetzt werden. Die Schleife löscht also den Gesamtfilter, welcher unter dem Variablenbezeichner *koeff* geführt wird. Ein weiterer Variablenbezeichner ist *koeff_orig*. In diesem sind die Koeffizienten der Einzelfilter gespeichert. Diese werden beim Linken aus der Datei *filter.dat* entnommen und in den entsprechenden Speicherbereich, welcher der Variable zugewiesen wurde, geschrieben. Diese Koeffizienten sind so eingerichtet, dass sie eine Verstärkung von 1 realisieren und dienen als Grundlage für die Berechnung des Gesamtfilters, welcher im Unterprogramm *_calc_coefficients* berechnet wird. Nach dem Löschen des Gesamtfilters wird dieses Unterprogramm aufgerufen. Zum Schluss der Routine wird das Circular Buffer Addressing und der Parallelbetrieb (SIMD) abgeschaltet.

2.1.5.1 filter.dat

In diesem Abschnitt soll die Bildung der Einzelfilter und des Gesamtfilters erläutert werden.

Die Einzelfilter wurden mithilfe des Programms *MatLab* (R2011a - Student Version) von *MathWorks* erstellt. Hierbei wurde das Plug-in *fdatool* aus der *DSP System Toolbox Simulink* verwendet und zur Bildung der Datei *filter.dat* ein eigenes MatLab-File erstellt. Ein Listing des MatLab-Files findet sich im Anhang.

Die Einzelfilter wurden in folgende Frequenzbereiche eingeteilt.

Filter	**Mitten-frequenz in Hz**	**Frequenz-bereich in Hz**
1	110	1 - 219
2	300	220 - 380
3	600	381 - 819
4	1000	820 - 1180
5	3000	1181 - 4819
6	6000	4820 - 7180
7	9000	7181 - 10819
8	12000	10820 - 13180
9	14000	13181 - 14819
10	16000	14820 - 17180
11	20590	17181 - 23999

Tabelle 1: Frequenzbereiche der Einzelfilter

Die *filter.dat* ist so aufgebaut, dass die Koeffizienten der einzelnen Filter nacheinander angeordnet sind. Dabei sind es immer zwei Werte pro Filter, mit jeweils demselben Koeffizientenwert. Dabei dient der Eine für den Linken und der Andere für den rechten Kanal. Das folgende Schema stellt den Aufbau der *filter.dat* bildlich dar. Die Koeffizienten werden mit a_{i_j} bezeichnet. Wobei i für den jeweiligen Koeffizienten steht und j der entsprechende Filter

ist.

$$
\begin{array}{c}
a_{0_1} \\
a_{0_1} \\
a_{0_2} \\
a_{0_2} \\
. \\
. \\
. \\
a_{0_{11}} \\
a_{0_{11}} \\
a_{1_1} \\
a_{1_1} \\
a_{1_2} \\
a_{1_2} \\
. \\
. \\
. \\
a_{1_{11}} \\
a_{1_{11}} \\
. \\
. \\
. \\
a_{512_{11}} \\
a_{512_{11}}
\end{array}
$$

Das entsprechende MatLab-File zur Zusammenstellung der Koeffizienten für die Datei *filter.dat* verwendet für die Bildung der Einzelfilterkoeffizienten den Befehl fir1 mit folgender Syntax.

```
fir1(N, [fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
```

Dabei ist N die Ordnung, hier 512, *fu* die untere und *fo* die obere Grenzfrequenz des Bandpasses. Der Bezeichner *bandpass* kennzeichnet, dass es sich um einen Bandpass handelt. Mit *win* wird die Fensterfunktion festgelegt. Der Bezeichner *flag* wird hier mit *scale* angegeben.

Für die Einzelfilter wird die Fensterfunktion *Blackman-Nutall* verwendet. Die Rechenvorschrift für die Koeffizienten lautet:[1]

$$\omega(n) = a_0 - a_1 \cos\left(2\pi \frac{n}{N-1}\right) + a_2 \cos\left(4\pi \frac{n}{N-1}\right) - a_3 \cos\left(6\pi \frac{n}{N-1}\right) \tag{22}$$

$$mit\ n = 0, 1, 2, ...N-1$$

$$a_0 = 0,3635819$$
$$a_1 = 0,4891775$$
$$a_2 = 0,1365995$$
$$a_3 = 0,0106411$$

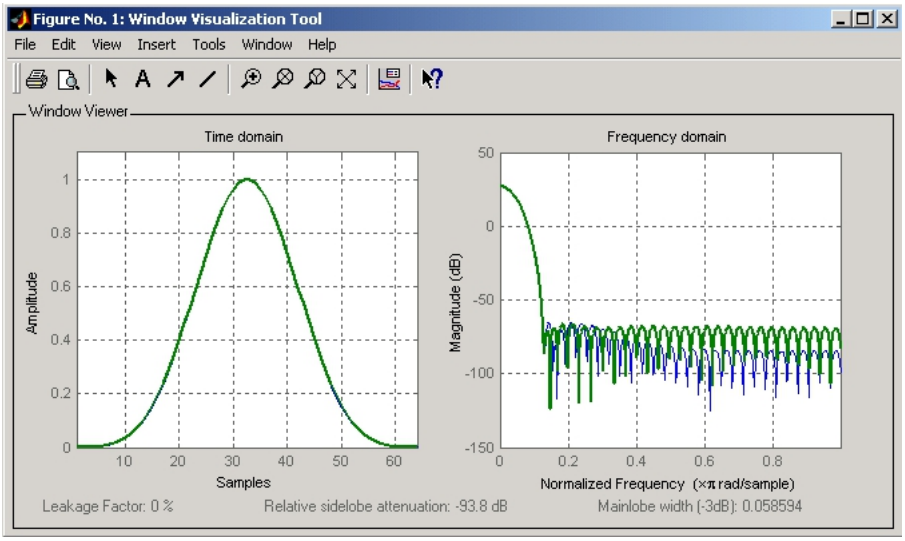


Abbildung 9: Blackman-Nuttall Fensterfunktion

[1] http://www.mathworks.de/help/toolbox/signal/ref/nuttallwin.html

Mit der Abtastfrequenz fa = 48000 Hz und der Ordnung N = 512 werden die Koeffizienten der Einzelfilter wie folgt berechnet:

```
fu = 1;      fo = 219;
f1 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 220;    fo = 380;
f2 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 381;    fo = 819;
f3 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 820;    fo = 1180;
f4 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 1181;   fo = 4819;
f5 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 4820;   fo = 7180;
f6 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 7181;   fo = 10819;
f7 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 10820;  fo = 13180;
f8 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 13181;  fo = 14819;
f9 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 14820;  fo = 17180;
f10 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 17181;  fo = 23999;
f11 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
```

Der Inhalt der Datei *filter.dat* wird über folgenden Code zusammengestellt:

```
l = 1;
for k = 1:N+1
    fx(1,l)   = f1(1,k);
    fx(1,l+1) = f1(1,k);
    l = l + 2;
    fx(1,l)   = f2(1,k);
    fx(1,l+1) = f2(1,k);
    l = l + 2;
    .
    .
    fx(1,l)   = f11(1,k);
    fx(1,l+1) = f11(1,k);
    l = l + 2;
end
```

Zunächst werden also einzelnen Koeffizienten dupliziert und anschließend in richtiger Reihenfolge in die Matrix fx gebracht. Diese Matrix wird dann in der Datei *filter.dat* gespeichert.

```
dlmwrite('..\Filter\filter.dat',fx,'delimiter','\n','precision','%.15f');
```

Die einzelnen Diagramme zu den Einzelfiltern und dem Gesamtfilter finden sich im Anhang.

2.1.5.2 _calc_coefficients

Im Unterprogramm *_calc_coefficients* wird die Verstärkung der einzelnen Filter und somit der Frequenzbereiche eingestellt und daraus der Gesamtfilter berechnet.

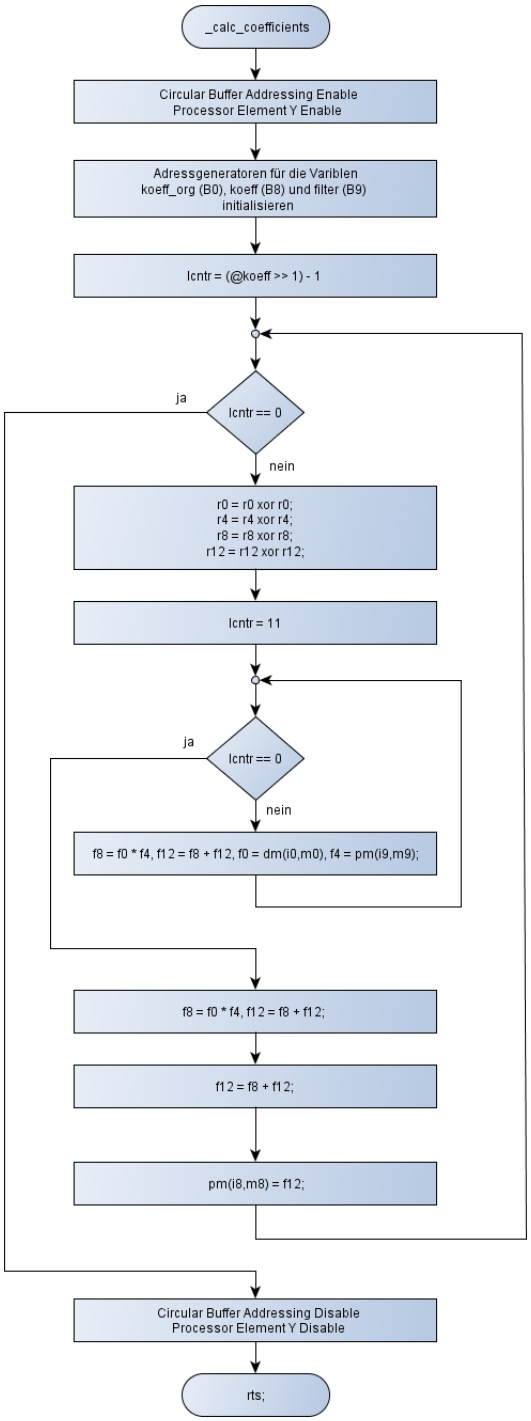


Abbildung 10: Programmablaufplan _calc_coefficients

Als Erstes wird das Circular Buffer Addressing eingeschaltet und das Processor Element Y zugeschaltet.

Der nächste Schritt ist das Initialisieren der Adressgeneratoren. Dabei werden B0 die Original-Koeffizienten der Einzelfilter (*koeff_orig*), B8 die Koeffizienten des Gesamtfilters (*koeff*) und B9 die Verstärkungsfaktoren (*filter*) der Einzelfilter zugewiesen.

```
B0      = koeff_orig;
M0      = 2;
L0      = N*2*11;

B8      = koeff;
M8      = 2;
L8      = N*2;

B9      = filter;
M9      = 2;
L9      = @filter;
```

Nun folgt die Berechnung des Gesamtfilters, welche über folgende Rechenvorschrift vorgenommen wird:

$$a(n) = \sum_{F=1}^{11} a_F(n) \cdot v_F \tag{23}$$

$$mit\ n = 0, 1, 2...512$$

Wobei $a(n)$ die Koeffizienten des Gesamtfilters, $a_F(n)$ die Koeffizienten der Einzelfilter und v_F die Verstärkung des jeweiligen Filter repräsentieren. F gibt den Filter und n den Koeffizienten an.

Es werden also die Koeffizienten einer bestimmten Stelle aller Filter summiert und als entsprechender Koeffizient des Gesamtfilters abgespeichert. Wobei jeder Koeffizient mit dem Verstärkungsfaktor des jeweiligen Filters multipliziert wird.

```
r1 = (@koeff >> 1) -1;
lcntr = r1, do _calc_coefficients_loop_1 until lce;

        r0 = r0 xor r0;
        r4 = r4 xor r4;
        r8 = r8 xor r8;
```

```
                r12 = r12 xor r12;

                lcntr = 11, do _calc_coefficients_loop_2 until lce;
                _calc_coefficients_loop_2:
                f8=f0*f4, f12=f8+f12, f0=dm(i0,m0), f4=pm(i9,m9);

                f8 = f0 * f4, f12 = f8 + f12;
                f12 = f8 + f12;

_calc_coefficients_loop_1: pm(i8,m8) = f12;
```

Im vorangegangenen Quellcode sind zwei Schleifen zu sehen. Die erste Schleife geht die einzelnen Koeffizienten durch und die zweite führt die Multiplikationen und Additionen aus. Dabei wird nicht nur der Parallelbetrieb, zur gleichzeitigen Berechnung der Koeffizienten für den linken und rechten Kanal, ausgenutzt, sondern auch die Möglichkeit mehrere Operationen gleichzeitig auszuführen. Daher müssen nach dem Beenden der zweiten Schleife noch die beiden Zeilen

```
f8 = f0 * f4, f12 = f8 + f12;
f12 = f8 + f12;
```

ausgeführt werden, da diese Berechnungen, während des letzten Schleifendurchgangs, noch nicht ausgeführt wurden.

Am Ende des Unterprogramms wird das Circular Buffer Addressing abgeschaltet und das Processor Element Y gesperrt.

2.1.6 _init_audio

Das Unterprogramm *_init_audio* initialisiert den Adressgenerator B5 für den Puffer (*buffer*). Der Puffer dient dazu die jeweiligen letzten 513 Eingangswerte des linken und rechten Kanals zwischenzuspeichern, um Sie für die Faltung abrufen zu können. Da die Routine so klein ist, verzichte ich an dieser Stelle auf einen Programmablaufplan und werde hier nur das Listing des Quellcodes anfügen.

```
_init_audio:

        B5 = buffer;
        M5 = 2;
        L5 = N*2;

_init_audio.end: rts;
```

2.2 Interrupt Service Routinen (ISR)

Nach der Initialisierung des Boards und des Prozessors durch das Hauptprogramm *main* reagiert das Programm nun nur noch auf Interrupts. Die zugehörigen Routinen werde ich im folgenden näher beschreiben.

2.2.1 _button1

Über den Interrupt IRQ1I wird die ISR *_button1* aufgerufen. Der Interrupt wird ausgelöst, wenn der Taster PB1 gedrückt wird. Das Unterprogramm ist dafür zuständig den nächsthöheren Verstärkungsfaktor für den gerade ausgewählten Filter einzustellen.

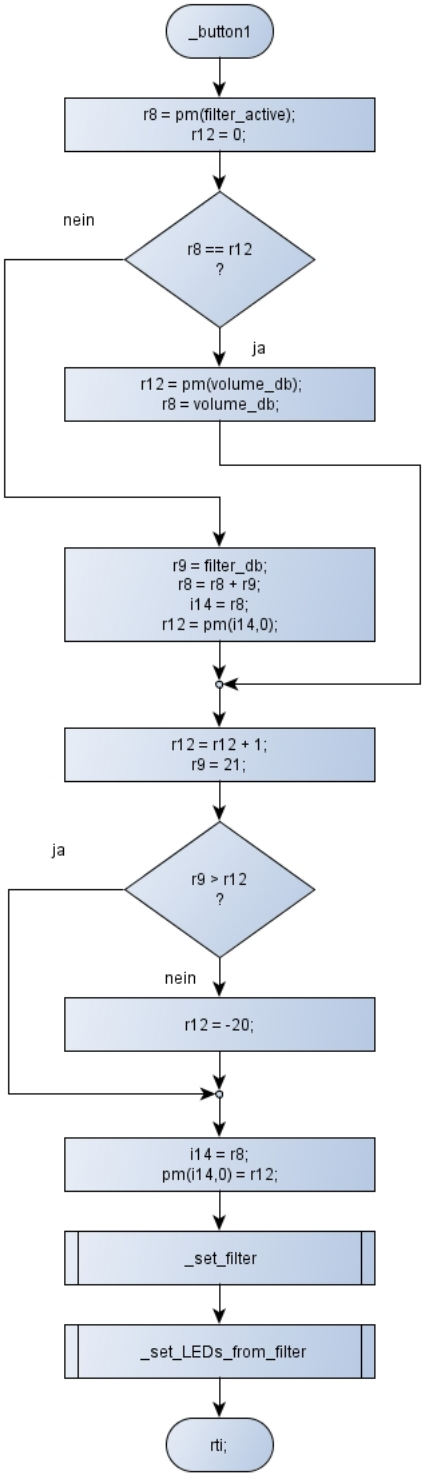


Abbildung 11: Programmablaufplan _button1

Als Erstes wird im Register R8 der derzeit ausgewählte Filter gespeichert. Anschließend wird geprüft, ob es sich bei dem Wert um eine Null handelt. Die Null kennzeichnet keinen Filter, sondern den Vorverstärker. Der Vorverstärker dient dazu das Eingangssignal, noch vor der Faltung, zu bedämpfen bzw. zu verstärken. Ist aktuell der Vorverstärker ausgewählt, so wird dessen db-Wert um eins erhöht und in der Variable *volume_db* gespeichert. Ist dagegen ein Filter ausgewählt, so wird der Index des Adressgenerators B14 auf die Adresse des entsprechenden Filterwertes gesetzt. Der Filterwert ist in der Variable *filter_db* gespeichert. Anschließend wird dieser Wert inkrementiert. Bei beiden Inkrementierungen wird geprüft, ob das Ergebnis über 21 ist und wenn dies der Fall ist, wird der dB-Wert auf -20 gesetzt. Erreicht der Anwender beim mehrmaligen betätigen der Taste die 20 dB und betätigt er den Taster erneut, so springt der Wert auf -20 dB. Alle Filter und der Vorverstärker können auf einen Wert im Bereich von -20 dB bis 20 dB in 1er-Schritten eingestellt werden.

Zum Schluss der Routine werden die Unterprogramme *_set_filter*, zur Neuberechnung des Gesamtfilters und *_set_LEDs_from_filter*, zum Einstellen der Anzeige, aufgerufen.

2.2.1.1 _set_filter

Das Unterprogramm *_set_filter* wandelt den vom Anwender ausgewählten dB-Wert des aktuell gewählten Filters in einen Verstärkungsfaktor und speichert diesen. Zusätzlich wird die Funktion *_calc_coefficients* aufgerufen, um den Gesamtfilter neu zu berechnen.

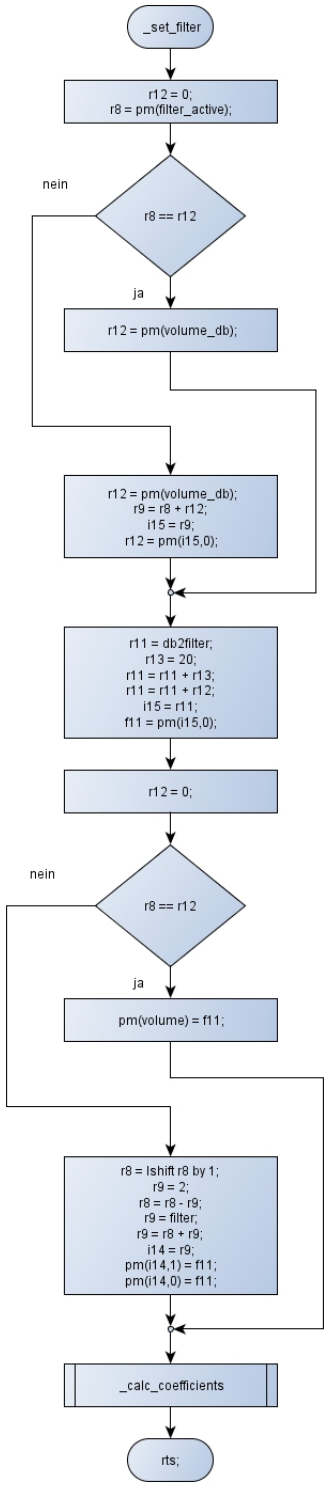


Abbildung 12: Programmablaufplan _set_filter

Das Unterprogramm ermittelt als Erstes ob es sich um den zu wandelnden Wert des Vorverstärkers oder einer der Einzelfilter handelt. Je nachdem, wird der gewandelte Wert in der Variable *volume*, für den Vorverstärker oder *filter* für einen der Einzelfilter, abgelegt.

Die Wandlung ist für alle gleich. In der Variablen *db2filter* ist eine Lookup-Table (LUT) hinterlegt. Die LUT wird beim Linken mit den Werten aus der Datei *db2filter.dat* gespeist. Dabei ist diese Datei so aufgebaut, dass dort zeilenweise der Verstärkungsfaktor hinterlegt ist. Dabei beginnt die erste Zeile mit dem Verstärkungsfaktor der zu -20 dB gehört und die letzte Zeile beinhaltet den Verstärkungsfaktor für 20 dB. Dazwischen sind die weiteren Verstärkungsfaktoren für die jeweiligen dB-Werte in 1er Schritten aufsteigend angeordnet. Die Umrechnung erfolgt nach folgender Rechenvorschrift:

$$v = 10^{\frac{v_{dB}}{20}} \tag{24}$$

Nach der Wandlung wird der ermittelte Verstärkungsfaktor v entsprechend in *volume* oder *filter* abgespeichert und das Unterprogramm *_calc_coefficients* wird zur Berechnung des Gesamtfilters aufgerufen.

2.2.1.2 _set_LEDs_from_filter

Das Unterprogramm *_set_LEDs_from_filter* ermittelt den anzuzeigenden dB-Wert und übergibt diesen beim Aufruf an die Funktion *_set_LEDs*, welche diesen auf den LEDs 1 bis 7 abbildet.

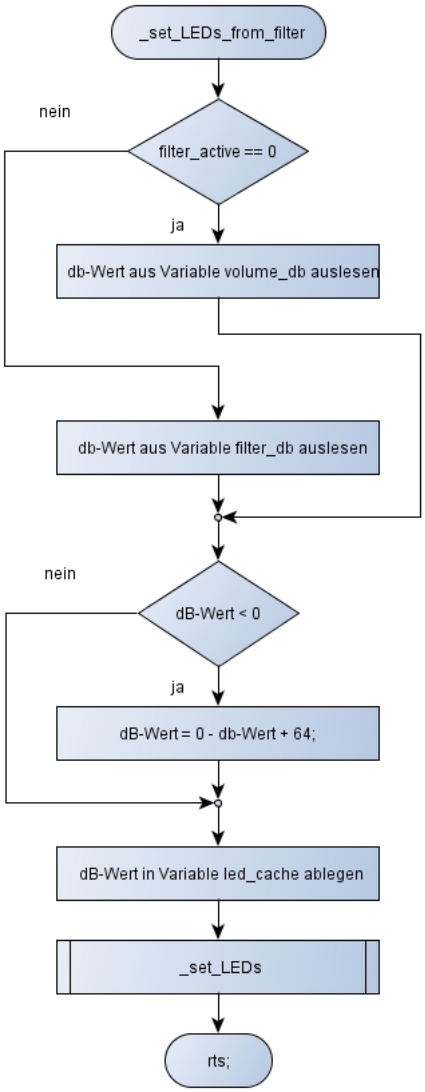


Abbildung 13: Programmablaufplan _set_LEDs_from_filter

2.2.1.2.1 _set_LEDs

Das Unterprogramm *_set_LEDs* liest den Wert aus der Variable *led_cache* aus und stellt die LEDs entsprechend ein. Zunächst werden alle LEDs mithilfe der Funktion *_LEDs_off* gelöscht. Diese Funktion wird hier nicht näher beschrieben, da sie lediglich drei Zeilen beinhaltet und selbsterklärend ist. Ein Listing findet sich im Anhang. Nach dem Löschen der LEDs werden die LEDs entsprechend den gesetzten Bits in der Variablen *led_cache* eingeschaltet. Dabei entsprechen die Bits 0 bis 6 den LEDs 1 bis 7. Die LEDs 0 bis 6 stellen den Wert als binären Zahlenwert dar und die LED 7 kennzeichnet ob es ein negativer Wert (LED leuchtet nicht) oder ob es sich um einen positiven Wert (LED löschtet) handelt.

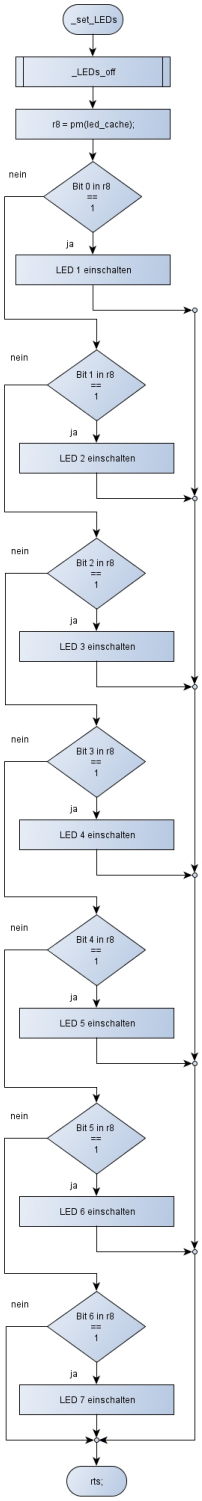


Abbildung 14: Programmablaufplan _set_LEDs_

2.2.2 _button2

Die Funktion *_button2* ist synonym zur Funktion *_button1* aufgebaut. Lediglich wird hier der dB-Wert des entsprechenden Filters oder des Vorverstärkers dekrementiert. Werden dabei -20 dB unterschritten, so wird der Wert auf 20 dB eingestellt. Des Weiteren wird die Funktion aufgerufen, wenn der Interrupt IRQ0I ausgelöst wird. Dieser wird ausgelöst wenn der Taster PB2 betätigt wird.

2.2.3 _button3or4

Die ISR *_button3or4* wird aufgerufen, wenn der Interrupt DAIHI ausgelöst wird. Dieser Interrupt wird ausgelöst wenn einer der Tasten PB3 oder PB4 betätigt wird.

Die Routine prüft welcher Taster betätigt wurde und ruft das entsprechende Unterprogramm auf.

Nach Abarbeitung des jeweiligen Unterprogramms wird die Funktion *_set_LEDs* aufgerufen, damit dem Anwender der ausgewählte Filter angezeigt wird, denn mit den Button PB3 und PB4 kann der Anwender den Filter auswählen.

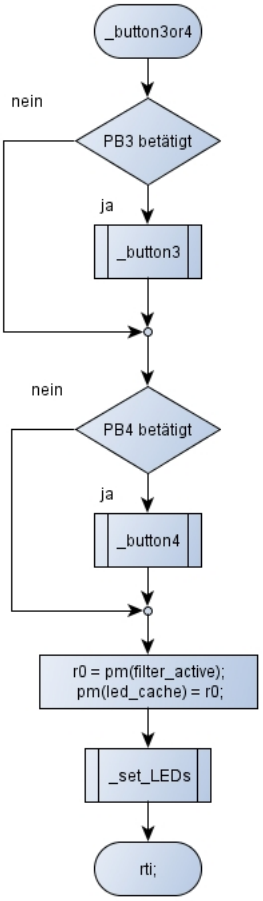


Abbildung 15: Programmablaufplan _button3or4

2.2.3.1 _button3

Mithilfe der Funktion *_button3* wird der nächste Filter ausgewählt. Dazu wird der Wert in *filter_active*, bei betätigen des Tasters PB3, inkrementiert. Überschreitet der neue Wert die Zahl 10 so wird der Wert auf 0 gesetzt.

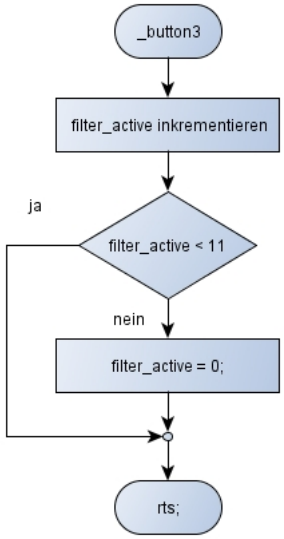


Abbildung 16: Programmablaufplan _button3

2.2.3.2 _button4

Die Funktion *_button4* ist synonym zur Funktion *_button3* aufgebaut. Hier wird lediglich der Wert von *filter_active* dekrementiert. Wird der Wert dabei kleiner als 0, so wird der Wert auf 10 gesetzt. Diese Funktion wird aufgerufen, wenn der Taster PB4 betätigt wird.

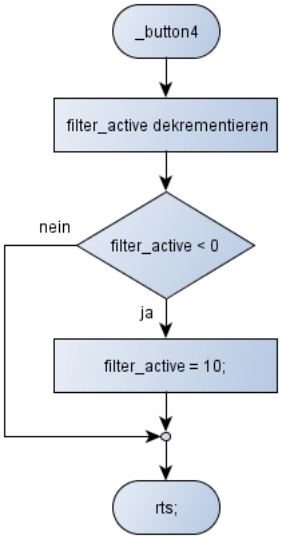


Abbildung 17: Programmablaufplan _button4

2.2.4 _audio

Die Interrupt Service Routine (ISR) *_audio* wird aufgerufen, wenn der Interrupt P6I ausgelöst wird. Dieser wird immer dann ausgelöst, wenn ein neuer Abtastwert des Eingangssignals vorliegt.

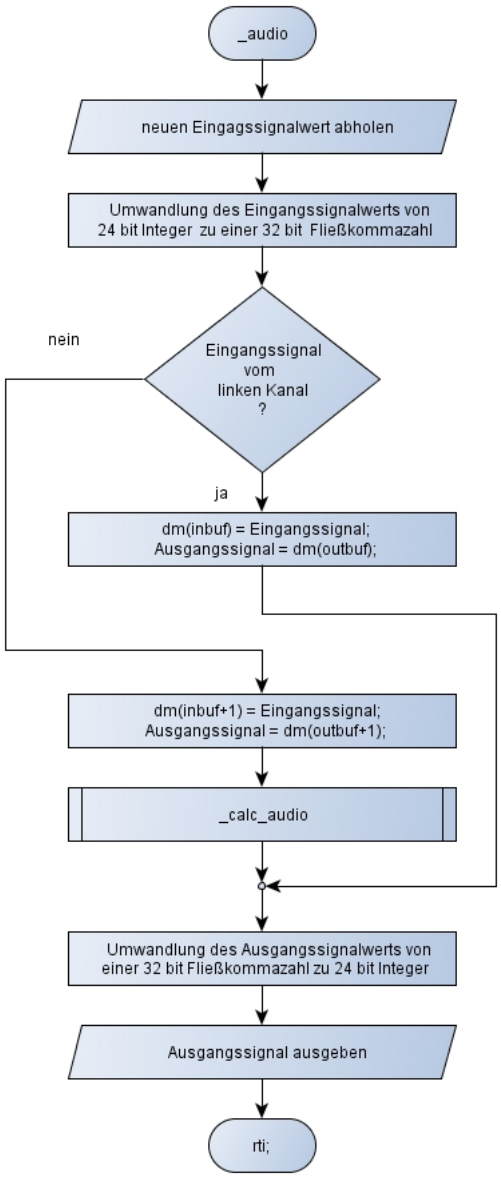


Abbildung 18: Programmablaufplan _audio

Zunächst holt sich die Funktion das Eingangssignal vom Port ab. Danach findet eine Umwandlung des 24 bit Integer-Werts in eine 32 bit Fließkommazahl statt. Dies hat den Hintergrund, dass das Vorzeichen erhalten bleibt und der Prozessor so eingestellt ist, dass er 32-bit-Werte verarbeitet.

Als nächstes wird geprüft, ob es sich bei dem Eingangssignal um einen Wert des rechten oder linken Kanals handelt. Handelt es sich um einen Wert des linken Kanals wird der Wert lediglich in der Variablen *inbuf* zwischengespeichert. Erst wenn es sich um den rechten Kanal handelt, wird der Wert nicht nur zwischengespeichert, sondern auch zur Faltung übergeben. Dies liegt daran, dass bei der Faltung beide Werte parallel berechnet werden.

Als nächstes wird das zuletzt berechnete Ausgangssignal aus der Variable *outbuf* ausgelesen. Das Ausgangssignal wird anschließend von einer 32 bit Fließkommazahl in einen 24 bit Integer-Wert gewandelt und ausgegeben.

2.2.4.1 _calc_audio

Das Unterprogramm *_calc_audio* ist im Grunde die eigentliche Filterung. Hier wird das Eingangssignal, welches mit 512 Werten im Puffer zwischengespeichert ist, mit dem Gesamtfilter gefaltet.

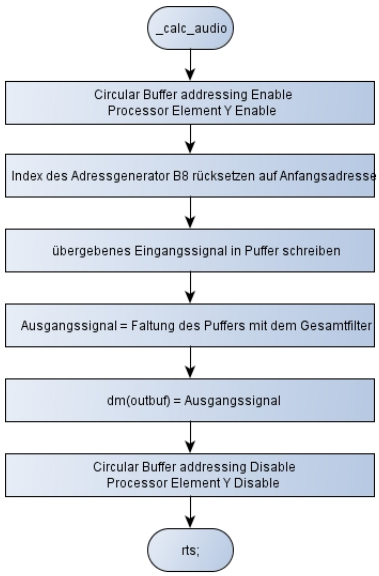


Abbildung 19: Programmablaufplan _calc_audio

Dazu wird zunächst das Circular Buffer Addressing freigegeben und das Processor Element Y (SIMD) zugeschaltet, welches für die parallele Bearbeitung von linken und rechten Kanal notwendig ist. Danach wird der Index I8 des Adressgenerators B8 auf die Anfangsadresse der Variablen *koeff* gesetzt, welche den Gesamtfilter beinhaltet. Als nächstes wird das in der Variablen *inbuf* zwischengespeicherte Eingangssignal in den Puffer, welcher durch die Variable *buffer* repräsentiert wird, geschrieben.

Der nächste Schritt ist die Faltung. Diese wird nach folgender Rechenvorschrift durchgeführt:

$$y(n) = \sum_{k=0}^{N-1} a_k \cdot x(n-k) \tag{25}$$

Da hier die jeweiligen Ergebnisse y(n) nicht zwischengespeichert werden, ist auch nur die Bildung einer Summe notwendig. Durch die Adressierung des Puffers wird immer der passende Signalwert x(n-k) zum jeweiligen Koeffizienten a_k des Gesamtfilters geliefert.

Der so berechnete Ausgangssignalwert wird nun im Ausgangssignalpuffer (outbuf) abgelegt. Zum Schluss wird das Circular Buffer Addressing gesperrt und das Processor Element Y (SIMD) abgeschaltet.

3 Test

Tests wurden im Rahmen der Implementation auditiv durchgeführt. Dabei konnte eine Veränderung der Lautstärke der jeweiligen Frequenzbereiche der Einzelfilter festgestellt werden. Eine quantitative Aussage kann hier allerdings nicht gegeben werden. Dazu wären Messungen, beispielsweise mit einem Bode-Analysator, notwendig.

Die Bedienung konnte jedoch getestet werden und es konnten keine Fehler festgestellt werden. Da die Bedienung nicht sehr umfangreich ist, wurden einfach alle Bedienungsmöglichkeiten durchgespielt.

Die Anzeige ist auch nicht sehr umfangreich und wurde im Zuge der Prüfung der Bedienung mitgeprüft.

4 Probleme

Probleme sind im Bereich der Auflösung (Filterordnung) und der zeitlichen Verzögerung zwischen Eingang des Signals und der Ausgabe des daraus gefilterten Signals festzustellen.

Das Problem der Auflösung liegt darin, dass im unteren Frequenzbereich bereits im Urzustand des Gesamtfilters eine Verstärkung von ca. 3 dB vorhanden ist. Dies ist geschuldet durch die Entscheidung für die Filterordnung von N = 512 und der daraus resultierenden Auflösung. Ein weiterer Aspekt ist die Wahl der Frequenzbereiche für die Einzelfilter. Bei der auditiven Prüfung konnte jedoch kein störender Effekt festgestellt werden. Allerdings wäre hier über eine Überarbeitung der Frequenzbereiche und damit der Einzel- bzw. des Gesamtfilters nachzudenken.

Das andere Problem ruft schon eher einen störenden Effekt hervor. Es handelt sich um die zeitliche Verzögerung zwischen dem Eingang eines Signalwertes und dessen Ausgabe nach der Filterung. Diese Verzögerung ist durch die Ordnung des Filters mit N = 512 geschuldet. Da die Ordnung eine Pufferung von 512 Signalwerten mit sich bringt, wird das Signal bei Eingang auf den aktuellsten Wert gesetzt und muss nun den Puffer durchlaufen und kommt nach 512 Berechnungen als gefiltertes Signal zum Ausgang. Mit einer Abtastfrequenz von f_a = 48 kHz ergibt sich so eine Zeitverzögerung von rund 11 ms. Dies wirkt sich teilweise störend auf die Ausgabe von Videos aus, da es hier zu einer Verschiebung zwischen Audio- und Videosignal kommt. Dieser Verschiebung könnte man durch eine Verzögerung des Videosignals entgegenwirken. Allerdings ist dies nicht geeignet, wenn es sich beispielsweise um ein Konzert handelt, da hier der Zuschauer den Akteur unverzögert wahrnimmt. Hier wäre die Lösung durch einen Equalizer mit IIR-Filter zu sehen, da es hier zu Verzögerungen im Mikrosekundenbereich kommt und diese nicht wahrgenommen werden. Daher ist dieser Equalizer nicht in Zusammenhang mit gleichzeitiger Übertragung von Video- und Audiosignalen geeignet.

5 Ausblick

Im vorangegangenen Abschnitt wurde die Zeitverzögerung als größtes Problem genannt und die Lösung darin gesehen, dass statt FIR-Filter IIR-Filter zur Anwendung kommen sollen. Dazu habe ich den Equalizer auf IIR-Filter umgestellt. Die Filterung erfolgt hier mithilfe kaskadierter IIR-Filter 2.Ordnung, also nach der sogenannten Second Order Structure (SOS).

Warum habe ich diesen Equalizer hier nicht vorgestellt?

Es gibt noch ein paar Probleme zu lösen. Die Filter beeinflussen sich gegenseitig und funktionieren somit nur bedingt. Hier muss eine bessere Abstimmung aufeinander vorgenommen werden. Des Weiteren ergeben sich im Urzustand der Filter unterschiedliche Verstärkungsfaktoren und keine Verstärkung von 1 über den gesamten Frequenzbereich. Auch hier muss noch eine Anpassung vorgenommen werden.

6 Anhang

6.1 Variablen

Variablenbezeichner	Beschreibung
inbuf	Zwischenspeicher Eingangssignal einmal rechter und linker Kanal
outbuf	Zwischenspeicher Ausgangssignal einmal rechter und linker Kanal
buffer	Zwischenspeicher der letzten 513 Eingangssignalwerte (rechter und linker Kanal) für die Faltung
koeff_orig	Koeffizienten der Einzelfilter mit der Verstärkung 1
filter_db	Verstärkungsfaktoren der jeweiligen Einzelfilter in dB
koeff	Koeffizienten des Gesamtfilters (für Faltung mit Eingangssignal)
filter	Verstärkungsfaktor der Einzelfilter
filter_active	momentan ausgewählter Filter
db2filter	Lookup Table zur Wandlung des db-Wertes in den entsprechenden Verstärkungsfaktor
volume_db	Verstärkungsfaktor in dB des Vorverstärkers
volume	Verstärkungsfaktor des Vorverstärkers
led_cache	Übergabeparameter in dem der codierte Wert zum setzen der LEDs übergeben wird (Bit0 = LED1, Bit1 = LED2 usw. ; 0 = aus, 1 = ein)

Tabelle 2: Liste der verwendeten Variablen

6.2 feststehende Adressgeneratoren

Die im folgenden aufgeführten Adressgeneratoren stehen allen Unterprogrammen zur Verfügung und dürfen nicht geändert werden.

Adressgenerator	zugewiesene Variable
B0	koeff_orig
B5	buffer
B8	koeff
B9	filter

Tabelle 3: Liste der feststehenden Adressgeneratoren

6.3 Diagramme der Einzelfilter

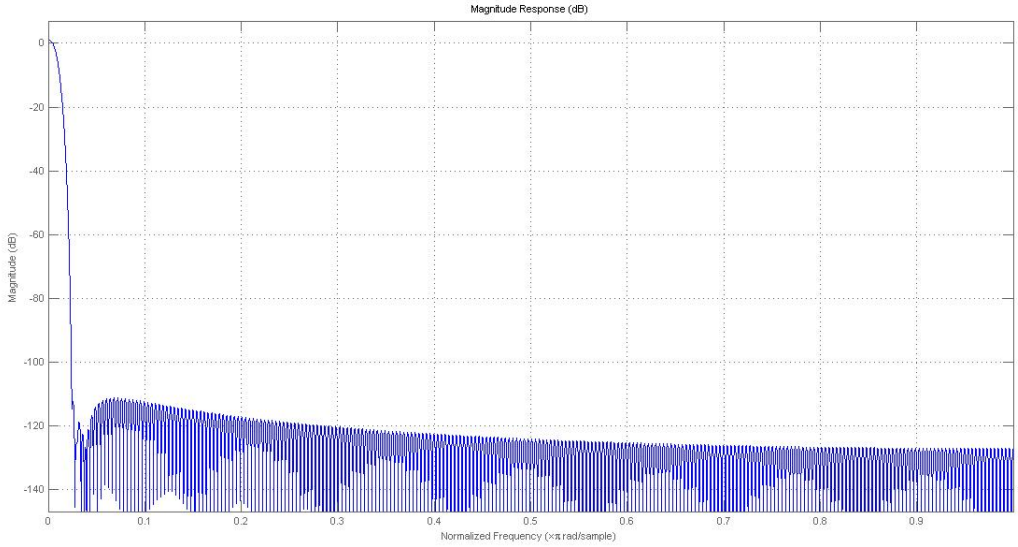


Abbildung 20: Amplitudengang Filter 1

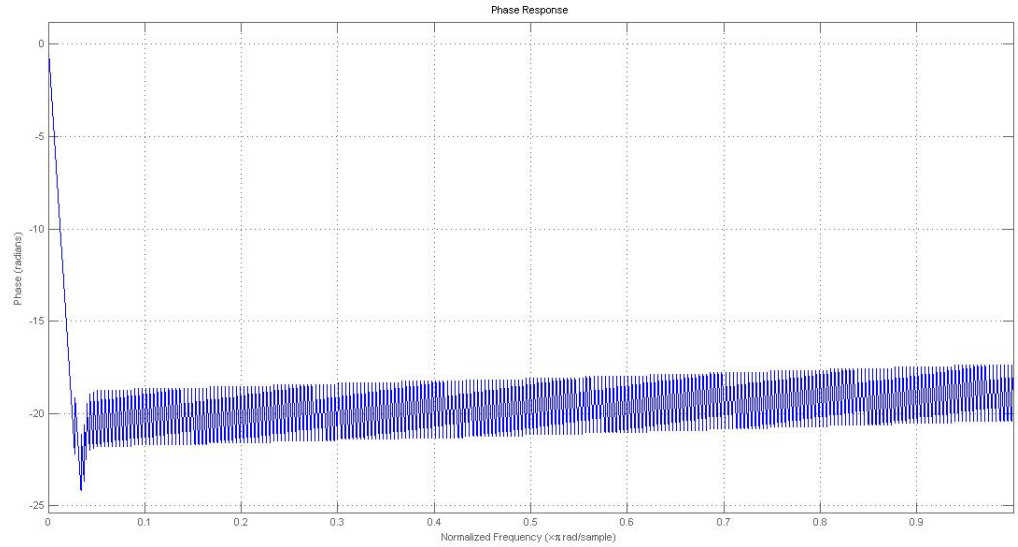


Abbildung 21: Phasengang Filter 1

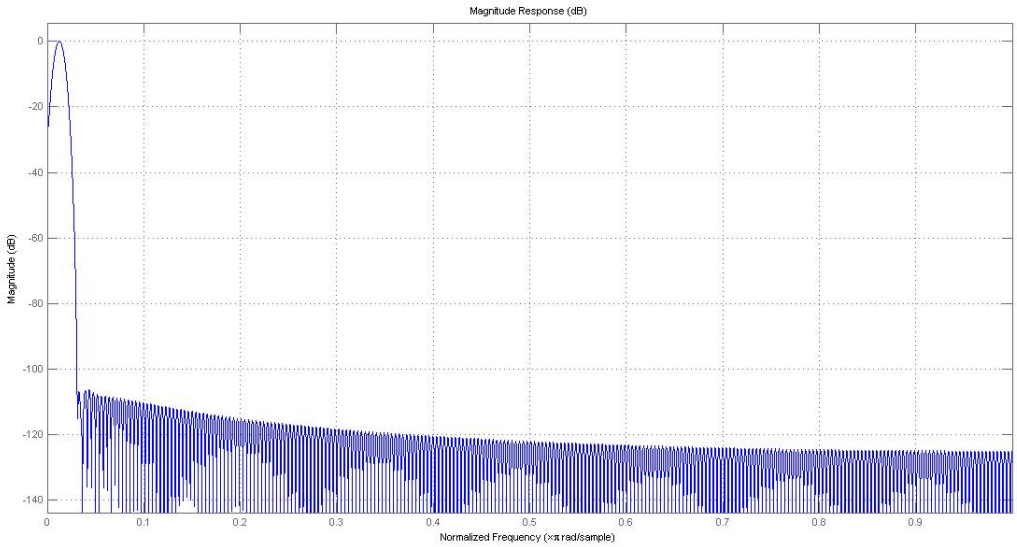


Abbildung 22: Amplitudengang Filter 2

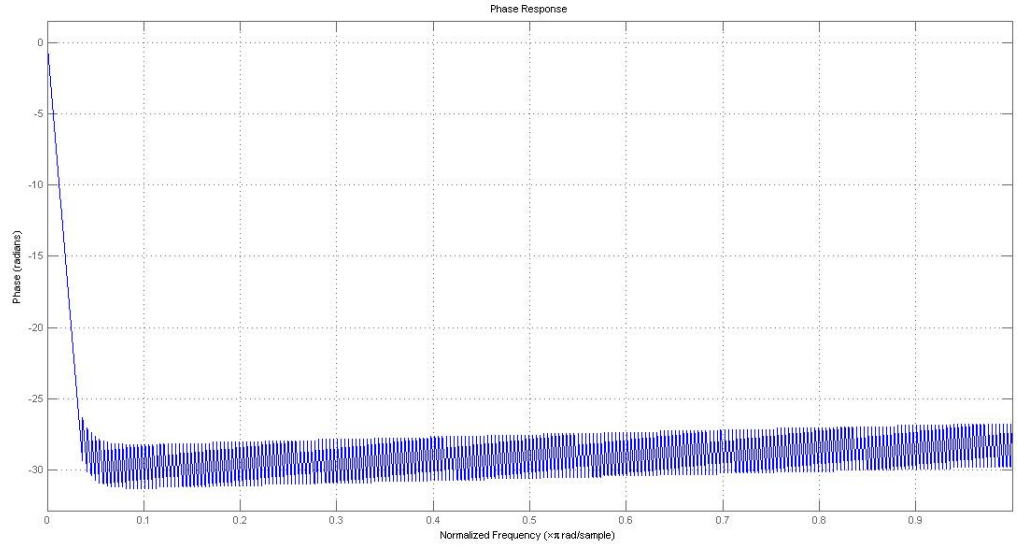


Abbildung 23: Phasengang Filter 2

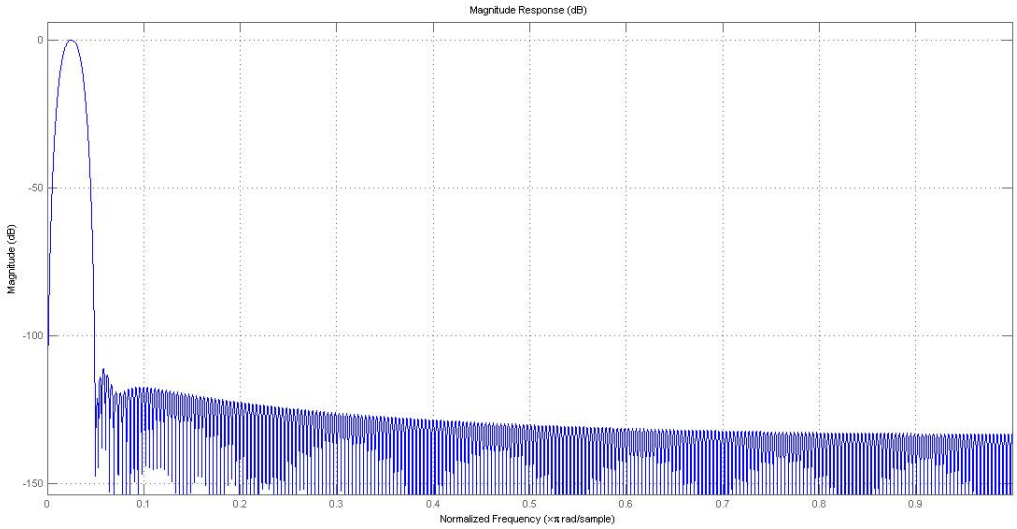


Abbildung 24: Amplitudengang Filter 3

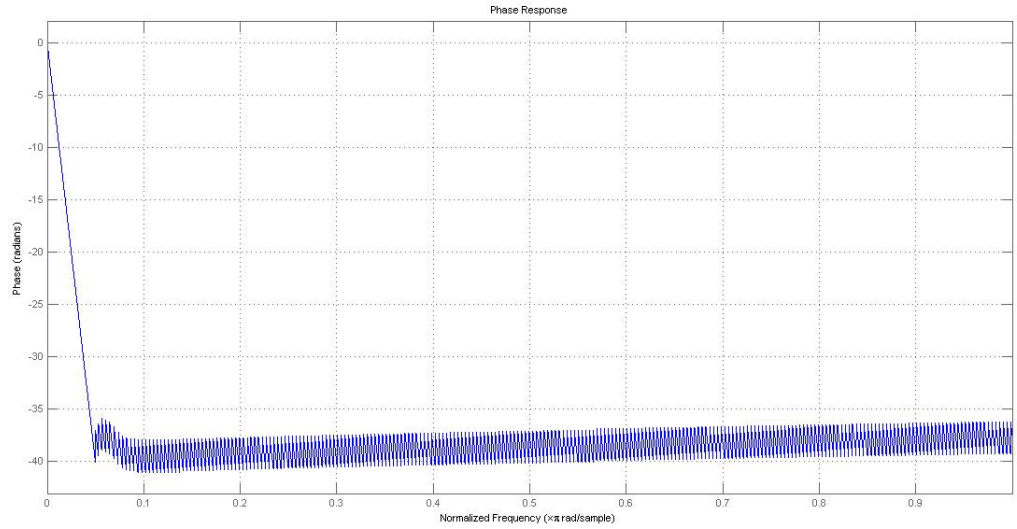


Abbildung 25: Phasengang Filter 3

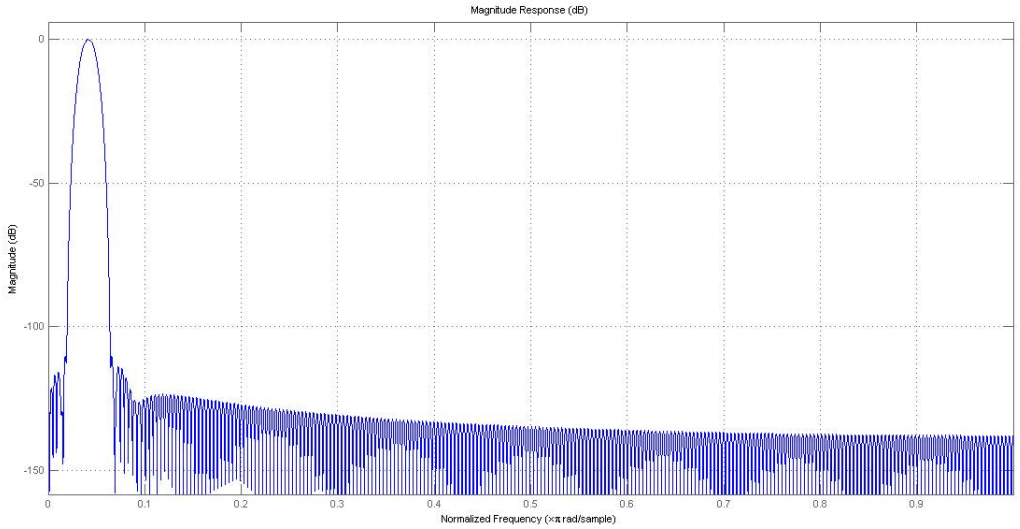


Abbildung 26: Amplitudengang Filter 4

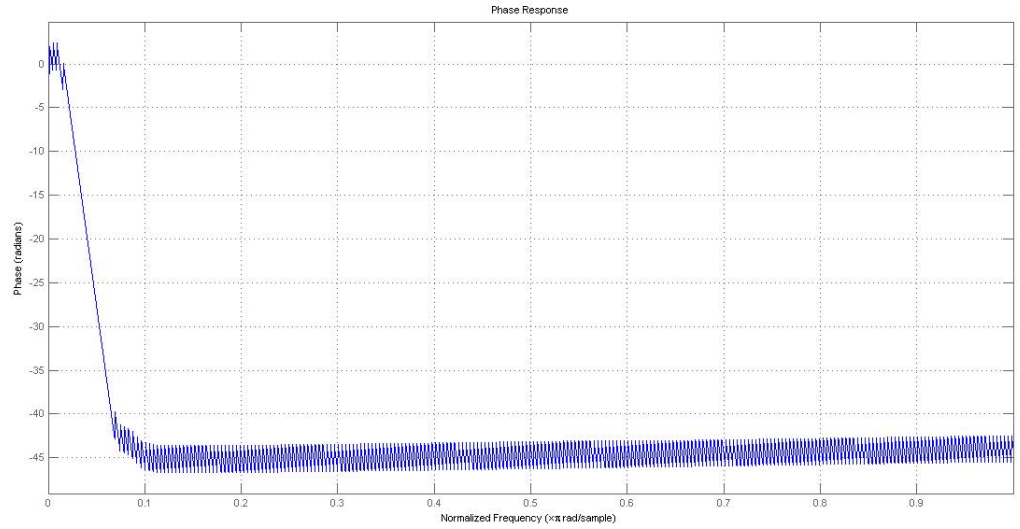


Abbildung 27: Phasengang Filter 4

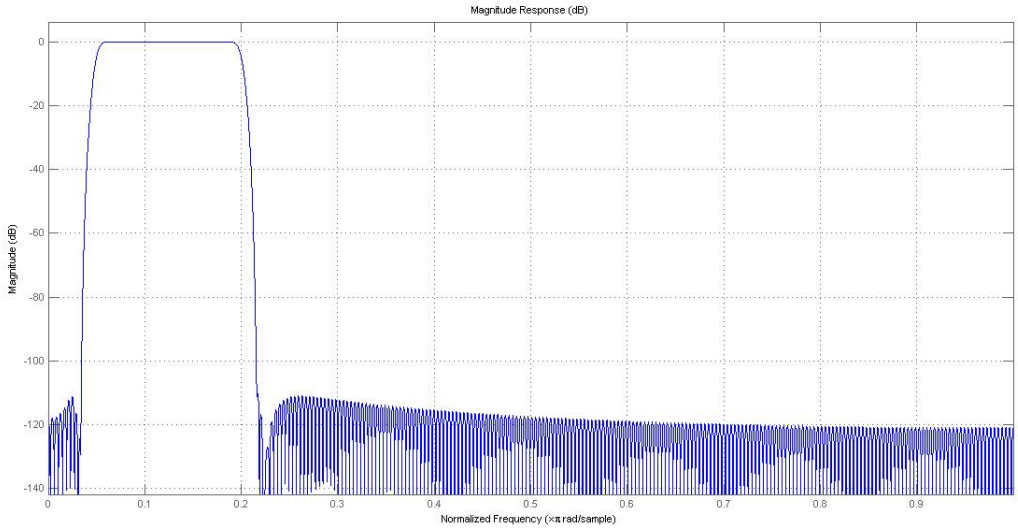


Abbildung 28: Amplitudengang Filter 5

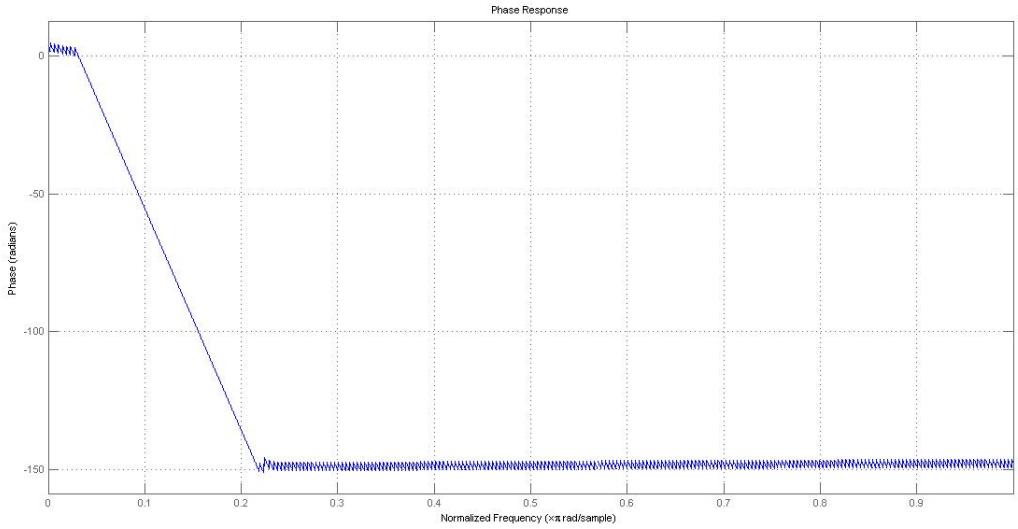


Abbildung 29: Phasengang Filter 5

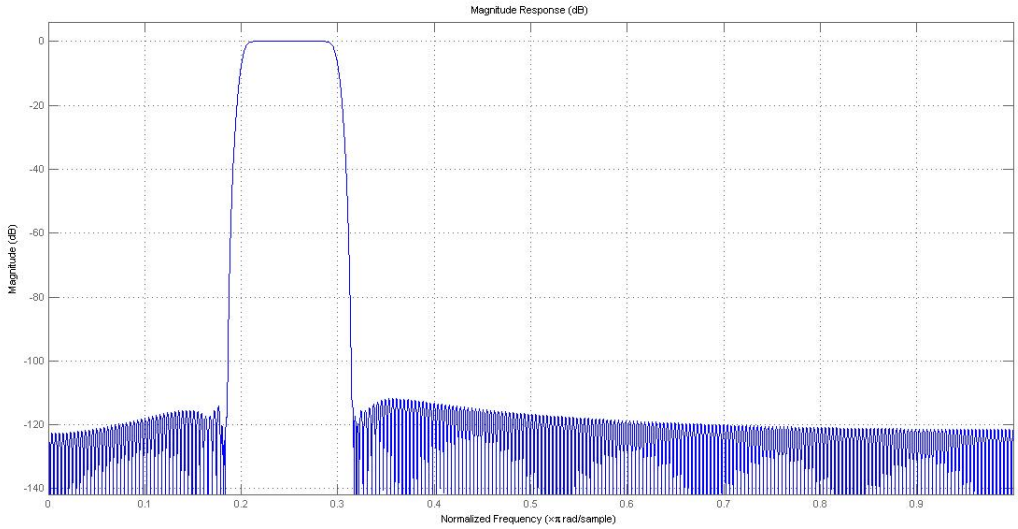


Abbildung 30: Amplitudengang Filter 6

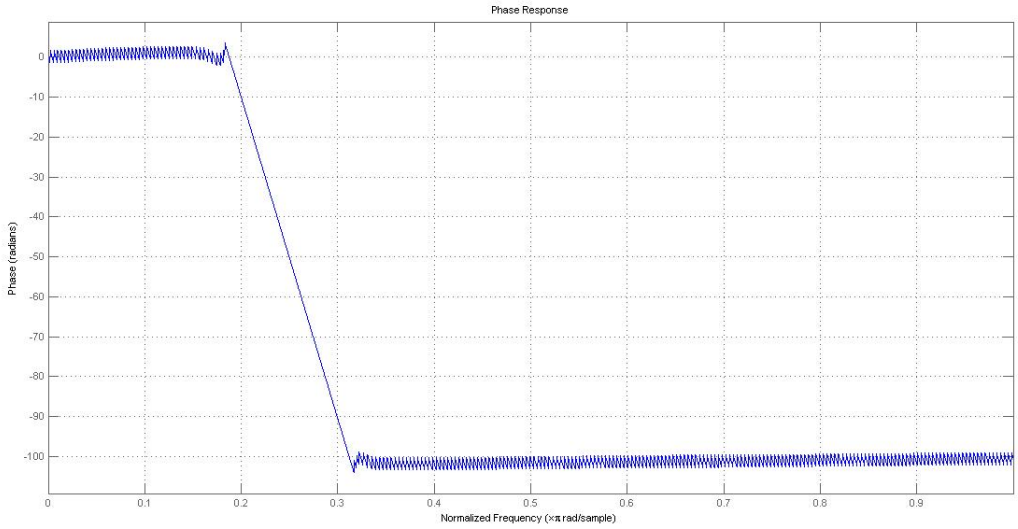


Abbildung 31: Phasengang Filter 6

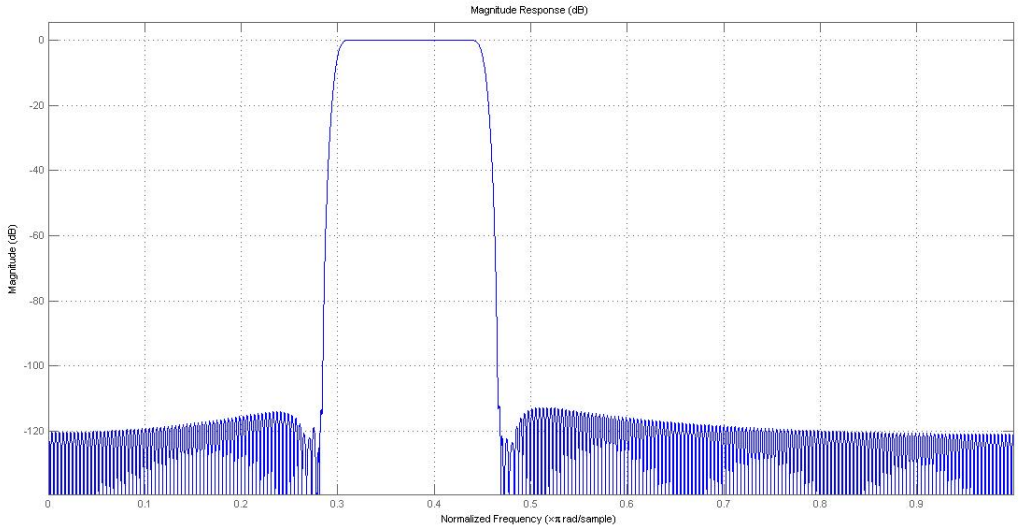


Abbildung 32: Amplitudengang Filter 7

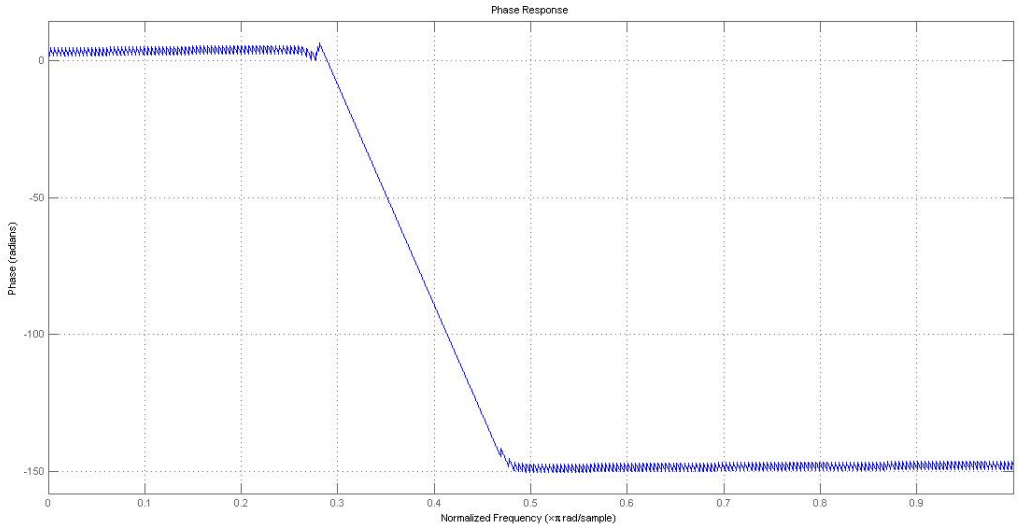


Abbildung 33: Phasengang Filter 7

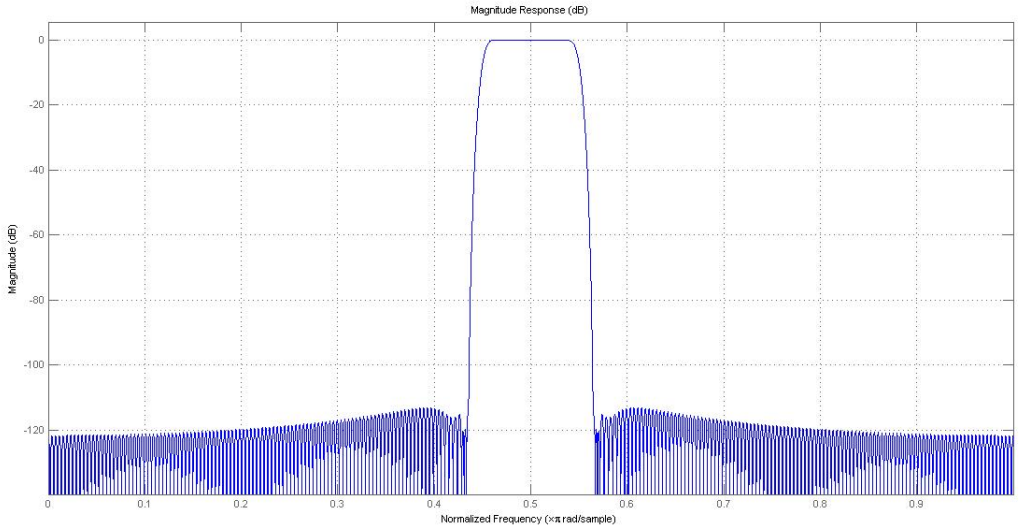


Abbildung 34: Amplitudengang Filter 8

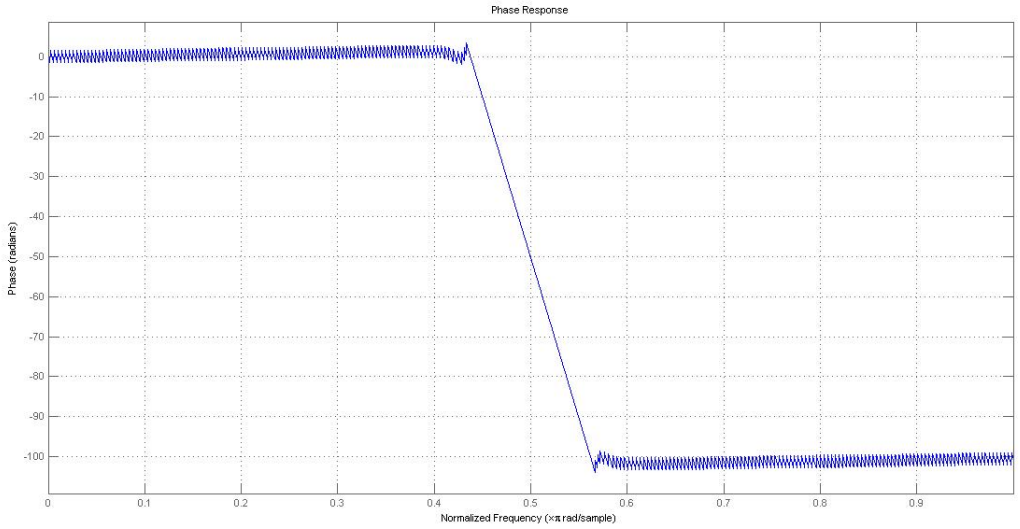


Abbildung 35: Phasengang Filter 8

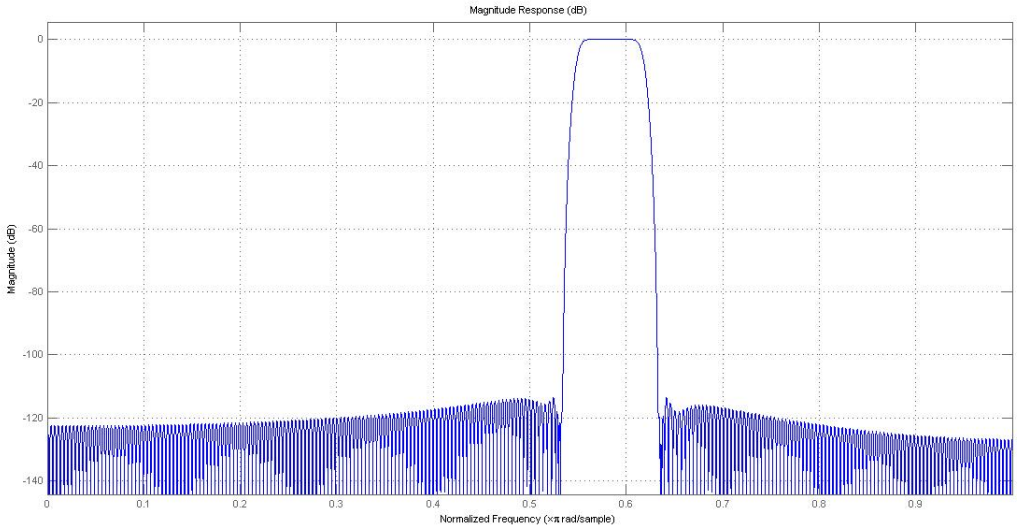


Abbildung 36: Amplitudengang Filter 9

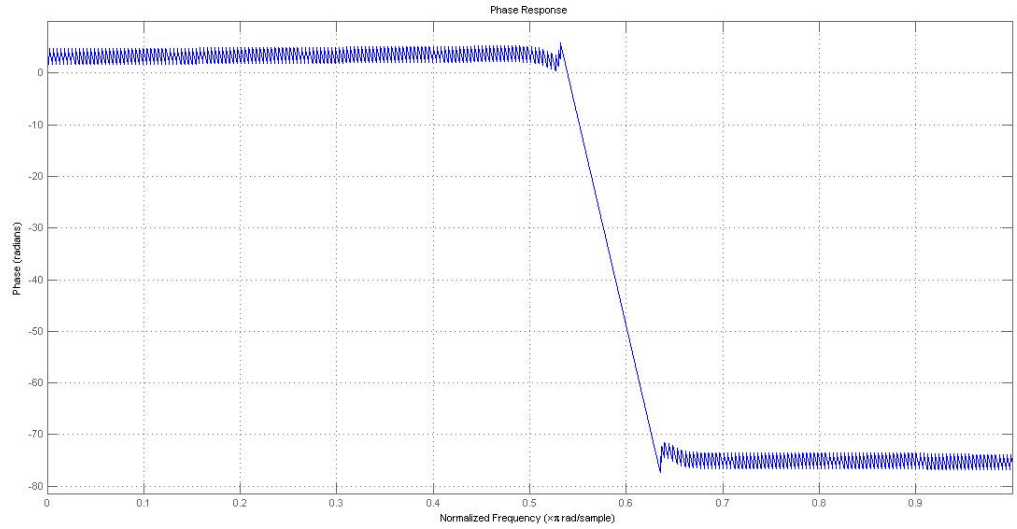


Abbildung 37: Phasengang Filter 9

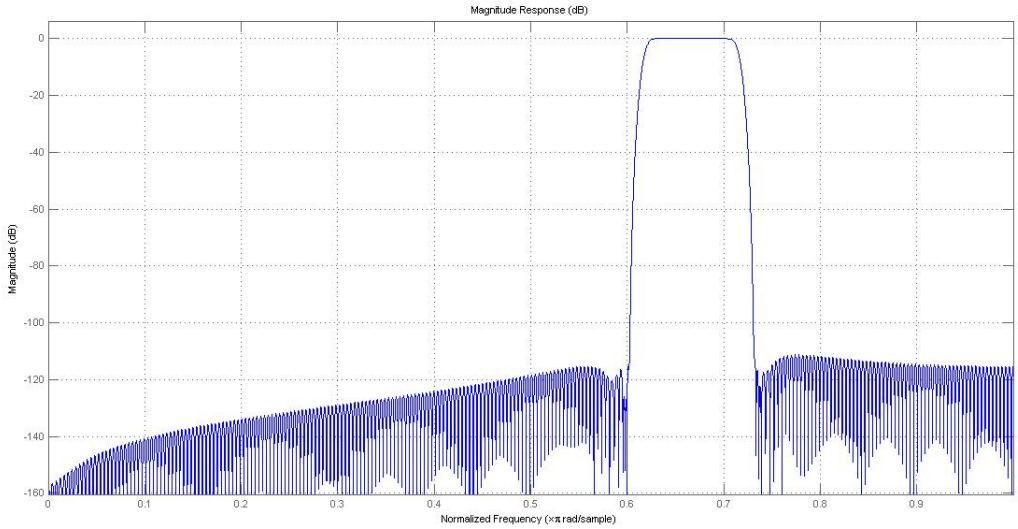


Abbildung 38: Amplitudengang Filter 10

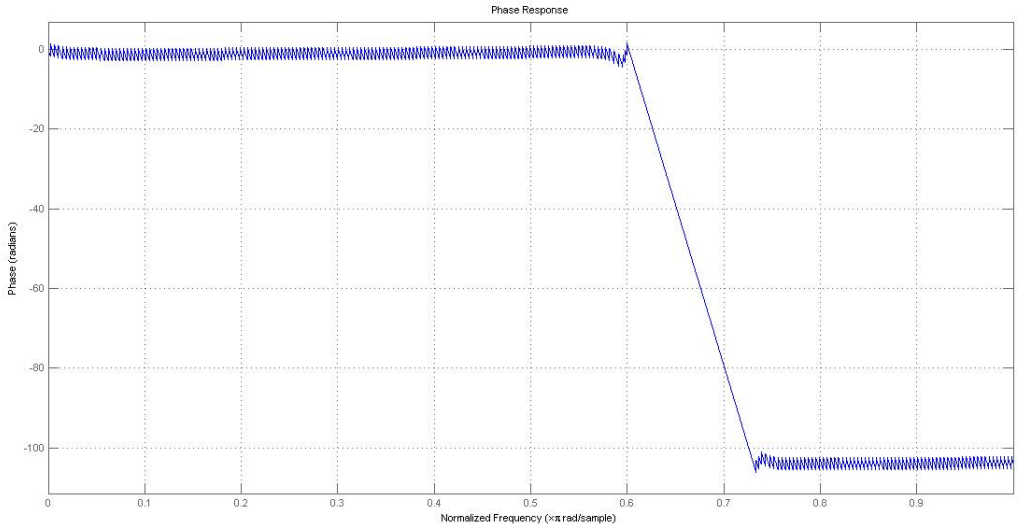


Abbildung 39: Phasengang Filter 10

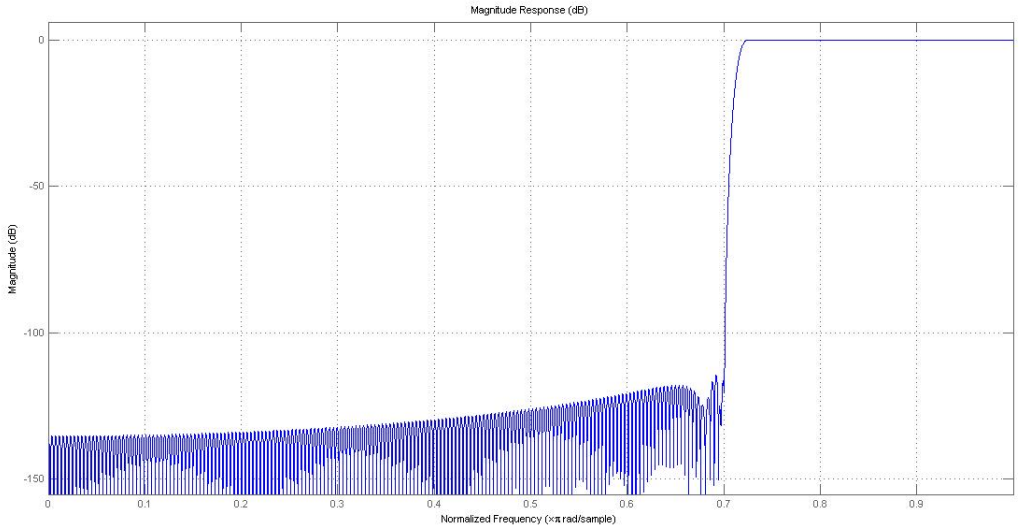


Abbildung 40: Amplitudengang Filter 11

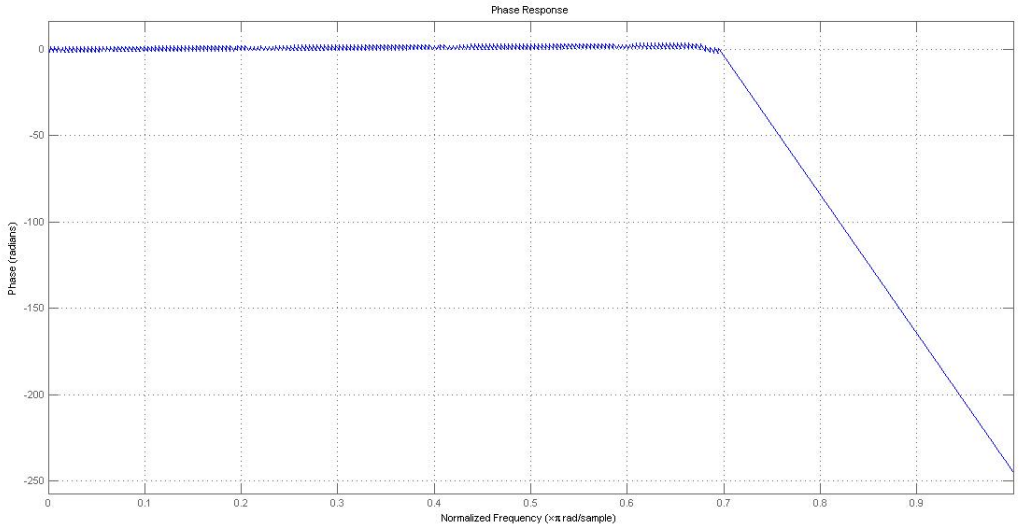


Abbildung 41: Phasengang Filter 11

6.4 Diagramme des Gesamtfilters

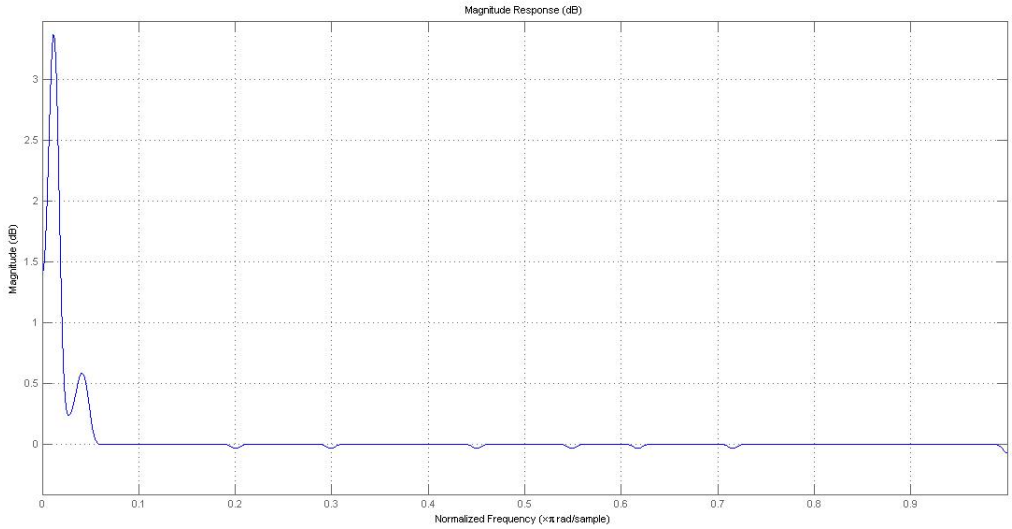


Abbildung 42: Amplitudengang Gesamtfilter

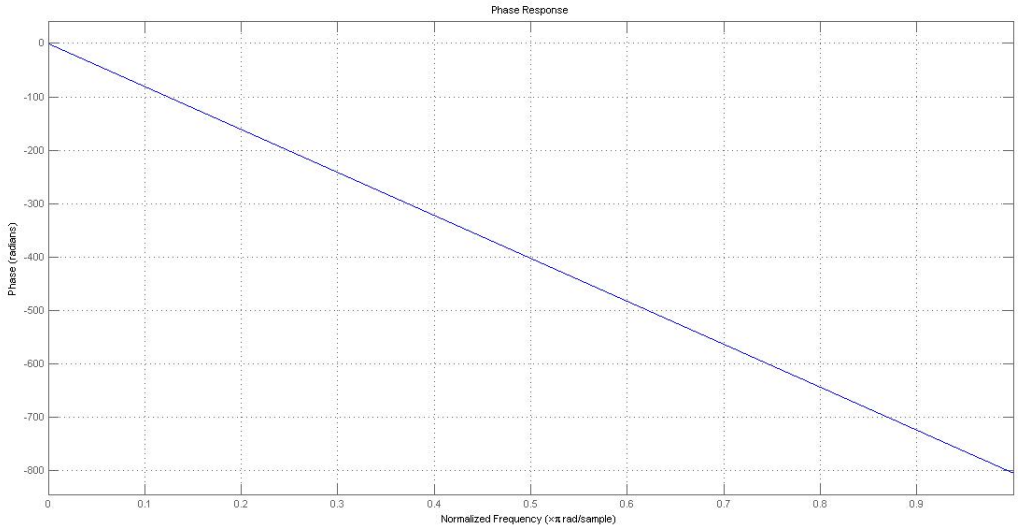


Abbildung 43: Phasengang Gesamtfilter

6.5 MatLab-File & filter.dat

```
function calcEQFilter(speicherort)

% Filterparameter initialisieren
N = 512; fa = 48000; win = nuttallwin(N+1); flag = 'scale';

% Einzelfilter berechnen
fu = 1;     fo = 219;   f1 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 220;   fo = 380;   f2 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 381;   fo = 819;   f3 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 820;   fo = 1180;  f4 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 1181;  fo = 4819;  f5 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 4820;  fo = 7180;  f6 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 7181;  fo = 10819; f7 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 10820; fo = 13180; f8 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 13181; fo = 14819; f9 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 14820; fo = 17180; f10 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);
fu = 17181; fo = 23999; f11 = fir1(N,[fu/(fa/2), fo/(fa/2)], 'bandpass', win, flag);

% Koeffizienten zusammenstellen fuer filter.dat
l = 1;
for k = 1:N+1
    fx(1,l)   = f1(1,k);  fx(1,l+1) = f1(1,k);  l = l + 2;
    fx(1,l)   = f2(1,k);  fx(1,l+1) = f2(1,k);  l = l + 2;
    fx(1,l)   = f3(1,k);  fx(1,l+1) = f3(1,k);  l = l + 2;
    fx(1,l)   = f4(1,k);  fx(1,l+1) = f4(1,k);  l = l + 2;
    fx(1,l)   = f5(1,k);  fx(1,l+1) = f5(1,k);  l = l + 2;
    fx(1,l)   = f6(1,k);  fx(1,l+1) = f6(1,k);  l = l + 2;
    fx(1,l)   = f7(1,k);  fx(1,l+1) = f7(1,k);  l = l + 2;
    fx(1,l)   = f8(1,k);  fx(1,l+1) = f8(1,k);  l = l + 2;
    fx(1,l)   = f9(1,k);  fx(1,l+1) = f9(1,k);  l = l + 2;
    fx(1,l)   = f10(1,k); fx(1,l+1) = f10(1,k); l = l + 2;
    fx(1,l)   = f11(1,k); fx(1,l+1) = f11(1,k); l = l + 2;
end

% filter.dat speichern
dlmwrite(speicherort,fx,'delimiter','\n','precision','%.15f')
```

6.6 db2filter.dat

0.100000000000000
0.112201845430196
0.125892541179417
0.141253754462275
0.158489319246111
0.177827941003892
0.199526231496888
0.223872113856834
0.251188643150958
0.281838293126445
0.316227766016838
0.354813389233575
0.398107170553497
0.446683592150963
0.501187233627272
0.562341325190349
0.630957344480193
0.707945784384138
0.794328234724281
0.891250938133746
1.000000000000000
1.122018454301963
1.258925411794167
1.412537544622754
1.584893192461114
1.778279410038923
1.995262314968880
2.238721138568339
2.511886431509580
2.818382931264454
3.162277660168380
3.548133892335755
3.981071705534972
4.466835921509632
5.011872336272722
5.623413251903491
6.309573444801933
7.079457843841379
7.943282347242816
8.912509381337454
10.000000000000000

6.7 Quellcode

Auf den folgenden Seiten ist der vollständige Quellcode des Equalizers aufgeführt. Dies beinhaltet folgende Dateien:

- main.asm
- initPLL_SDRAM.asm
- sru.asm
- initSPORT.asm
- init1835viaSPI.asm
- interrupts.asm
- filter.asm
- audio.asm
- buttons.asm
- leds.asm

6.7.1 main.asm

```
////////////////////////////////
/*******************************/
/*                             */
/*  Fachhochschule Jena        */
/*  FB ET/IT                   */
/*  Jürgen Döffinger (631551)  */
/*                             */
/*  Projekt:    Equalizer      */
/*  Datei:      main.asm       */
/*  Version:    2012-01-11-01  */
/*                             */
/*******************************/
////////////////////////////////


// Headerdateien einbinden
#include <def21369.h>


// Code - Section
.section /pm seg_pmco;


// globale Funktionen deklarieren
.global _main;


// externe Funktionen und Variablen einbinden
.extern _initPLL;
.extern _initSDRAM;
.extern _init_SRU;
.extern _initSPORT;
.extern _init1835viaSPI;
.extern _init_filter;
.extern _init_audio;


/*********************   START MAIN   ********************/

_main:
```

```
    // Initializes PLL for the correct core clock (CCLK) frequency
    call _initPLL;

    // Initializes SDRAM for the correct SDRAM clock (SDCLK) frequency
    call _initSDRAM;

    // Initializes the SRU & DAI/DPI pins
    call _init_SRU;

    // Initializes the transmit and receive serial ports (SPORTS)
    call _initSPORT;

    // Initializes the codec
    call _init1835viaSPI;

    // Filter initialisieren und mit Startwerten füllen und Koeffizienten für die Faltung berechnen
    call _init_filter;

    // Audiobearbeitung initialisieren
    call _init_audio;

    // Enable interrupts (globally)
    BIT SET MODE1 IRPTEN;

    // Unmask the SPORT0 ISR
    LIRPTL = SP0IMSK;

_main.end:

/*********************  END MAIN  ********************/


// Loop forever.  Work is driven by interrupts

jump (pc,0);
```

6.7.2 initPLL_SDRAM.asm

```
/* Sets up the SDRAM controller to access SDRAM. In this file are two subroutines, the first
   to set up the SHARC's PLL, and the second to set up the SDRAM controller.
   CLKIN= 24.576 MHz, Multiplier= 27, Divisor= 2, CCLK_SDCLK_RATIO 2.0.
   Core clock = (24.576MHz * 27) /2 = 331.776 MHz
*/

#include <def21369.h>

.global _initPLL;
.global _initSDRAM;

.section/pm seg_pmco;

_initPLL:

// CLKIN= 24.576 MHz, Multiplier= 27, Divisor= 2, CCLK_SDCLK_RATIO 2.
// Core clock = (24.576MHz * 27) /2 = 331.776 MHz
    ustat3 = PLLM27|PLLD2|DIVEN;

changePLL:

    // Set the Core clock (CCLK) to SDRAM clock (SDCLK) ratio to 2
   bit set ustat3 SDCKR2;

    dm(PMCTL) = ustat3;
    bit set ustat3 PLLBP;

    dm(PMCTL) = ustat3;

    // Wait for at least 4096 cycles for the pll to lock
    lcntr = 5000, do loopend2 until lce;
loopend2:    nop;

    ustat3 = dm(PMCTL);
    bit clr ustat3 PLLBP;
    dm(PMCTL) = ustat3;

_initPLL.end: rts;


_initSDRAM:

    // Programming SDRAM control registers.
    // RDIV = ((f SDCLK X t REF ) /NRA) - (tRAS + tRP )

    // CCLK_SDCLK_RATIO==2
```

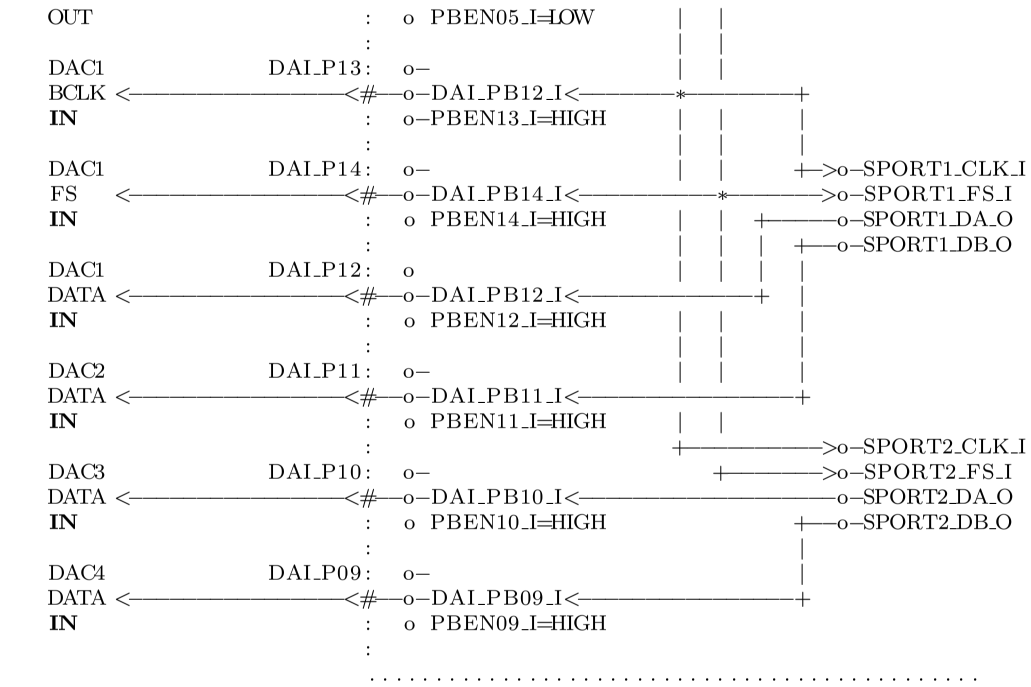
```
    ustat4 = 0xA17; //(166*(10^6)*64*(10^-3)/4096) - (7+3) = 2583


    //=============================================================================
    //
    //Configure SDRAM Control Register (SDCTL) for the Micron MT48LC4M32
    //
    //  SDCL3  : SDRAM CAS Latency= 3 cycles
    //  DSDCLK1: Disable SDRAM Clock 1
    //  SDPSS  : Start SDRAM Power up Sequence
    //  SDCAW8 : SDRAM Bank Column Address Width= 8 bits
    //  SDRAW12: SDRAM Row Address Width= 12 bits
    //  SDTRAS7: SDRAM tRAS Specification . Active Command delay = 7 cycles
    //  SDTRP3 : SDRAM tRP Specification . Precharge delay = 3 cycles .
    //  SDTWR2 : SDRAM tWR Specification . tWR = 2 cycles .
    //  SDTRCD3: SDRAM tRCD Specification . tRCD = 3 cycles .
    //
    //-----------------------------------------------------------------------------

    ustat3 = SDCL3|DSDCLK1|SDPSS|SDCAW8|SDRAW12|SDTRAS7|SDTRP3|SDTWR2|SDTRCD3;

    dm(SDCTL) = ustat3;

    //Change this value to optimize the performance for quazi-sequential accesses (step >1)
    #define SDMODIFY 1                    //Setting the Modify to 1
    bit set ustat4 (SDMODIFY<<17)|SDROPT; //Enabling SDRAM read optimization

    dm(SDRRC) = ustat4;

    //Note that MS2 & MS3 pin multiplexed with flag2 & flag3 .
    //MSEN bit must be enabled to access SDRAM, but LED8 cannot be driven with sdram
    ustat3=dm(SYSCTL);
    bit set ustat3 MSEN;    //This setting allows SDRAM access
    //bit clr ustat3 MSEN;  //This setting allows Flag3 to be used
    dm(SYSCTL)=ustat3;

    //Mapping Bank 2 to SDRAM
    //Make sure that jumper is set appropriately so that MS2 is connected to
    //chip select of 16-bit SDRAM device
    ustat3 = dm(EPCTL);
    bit set ustat3 B2SD;
    bit clr ustat3 B0SD|B1SD|B3SD;
    dm(EPCTL) = ustat3;

    //=============================================================================
    //
    //Configure AMI Control Register (AMICTL0) Bank 0 for the ISSI IS61LV5128
    //
    //  WS2 : Wait States= 2 cycles
```

```
    //  HC1   : Bus Hold Cycle (at end of write access) = 1 cycle.
    //  AMIEN : Enable AMI
    //  BW8   : External Data Bus Width = 8 bits.
    //
    //-------------------------------------------------------------------------

    //SRAM Settings
    ustat4 = WS2|HC1|AMIEN|BW8;
    dm(AMICTL0) = ustat4;

    //=========================================================================
    //
    // Configure AMI Control Register (AMICTL) Bank 1 for the AMD AM29LV08
    //
    //  WS23  : Wait States = 23 cycles
    //  AMIEN : Enable AMI
    //  BW8   : External Data Bus Width = 8 bits.
    //
    //-------------------------------------------------------------------------

    //Flash Settings
    ustat4 = WS23|AMIEN|BW8;
    dm(AMICTL1) = ustat4;


_initSDRAM.end: rts;
```

6.7.3 sru.asm

```
////////////////////////////////
/*******************************/
/*                             */
/*  Fachhochschule Jena        */
/*  FB ET/IT                   */
/*  Jürgen Döffinger (631551)  */
/*                             */
/*  Projekt:    Equalizer      */
/*  Datei:      sru.asm        */
/*  Version:    2012-01-11-01  */
/*                             */
/*******************************/
////////////////////////////////


// Headerdateien einbinden
#include <def21369.h>
#include <SRU.h>


// globale Funktionen deklarieren
.global _init_SRU;


// externe Funktionen einbinden
.extern _init_Buttons;
.extern _init_LEDs;


// Code - Section
.section /pm seg_pmco;


/********************   START _init_SRU    ********************/

_init_SRU:

  // LEDs initialisieren
  call _init_LEDs;

  // Buttons initialisieren
```

```
  call _init_Buttons;

  // SRU für Audio initialisieren
  call _init_SRU_Audio;

_init_SRU.end: rts;

/*******************   END _init_SRU    ******************/


/*******************   START init_SRU_Audio    ******************/

_init_SRU_Audio:


// The following definition allows the SRU macro to check for errors. Once the routings have
// been verified, this definition can be removed to save some program memory space.
// The preprocessor will issue a warning stating this when using the SRU macro without this
// definition
#define SRUDEBUG  // Check SRU Routings for errors.


/*-------------------------------------------------------------------------------

                *** EZ-KIT ANALOG-IN ROUTING OVERVIEW ***

   XTAL
   OSC  >---+              ..........................................
   12.288   |             :SHARC
            |
   ADC      |      DAI_P06:  o-DAI_PB06_O------ (MCLK routed to the ADC via SRU)
   MCLK <---+------------>#--o-
   IN                    :  o PBEN06_I=LOW
                         :
   ADC             DAI_P17:  o-DAI_PB17_O------ (MCLK routed to the DACs via SRU)
   MCLK >---------------->#--o-
   IN                    :  o PBEN17_I=LOW
                         :
   ADC             DAI_P07:  o-DAI_PB07_O--------*----------->o-SPORT0_CLK_I
   BCLK >---------------->#--o-                  |  +-------->o-SPORT0_FS_I
   OUT                   :  o-PBEN07_I=LOW       |  |      +->o-SPORT0_DA_I
                         :                       |  |      |
   ADC             DAI_P08:  o-DAI_PB08_O--------*      |
   FS   >---------------->#--o-                  |  |      |
   OUT                   :  o-PBEN04_I=LOW       |  |      |
                         :                       |  |      |
   ADC             DAI_P05:  o-DAI_PB05_O-------------------+
   DATA >---------------->#--o-                  |  |
```



*/

```
  // Disable the pull-up resistors on all 20 pins
  r0 = 0x000FFFFF;
  dm(DAI_PIN_PULLUP) = r0;



//-------------------------------------------------------------------------
//
//  MCLK: The output of the 12.288 MHz xtal is either directly connected to the
//          codec, but also connected to DAI_P06, or just to DAI_P17. This is
//             determined by switch SW3. For this example we route the
//             MCLK into DAI_P17 and supply the clock to the ADC via DAI_P06
//             by routing the signal through the SRU.

//  Tie the pin buffer input LOW.
    SRU(LOW, DAI_PB17_I);
```

```
//  Tie the pin buffer enable input LOW
    SRU(LOW,PBEN17_I);

//-------------------------------------------------------------------------------
//
//  Connect the ADC: The codec drives a BCLK output to DAI pin 7, a LRCLK
//          (a.k.a. frame sync) to DAI pin 8 and data to DAI pin 5.
//
//          Connect the ADC to SPORT0, using data input A
//
//          All three lines are always inputs to the SHARC so tie the pin
//          buffer inputs and pin buffer enable inputs all low.


//-------------------------------------------------------------------------------
//  Connect the ADC to SPORT0, using data input A

    //  Clock in on pin 7
    SRU(DAI_PB07_O,SPORT0_CLK_I);

    //  Frame sync in on pin 8
    SRU(DAI_PB08_O,SPORT0_FS_I);

    //  Data in on pin 5
    SRU(DAI_PB05_O,SPORT0_DA_I);

//-------------------------------------------------------------------------------
//  Tie the pin buffer inputs LOW for DAI pins 5, 7 and 8.  Even though
//    these pins are inputs to the SHARC, tying unused pin buffer inputs
//    LOW is "good coding style" to eliminate the possibility of
//    termination artifacts internal to the IC.  Note that signal
//    integrity is degraded only with a few specific SRU combinations.
//    In practice, this occurs VERY rarely, and these connections are
//    typically unnecessary.


    SRU(LOW,DAI_PB05_I);
    SRU(LOW,DAI_PB07_I);
    SRU(LOW,DAI_PB08_I);

//-------------------------------------------------------------------------------
//  Tie the pin buffer enable inputs LOW for DAI pins 5, 7 and 8 so
//    that they are always input pins.

    SRU(LOW,PBEN05_I);
    SRU(LOW,PBEN07_I);
    SRU(LOW,PBEN08_I);

//-------------------------------------------------------------------------------
```

```
//
//  Connect the DACs: The codec accepts a BCLK input from DAI pin 13 and
//          a LRCLK (a.k.a. frame sync) from DAI pin 14 and has four
//          serial data outputs to DAI pins 12, 11, 10 and 9
//
//          Connect DAC1 to SPORT1, using data output A
//          Connect DAC2 to SPORT1, using data output B
//          Connect DAC3 to SPORT2, using data output A
//          Connect DAC4 to SPORT2, using data output B
//
//          Connect the clock and frame sync inputs to SPORT1 and SPORT2
//          should come from the ADC on DAI pins 7 and 8, respectively
//
//          Connect the ADC BCLK and LRCLK back out to the DAC on DAI
//          pins 13 and 14, respectively.
//
//          All six DAC connections are always outputs from the SHARC
//          so tie the pin buffer enable inputs all high.
//

//-------------------------------------------------------------------------
//  Connect the pin buffers to the SPORT data lines and ADC BCLK & LRCLK

    SRU(SPORT2_DB_O, DAI_PB09_I);
    SRU(SPORT2_DA_O, DAI_PB10_I);
    SRU(SPORT1_DB_O, DAI_PB11_I);
    SRU(SPORT1_DA_O, DAI_PB12_I);

//-------------------------------------------------------------------------
//  Connect the clock and frame sync input from the ADC directly
//    to the output pins driving the DACs.

    SRU(DAI_PB07_O, DAI_PB13_I);
    SRU(DAI_PB08_O, DAI_PB14_I);
    SRU(DAI_PB17_O, DAI_PB06_I);

//-------------------------------------------------------------------------
//  Connect the SPORT clocks and frame syncs to the clock and
//    frame sync from the SPDIF receiver

    SRU(DAI_PB07_O, SPORT1_CLK_I);
    SRU(DAI_PB07_O, SPORT2_CLK_I);
    SRU(DAI_PB08_O, SPORT1_FS_I);
    SRU(DAI_PB08_O, SPORT2_FS_I);

//-------------------------------------------------------------------------
//  Tie the pin buffer enable inputs HIGH to make DAI pins 9-14 outputs.
    SRU(HIGH, PBEN06_I);
    SRU(HIGH, PBEN09_I);
```

```
    SRU(HIGH,PBEN10_I);
    SRU(HIGH,PBEN11_I);
    SRU(HIGH,PBEN12_I);
    SRU(HIGH,PBEN13_I);
    SRU(HIGH,PBEN14_I);
//-------------------------------------------------------------------------
// Route SPI signals to AD1835.

    SRU(SPI_MOSI_O,DPI_PB01_I)      //Connect MOSI to DPI PB1.
    SRU(DPI_PB02_O, SPI_MISO_I)     //Connect DPI PB2 to MISO.
    SRU(SPI_CLK_O, DPI_PB03_I)      //Connect SPI CLK to DPI PB3.
    SRU(SPI_FLG3_O, DPI_PB04_I)     //Connect SPI FLAG3 to DPI PB4.
//-------------------------------------------------------------------------
// Tie pin buffer enable from SPI peipherals to determine whether they are
// inputs or outputs

    SRU(SPI_MOSI_PBEN_O, DPI_PBEN01_I);
    SRU(SPI_MISO_PBEN_O, DPI_PBEN02_I);
    SRU(SPI_CLK_PBEN_O, DPI_PBEN03_I);
    SRU(SPI_FLG3_PBEN_O, DPI_PBEN04_I);


//-------------------------------------------------------------------------


_init_SRU_Audio.end: rts;

/*********************   START init_SRU_Audio    *********************/
```

6.7.4 initSPORT.asm

```
////////////////////////////////////////////////////////////////////////////////////
//NAME:       initSPORT.asm
//DATE:       7/29/05
//USAGE:      This file initializes the transmit and receive serial ports (SPORTS). It uses
//                          uses SPORT0 to receive data from the ADC and transmits the data to the DAC's
//                  via SPORT1A, SPORT1B, SPORT2A and SPORT2B.
//
////////////////////////////////////////////////////////////////////////////////////

#include <def21369.h>

.global _initSPORT;
.section/pm seg_pmco;

_initSPORT:

        //=============================================================
        //
        // Make sure that the multichannel mode registers are cleared
        //
        //=============================================================

        r0 = 0;
        dm(SPMCTL0) = r0;
        dm(SPMCTL1) = r0;
        dm(SPMCTL2) = r0;
        dm(SPCTL0) = r0;
        dm(SPCTL1) = r0;
        dm(SPCTL2) = r0;

        //=============================================================
        //
        // Configure SPORT 0 as a receiver (input from ADC)
        //
        //    OPMODE = I2S mode
        //    SLEN24 = 24 bit of data in each 32-bit word
        //    SPEN_A = Enable data channel A
        //
        //-------------------------------------------------------------

        r0 = OPMODE | SLEN24 | SPEN_A;
        dm(SPCTL0) = r0;

        //=============================================================
        //
        // Configure SPORTs 1 & 2 as transmitter (output to DACs 1-4)
```

```
        //
        //    SPTRAN = Transmit on serial port
        //    OPMODE = I2S mode
        //    SLEN24 = 24 bit of data in each 32-bit word
        //    SPEN_A = Enable data channel A
        //    SPEN_B = Enable data channel B
        //
        //-----------------------------------------------------------------

        r0 = SPTRAN | OPMODE | SLEN24 | SPEN_A | SPEN_B;
        dm(SPCTL1) = r0;
        dm(SPCTL2) = r0;

        rts;

_initSPORT.end:
```

6.7.5 init1835viaSPI.asm

```
#include <def21369.h>
#include "ad1835.h"

.global _init1835viaSPI;


//=============================================================

.section/dm seg_dmda;

.var spi_semaphore;
.var config_tx_buf[]=           // Buffer of configuration data
            WR | DACCTRL1 | DACI2S | DAC24BIT | DACFS48,
            WR | DACCTRL2, // e.g.: | DACMUTE_R1 | DACMUTE_L2,
            WR | DACVOL_L1 | DACVOL_MAX,
            WR | DACVOL_R1 | DACVOL_MAX,
            WR | DACVOL_L2 | DACVOL_MAX,
            WR | DACVOL_R2 | DACVOL_MAX,
            WR | DACVOL_L3 | DACVOL_MAX,
            WR | DACVOL_R3 | DACVOL_MAX,
            WR | DACVOL_L4 | DACVOL_MAX,
            WR | DACVOL_R4 | DACVOL_MAX,
            WR | ADCCTRL1 | ADCFS48,
            WR | ADCCTRL2 | ADCI2S | ADC24BIT,
            WR | ADCCTRL3 | IMCLKx2 | PEAKRDEN;

//=============================================================

.section/pm seg_pmco;

_init1835viaSPI:

    //-------------------------
    // Clear SPTFLG and SPICTL regs to start
    r0 = 0;
    dm(SPICTL)=r0;
    dm(SPIFLG)=r0;

    //-------------------------------------------------------------
    // Writing TXFLSH and RXFLSH bits in SPICTL clear the SPI
    // transmit and receive FIFOs, respectively.
    r0 = dm(SPICTL);
    r1 = (TXFLSH | RXFLSH );
    r0 = r0 OR r1;
    dm(SPICTL)=r0;
```

```
//-------------------------------
// Setup the baud rate to 500 KHz
r0 = 100;
dm(SPIBAUD) = r0;


//-----------------------------------------------
// Set the SPIFLG register to FLAG3 (0xF708)
r0 = 0xF708;
dm(SPIFLG) = r0;


//-------------------------------------------------------
// Now set the SPI control register
r0 = (SPIEN |    // enable the port
      SPIMS |    // set SHARC as SPI master
      MSBF |     // send MSB first
      WL16 |     // word length = 16 bits
      TIMOD1);   // Initialize SPI port to begin
                 // transmitting when DMA is enabled
dm(SPICTL) = r0;


//-----------------------------------------
// Set up DAG registers to transmit via SPI
i4 = config_tx_buf;
m4 = 1;


//-------------------------------
// Set up loop to transmit data
lcntr = LENGTH(config_tx_buf), do word_sent until lce;

    // Send a word
    r0=dm(i4,m4);
    dm(TXSPI)=r0;

    // Wait until "SPI transfer complete" status bit
    //   in SPISTAT (SPIFE) indicates that we can send more
    do checkIfTXisDone until TF;

        ustat3 = dm(SPISTAT);
        BIT TST ustat3 SPIFE;

    checkIfTXisDone:
    nop;

    // Wait an extra 100 cycles to meet the timing
```

```
        //    requirements of the AD1835A
        lcntr = 100, do pauseFor1835 until lce;
        pauseFor1835:
        nop;


    word_sent:
    nop;

/*
    //-----------------------------------------
    // Flush SPI buffers after initialization
    //   You may want to do this before sending
    //   other SPI commands to guarantee that
    //   you have not accidentally left data in
    //   the transmit or receive FIFOs.
    r0 = dm(SPICTL);
    r1 = (TXFLSH | RXFLSH );
    r0 = r0 OR r1;
    dm(SPICTL)=r0;
*/

_init1835viaSPI.end:
    rts;
```

6.7.6 interrupts.asm

```
////////////////////////////////
/*******************************/
/*                             */
/*  Fachhochschule Jena        */
/*  FB ET/IT                   */
/*  Jürgen Döffinger (631551)  */
/*                             */
/*  Projekt:    Equalizer      */
/*  Datei:      interrupts.asm */
/*  Version:    2012-01-11-01  */
/*                             */
/*******************************/
////////////////////////////////



// externe Funktionen einbinden
.extern _main;
.extern _audio;
.extern _button1;
.extern _button2;
.extern _button3or4;



// 21369 Interrupt Vector Table
.section/pm seg_rth;



__EMUI:       // 0x00: Emulator interrupt (highest priority, read-only, non-maskable)
        nop;
        nop;
        nop;
        nop;
__RSTI:         // 0x04: Reset (read-only, non-maskable)
        nop;    // <-- (this line is not executed)
        jump _main;
        nop;
        nop;
__IICDI:        // 0x08: Illegal Input Condition Detected
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
```

```
__SOVFI:        // 0x0C: Status loop or mode stack overflow or PC stack full
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__TMZHI:        // 0x10: Core timer interrupt (higher priority option)
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__SPERRI:         // 0x14:
        jump (pc,0); jump (pc,0); jump (pc,0); jump (pc,0);

__BKPI:         // 0x18: Hardware breakpoint interrupt
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__res1I:         // 0x1C: (reserved)
        jump (pc,0); jump (pc,0); jump (pc,0); jump (pc,0);


__IRQ2I:        // 0x20: IRQ2 is asserted
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__IRQ1I:        // 0x24: IRQ1 is asserted
        jump _button1;
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__IRQ0I:        // 0x28: IRQ0 is asserted
        jump _button2;
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__DAIHI:        // 0x2C: DAI interrupt (higher priority option)
__P0I :
        jump _button3or4;
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
```

```
__SPIHI:
__P1I :          // 0x30: SPI transmit or receive (higher priority option)
__SPII :
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__GPTMR0I:       // 0x34: General Purpose timer 0 interrupt
__P2I :
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__SP1I:          // 0x38: SPORT 1 interrupt
__P3I :
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__SP3I:          // 0x3C: SPORT 3 interrupt
__P4I :
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__SP5I:          // 0x40: SPORT 5 interrupt
__P5I :
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__SP0I:          // 0x44: SPORT 0 interrupt
__P6I :
        jump _audio;
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__SP2I:          // 0x48: SPORT 2 interrupt
__P7I :
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
```

```
        jump (pc,0);

__SP4I:         // 0x4C: SPORT 4 interrupt
__P8I :
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);
        jump (pc,0);

__EP0I:     // 0x50: External port0 interrupt. Thats the only label we're using here
__P9I_:

________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__GPTMR1I:______// 0x54: General Purpose timer 1 interrupt
__P10I_:
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__SP7I:_________// 0x58: serial port 7 interrupt
__P11I_:
________jump (pc,0); jump (pc,0); jump (pc,0); jump (pc,0);

__DAILI:________// 0x5C: DAI interrupt (lower priority option)
__P12I_:
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__EP1I:_________// 0x60: External port1 interrupt
__P13I_:
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__DPII:_________// 0x64: DPI interrupt
__P14I_:
________jump (pc,0); jump (pc,0); jump (pc,0); jump (pc,0);

_MTMI:_________// 0x68: Memory to Memory interface interrupt
__P15I_:
________jump (pc,0); jump (pc,0); jump (pc,0); jump (pc,0);
```

```
__SP6I:__________//_0x6C:_serial_port_6_interrupt
__P16I_:
________jump_(pc,0);_jump_(pc,0);_jump_(pc,0);_jump_(pc,0);

__GPTMR2I:______//_0x70:_General_Purpose_timer_2_interrupt
__P17I_:
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);

__SPILI:________//_0x74:_SPI_transmit_or_receive_(lower_priority_option)
__P18I_:
__SPIBI:
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);

__CB7I:_________//_0x78:_Circular_buffer_7_overflow_exception
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);

__CB15I:________//_0x7C:_Circular_buffer_15_overflow_exception
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);

__TMZLI:________//_0x80:_Core_timer_interrupt_(lower_priority_option)
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);

__FIXI:_________//_0x84:_Fixed-point_overflow_exception
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);

__FLTOI:________//_0x88:_Floating-point_overflow_exception
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);
________jump_(pc,0);
```

```
__FLTUI:________// 0x8C: Floating-point underflow exception
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);


__FLTII:________// 0x90: Floating-point invalid exception
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__EMULI:________// 0x94: Emulator low priority interrupt
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__SFT0I:________// 0x98: User software interrupt 0
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__SFT1I:________// 0x9C: User software interrupt 1
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__SFT2I:________// 0xA0: User software interrupt 2
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);

__SFT3I:________// 0xA4: User software interrupt 3 (lowest priority)
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
________jump (pc,0);
```

6.7.7 filter.asm

```
/////////////////////////////////
/*******************************/
/*                             */
/*  Fachhochschule Jena        */
/*  FB ET/IT                   */
/*  Jürgen Döffinger (631551)  */
/*                             */
/*  Projekt:    Equalizer      */
/*  Datei:      filter.asm     */
/*  Version:    2012-02-15-01  */
/*                             */
/*******************************/
/////////////////////////////////



// Headerdateien einbinden
#include <def21369.h>



// globale Variablen und Funktionen deklarieren
.global _init_filter;
.global koeff;
.global filter_active;
.global filter_db;
.global _set_filter;
.global volume;
.global volume_db;



// Konstante deklarieren
#define N 513


// Datenspeicher - Section
.section  /dm seg_dmda;

.VAR koeff_orig[N*2*11] = "./Filter/filter.dat";   // Koeffizienten der Einzelfilter mit der Verstärkung 1
.VAR filter_db[11] = 0,0,0,0,0,0,0,0,0,0,0;        // Verstärkungswerte in dB des jeweiligen Bandpasses

// Programmspeicher - Section
.section  /pm seg_pmda;

.VAR koeff[N*2];   // Koeffizienten des Gesamtfilters (für Faltung mit Eingangssignal)
```

```
.VAR filter[11*2] = 1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0;   // voreingestellte Verstä
.VAR filter_active = 0;
.VAR db2filter[] = "./Filter/db2filter.dat";   // Look up Table zur Wandlung des db-Wertes in den Faktor der Verstärkung
.VAR volume_db = 0; // Vorverstärkung in dB
.VAR volume = 1.0;  // Vorverstärkungsfaktor


// Code - Section
.section /pm seg_pmco;


/********************   START _reset_filter    ********************/

_init_filter:

  // Circular Buffer Addressing Enable and  Processor Element Y Enable
  bit set mode1 CBUFEN | PEYEN;

  // Adressgenerator für Koeffizientenvektor initialisieren
  B8 = koeff;
  M8 = 2;
  L8 = N*2;

  // Koeffizientenvektor löschen
  r0  = N;
  f10 = 0.0;
  lcntr = r0, do _reset_filter_clear until lce;
  _reset_filter_clear: pm(i8,m8) = f10;

  // Circular Buffer Addressing Disable and Processor Element Y Disable
  bit clr mode1 CBUFEN | PEYEN;

  //    Koeffizienten berechnen
  call _calc_coefficients;

_init_filter.end:  rts;

/********************   END _reset_filter    ********************/



/********************   START _calc_coefficients    ********************/

_calc_coefficients:

  //  Circular Buffer Addressing Enable and Processor Element Y Enable
  bit set mode1 CBUFEN | PEYEN;

  //  Adressgeneratoren initialisieren
```

```
  B0 = koeff_orig;
  M0 = 2;
  L0 = N*2*11;

  B8 = koeff;
  M8 = 2;
  L8 = N*2;

  B9 = filter;
  M9 = 2;
  L9 = @filter;

  // Berechnung der Koeffizienten mit entsprechender Verstärkung
  r1 = (@koeff >> 1) -1;
  lcntr = r1, do _calc_coefficients_loop_1 until lce;

    r0 = r0 xor r0;
    r4 = r4 xor r4;
    r8 = r8 xor r8;
    r12 = r12 xor r12;

    lcntr = 11, do _calc_coefficients_loop_2 until lce;
    _calc_coefficients_loop_2: f8 = f0 * f4, f12 = f8 + f12, f0 = dm(i0,m0), f4 = pm(i9,m9);
    f8 = f0 * f4, f12 = f8 + f12;
    f12 = f8 + f12;

  _calc_coefficients_loop_1: pm(i8,m8) = f12;


  //    Circular Buffer Addressing Disable and Processor Element Y Disable
  bit clr mode1 CBUFEN | PEYEN;

_calc_coefficients.end: rts;

/*******************   END _calc_coefficients    *******************/



/*******************   START _set_filter    *******************/

_set_filter:

  // mittels LUT dB-Werte des ausgewählten Filters in Verstärkungsfaktoren wandeln

  // Wert in dB in r12 laden
  // Filter in r8 merken
  r8 = pm(filter_active);
  r12 = 0;
  comp(r8,r12);
```

```
  if NE jump _set_filter_db;
  r12 = pm(volume_db);
  jump _set_filter_0;

_set_filter_db:

  // Adresse des Filterwertes des ausgewählten Filters ermitteln
  r12 = filter_db;
  r9 = r8 + r12;
  i15 = r9;
  r12 = pm(i15,0);

_set_filter_0:

  // dB-Wert in Faktor wandeln
  r11 = db2filter;
  r13 = 20;
  r11 = r11 + r13;
  r11 = r11 + r12;
  i15 = r11;
  f11 = pm(i15,0);

  // Filterwert speichern
  r12 = 0;
  comp(r8,r12);
  if NE jump _set_filter_1;
  pm(volume) = f11;
  jump _set_filter_calc;

_set_filter_1:

  r8 = lshift r8 by 1;
  r9 = 2;
  r8 = r8 - r9;
  r9 = filter;
  r9 = r8 + r9;
  i14 = r9;
  pm(i14,1) = f11;
  pm(i14,0) = f11;

_set_filter_calc:

  // Koeffizienten des Gesamtfilters berechnen
  call _calc_coefficients;

_set_filter.end: rts;

/********************   END _set_filter    ********************/
```

6.7.8 audio.asm

```
////////////////////////////////
/*******************************/
/*                             */
/*  Fachhochschule Jena        */
/*  FB ET/IT                   */
/*  Jürgen Döffinger (631551)  */
/*                             */
/*  Projekt:    Equalizer      */
/*  Datei:      audio.asm      */
/*  Version:    2012-01-11-01  */
/*                             */
/*******************************/
////////////////////////////////


// Headerdateien einbinden
#include <def21369.h>



// Konstanten deklarieren und initialisieren
#define FRAMESIZE 2
#define N 513


// Variablen für den Datenspeicher deklarieren
.section  /dm seg_dmda;

.VAR inbuf[FRAMESIZE];   // Zwischenspeicher Eingangssignal einmal rechter und linker Kanal
.VAR outbuf[FRAMESIZE]; // Zwischenspeicher Ausgangssignal einmal rechter und linker Kanal
.VAR buffer[N*2];        // Zwischenspeicher der letzten 513 Eingangssignale (rechter und linker Kanal) für die Faltung



// Code - Section
.section /pm seg_pmco;

// globale Funktionen und Variablen deklarieren
.global _audio;
.global _init_audio;
.global buffer;



// externe Funktionen und Variablen einbinden
.extern koeff;
```

```
.extern volume;


/*********************   START _init_audio    ******************/

_init_audio:

  B5 = buffer;
  M5 = 2;
  L5 = N*2;

_init_audio.end: rts;

/*********************   END _init_audio    ******************/



/*********************   START _audio (ISR)    ******************/

_audio:

  r10 = dm(RXSP0A);  // Read new left or right sample from ADC

  // 24-bit Integer in 32 bit Fließkommazahl wandeln
  r10 = lshift r10 by 8;
  f10 = float r10;

  // Vorverstärkung
  f12 = pm(volume);
  f10 = f10 * f12;

  //  Ermittlung des Kanals (rechts/links) und in entsprechende
  //  Unterroutine springen
  r15 = dm(DAI_PIN_STAT);
  btst r15 by 7;
  if not sz jump right_ch;


left_ch:

  // Sample in 1. Wert des Eingangspuffers schreiben und zuletzt
  // berechnetes Sample ausgeben.
  dm(inbuf) = f10;
  f10 = dm(outbuf);
  jump send_out;


right_ch:
```

```
  // Sample in 2. Wert des Eingangspuffers schreiben und
  // Samples filtern und anschließend zuletzt
  // berechnetes Sample ausgeben.
  dm(inbuf+1) = f10;
  f10 = dm(outbuf+1);
  call _calc_audio;

send_out:

        // 32-bit Fließkommazahl in 24-bit Integer wandeln
        r10 = fix f10;
        r10 = lshift r10 by -8;

        dm(TXSP1A) = r10;    // Write to DAC1
        dm(TXSP1B) = r10;    // Write to DAC2
        dm(TXSP2A) = r10;    // Write to DAC3
        dm(TXSP2B) = r10;    // Write to DAC4

_audio.end: rti;

/*********************   END _audio (ISR)    ********************/



/*********************   START _calc_audio    ********************/

_calc_audio:

  // Circular Buffer Addressing Enable and Processor Element Y Enable
  bit set mode1 CBUFEN | PEYEN ;

  // Adressgenerator der Koeffizienten zurücksetzen
  i8 = koeff;

  // aktuelles Sample (rechter und linker Kanal) in
  // den Puffer schreiben
  f0 = dm(inbuf);
  dm(i5,m5) = f0;

  // Register vor der Berechnung löschen
  r12 = r12 xor r12;
  r8  = r8 xor r8;
  r4  = r4 xor r4;
  r0  = r0 xor r0;
  r1  = r1 xor r1;
  r2  = r2 xor r2;
  r5  = r5 xor r5;
  r6  = r6 xor r6;
```

```
  //    Faltung des Filters mit dem Puffer durchführen
  f0 = dm(i5,m5), f4 = pm(i8,m8);
  r1 = (@koeff >> 1) -1;
  lcntr = r1, do _calc_audio_loop until lce;
  _calc_audio_loop: f12 = f0 * f4, f8 = f8 + f12, f0 = dm(i5,m5), f4 = pm(i8,m8);
  f12 = f0 * f4, f8 = f8 + f12;
  f8 = f8 + f12;

  // berechnetes Sample (linker und rechter Kanal) in den Ausgangspuffer schreiben
  dm(outbuf) = f8;

  // Circular Buffer Addressing Disable and Processor Element Y Disable
  bit clr mode1 CBUFEN | PEYEN;

_calc_audio.end: rts;

/*********************   END _calc_audio   ********************/
```

6.7.9 buttons.asm

```
////////////////////////////////
/*******************************/
/*                             */
/*  Fachhochschule Jena        */
/*  FB ET/IT                   */
/*  Jürgen Döffinger (631551)  */
/*                             */
/*  Projekt:    Equalizer      */
/*  Datei:      buttons.asm    */
/*  Version:    2012-01-11-01  */
/*                             */
/*******************************/
////////////////////////////////



// Headerdateien einbinden
#include <def21369.h>
#include <SRU.h>



// Code - Section
.section /pm seg_pmco;



// globale Funktionen deklarieren
.global _init_Buttons;
.global _button1;
.global _button2;
.global _button3or4;



// externe Funktionen und Variablen einbinden
.extern filter_active;
.extern filter_db;
.extern _set_LEDs;
.extern led_cache;
.extern _set_filter;
.extern _set_LEDs_from_filter;
.extern volume;
.extern volume_db;
```

```
/********************   START _init_Buttons    *******************/

_init_Buttons:

  // The DAI signals can be routed to the miscellaneous channels (MISCAx) which are
  // associated with DAI interrupts, and used as inputs from the Push Buttons.
  ustat3=SRU_EXTMISCA1_INT  | SRU_EXTMISCA2_INT;     //unmask individual interrupts
  dm(DAI_IRPTL_PRI)=ustat3;
  ustat3=SRU_EXTMISCA1_INT  | SRU_EXTMISCA2_INT;     //make sure interrupts latch on the rising edge
  dm(DAI_IRPTL_RE)=ustat3;


  //Set up the IRQ pins to use with the Push Buttons
  ustat3=dm(SYSCTL);
  bit set ustat3 IRQ0EN|IRQ1EN;
  dm(SYSCTL)=ustat3;

  bit set mode2 IRQ0E|IRQ1E;


  //Pin Assignments in SRU_PIN3 (Group D)

  //assign pin buffer 19 low so it is an input
  SRU(LOW,DAI_PB19_I);

  //assign pin buffer 20 low so it is an input
  SRU(LOW,DAI_PB20_I);


  //Route MISCA signals in SRU_EXT_MISCA (Group E)

  //route so that DAI pin buffer 19 connects to MISCA1
    SRU(DAI_PB19_O,MISCA1_I);

  //route so that DAI pin buffer 20 connects to MISCA2
  SRU(DAI_PB20_O,MISCA2_I);

  //Pin Buffer Disable in SRU_PINEN0 (Group F)
  //assign pin 19 low so it is an input
  SRU(LOW,PBEN19_I);

  //assign pin 20 low so it is an input
  SRU(LOW,PBEN20_I);

  //Set up the interrupt hardware
  // Enable Interrupts to handle push buttons
  bit clr irptl IRQ1I|IRQ0I;
  bit set imask IRQ1I|IRQ0I|DAIHI;     //enable IRQ interrupts
```

```
  //Set up interrupt priorities
  bit set mode1 IRPTEN;   //enable global interrupts

  ustat3 = SRU_EXTMISCA1_INT  | SRU_EXTMISCA2_INT ;
  dm(DAI_IRPTL_PRI)=ustat3;   //unmask individual interrupts
  dm(DAI_IRPTL_RE)=ustat3;    //make sure interrupts latch on the rising edge

_init_Buttons.end: rts;

/*******************   END _init_Buttons   *******************/



/*******************   START button1   *******************/

_button1:

  // Filterwert in dB inkrementieren. Werden 20 dB überschritten,
  // dann Filterwert auf -20 dB setzen. Wenn das gewählte Filter
  // das Filter 0 ist, dann den Vorverstärker in dB inkrementieren und
  // ebenfalls bei überschreiten von 20 dB auf -20 dB setzen.
  // Anschließend Gesamtfilter neu berechnen und LEDs auf den dB-Wert
  // des Filters bzw. des Vorverstärkers einstellen.

  r8 = pm(filter_active);
  r12 = 0;
  comp(r8,r12);
  if NE jump _button1_1;
  r12 = pm(volume_db);
  r8 = volume_db;
  jump _button1_2;

_button1_1:

  r9 = filter_db;
  r8 = r8 + r9;
  i14 = r8;
  r12 = pm(i14,0);

_button1_2:

  r12 = r12 + 1;
  r9 = 21;
  comp(r9,r12);
  if GT jump _button1_3;
  r12 = -20;

_button1_3:
```

```
  i14 = r8;
  pm(i14,0) = r12;

  call _set_filter;
  call _set_LEDs_from_filter;

_button1.end: rti;

/********************   END button1    ********************/



/********************   START button2    ********************/

_button2:

  //  Filterwert in dB dekrementieren. Werden -20 dB unterschritten,
  //  dann Filterwert auf 20 dB setzen. Wenn das gewählte Filter
  //  das Filter 0 ist, dann den Vorverstärker in dB dekrementieren und
  //  ebenfalls bei unterschreiten von -20 dB auf 20 dB setzen.
  //  Anschließend Gesamtfilter neu berechnen und LEDs auf den dB-Wert
  //  des Filters bzw. des Vorverstärkers einstellen.


  r8 = pm(filter_active);
  r12 = 0;
  comp(r8,r12);
  if NE jump _button2_1;
  r12 = pm(volume_db);
  i15 = volume_db;
  jump _button2_2;

_button2_1:

  r9 = filter_db;
  r8 = r8 + r9;
  i15 = r8;
  r12 = pm(i15,0);

_button2_2:

  r12 = r12 - 1;
  r9 = -21;
  comp(r9,r12);
  if LT jump _button2_3;
  r12 = 20;

_button2_3:
```

```
  pm(i15,0) = r12;

  call _set_filter;
  call _set_LEDs_from_filter;

_button2.end: rti;

/********************   END button2    ********************/



/********************   START button3or4    ********************/

_button3or4:

  //test for SRU_EXTMISCA1_INT
  ustat4=dm(DAI_IRPTL_H);
  it tst ustat4 SRU_EXTMISCA1_INT;   //SRU_EXTMISCA1_INT represents bit 29
  if TF call (_button3);

  //test for SRU_EXTMISCA2_INT
  bit tst ustat4 SRU_EXTMISCA2_INT;   //SRU_EXTMISCA2_INT represents bit 30
  if TF call (_button4);

  //    aktiven Filter über LEDs anzeigen
  r0 = pm(filter_active);
  pm(led_cache) = r0;
  call _set_LEDs;

_button3or4.end: rti;

/********************   END button3or4    ********************/



/********************   START button3    ********************/

_button3:

  // Filterauswahl inkrementieren und bei überschreiten der Zahl 10
  // den ausgewählten Filter auf 0 (Vorverstärker) setzen.

  r8 = pm(filter_active);
  r0 = r8 + 1;
  r4 = 11;
  comp(r0, r4);
  if LT jump _button3_1;
  r0 = 0;
```

```
_button3_1:

        pm(filter_active) = r0;

_button3.end: rts;

/********************   END button3    ********************/



/********************   START button4    ********************/

_button4:

  // Filterauswahl dekrementieren und bei unterschreiten der Zahl 0
  // den ausgewählten Filter auf 10 (Filter 10) setzen.

  r8 = pm(filter_active);
  r0 = r8 - 1;
  r4 = 0;
  comp(r0, r4);
  if GE jump _button4_1;
  r0 = 10;

_button4_1:

  pm(filter_active) = r0;

_button4.end: rts;

/********************   END button4    ********************/
```

6.7.10 leds.asm

```
////////////////////////////////
/*******************************/
/*                             */
/*  Fachhochschule Jena        */
/*  FB ET/IT                   */
/*  Jürgen Döffinger (631551)  */
/*                             */
/*  Projekt:    Equalizer      */
/*  Datei:      leds.asm       */
/*  Version:    2012-01-11-01  */
/*                             */
/*******************************/
////////////////////////////////



// Headerdateien einbinden
#include <def21369.h>
#include <SRU.h>



// Programmspeicher - Section
.section  /pm seg_pmda;

.var led_cache = 0;     // Übergabeparameter in dem der codierte Wert übergeben wird (Bit0 = LED1, Bit1 = LED2 usw. 0 = aus, 1 = ein)



// Code - Section
.section /pm seg_pmco;



// globale Funktionen deklarieren
.global _init_LEDs;
.global _set_LEDs;
.global _set_LEDs_from_filter;
.global led_cache;



// externe Funktionen und Variablen einbinden
.extern filter_active;
.extern filter_db;
.extern volume_db;
```

```
/********************   START _init_LEDs   ********************/

_init_LEDs:

  //Setting the SRU and route so that Flag pins connects to DPI pin buffers.
  //Use Flags 4 to 15 only. Flags 0 to 3 not available on the DPI.

  SRU(FLAG6_O, DPI_PB08_I);     // Connect Flag6 output to DPI_PB08 input (LED1)
  SRU(FLAG7_O, DPI_PB13_I);     // Connect Flag7 output to DPI_PB13 input (LED2)
  SRU(FLAG4_O, DPI_PB06_I);     // Connect Flag4 output to DPI_PB06 input (LED3)
  SRU(FLAG5_O, DPI_PB07_I);     // Connect Flag5 output to DPI_PB07 input (LED4)
  SRU(FLAG8_O, DPI_PB14_I);     // Connect Flag8 output to DPI_PB14 input (LED5)

  SRU(LOW, DAI_PB15_I);         // Connect Input LOW to LED6
  SRU(LOW, DAI_PB16_I);         // Connect Input LOW to LED7


  //Enabling the Buffer using the following sequence: High -> Output, Low -> Input

  SRU(HIGH, DPI_PBEN08_I);
  SRU(HIGH, DPI_PBEN13_I);
  SRU(HIGH, DPI_PBEN06_I);
  SRU(HIGH, DPI_PBEN07_I);
  SRU(HIGH, DPI_PBEN14_I);
  SRU(HIGH, PBEN15_I);
  SRU(HIGH, PBEN16_I);

  //Setting flag pins
  bit set flags FLG3O|FLG4O|FLG5O|FLG6O|FLG7O|FLG8O;

  //Clearing flag pins
  bit clr flags FLG3|FLG4|FLG5|FLG6|FLG7|FLG8;

_init_LEDs.end: rts;

/********************   END _init_LEDs   ********************/



/********************   START _set_LEDs   ********************/

_set_LEDs:

  // Es ist darauf zu achten das für die LEDs 1 bis 7 in der
  // Variablen led_cache die entsprechende Codierung übergeben wird.
  // Dabei steht 0 für Aus und 1 für Ein.
```

```
  // Es sind die untersten 7 bits in entsprechender
  // Reihenfolge für die LEDs zu nutzen.

  // Alle LEDs ausschalten
  call _LEDs_off;

  // Code laden
  r8 = pm(led_cache);

  // entsprechende LEDs einschalten

_set_LEDs_1:

  BTST r8 by 0;
  IF SZ jump _set_LEDs_2;
  bit set flags FLG4;

_set_LEDs_2:

  BTST r8 by 1;
  IF SZ jump _set_LEDs_3;
  bit set flags FLG5;

_set_LEDs_3:

  BTST r8 by 2;
  IF SZ jump _set_LEDs_4;
  bit set flags FLG6;

_set_LEDs_4:

  BTST r8 by 3;
  IF SZ jump _set_LEDs_5;
  bit set flags FLG7;

_set_LEDs_5:

  BTST r8 by 4;
  IF SZ jump _set_LEDs_6;
  bit set flags FLG8;

_set_LEDs_6:

  BTST r8 by 5;
  IF SZ jump _set_LEDs_7;
  SRU(HIGH, DAI_PB15_I);

_set_LEDs_7:
```

```
  BTST r8 by 6;
  IF SZ jump _set_LEDs.end;
  SRU(HIGH, DAI_PB16_I);

_set_LEDs.end: rts;

/********************   END _set_LEDs    ********************/



/********************   START _LEDs_off    ********************/

_LEDs_off:

  // Alle LEDs ausschalten
  bit clr flags FLG4|FLG5|FLG6|FLG7|FLG8;
  SRU(LOW, DAI_PB15_I);
  SRU(LOW, DAI_PB16_I);

_LEDs_off.end: rts;

/********************   END _LEDs_off    ********************/



/********************   START _set_LEDs_from_filter    ********************/

_set_LEDs_from_filter:

  // dB-Wert auslesen
  r8 = pm(filter_active);
  r9 = 0;
  comp(r8,r9);
  if NE jump _set_LEDs_from_filter_0;
  r12 = pm(volume_db);
  jump _set_LEDs_from_filter_0a;

_set_LEDs_from_filter_0:

  r9 = filter_db;
  r8 = r8 + r9;
  i15 = r8;
  r12 = pm(i15,0);

_set_LEDs_from_filter_0a:

  // Prüfen ob ein positiver oder negativer Wert vorliegt.
  // Bei negativen Wert diesen in einen positiven Wert wandeln und
  // für die Anzeige (LED7 0-positiv 1-negativ) den Wert 64
```

```
  // dazu addieren, damit LED7 anzeigt das es ein negativer Wert ist.
  r9 = 0;
  comp(r12,r9);
  if GE jump _set_LEDs_from_filter_1;
  r9 = 64;


  r12 = -r12;
  r12 = r9 + r12;

_set_LEDs_from_filter_1:

  // LED-Wert in den LED-Speicher schreiben und
  //  LEDs setzen
  pm(led_cache) = r12;
  call _set_LEDs;

_set_LEDs_from_filter.end: rts;

/********************   END _set_LEDs_from_filter    ********************/
```

Literatur

[1] Adsp-21369 ez-kit lite evaluation system manual. http://www.analog.com/static/imported-files/eval_kit_manuals_legacy/51593613383456ADSP_21369_EZ_KIT_Lite_Evaluation_System_Manual_Rev_1.pdf, 2005.

[2] In-circuit flash programming on sharc® processors. http://www.analog.com/static/imported-files/application_notes/EE.223.In.Circuit.Flash.Programming.on.SHARC.Processors.Rev.2.02.07.pdf, 2007.

[3] Adsp-21367/adsp-21368/adsp-21369 sharc processors data sheet. http://www.analog.com/static/imported-files/data_sheets/ADSP-21367_21368_21369.pdf, 2009.

[4] Sharc processor programming reference (includes the adsp-2136x/adsp-2137x/adsp-214xx processors), rev. 2.2, march 2011. http://www.analog.com/static/imported-files/processor_manuals/ADSP-2136x_PRM_Rev_2_2.pdf, 2011.

[5] Dr-Ing. Seyed Ali Azizi. *Entwurf und Realisierung digitaler Filter*. Oldenbourg Verlag GmbH, 5., verb u. erw. aufl. edition, 1990.

[6] Prof. Dr.-Ing. habil Rüdiger Hoffmann. *Grundlagen der Frequenzanalyse*. expert verlag, 2., durchgesehene auflage edition, 2005.

[7] Prof. Dr.-Ing. Josef Hoffmann. *Spektrale Analyse mit MatLab und Simulink*. Oldenbourg Wissenschaftsverlag GmbH, 2011.

[8] Prof. Dr.-Ing. Otto Mildenberger. *Entwurf analoger und digitaler Filter*. Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1992.

[9] Prof. Dr. Franz Quint Prof. Dr.-Ing. Josef Hoffmann. *Signalverarbeitung mit MATLAB und Simulink*. Oldenbourg Wissenschaftsverlag GmbH, 2007.

[10] Prof. Dipl.-Ing Herbert Wagner. Programmierbeispiele. http://www.et.fh-jena.de/wagner/dsp_programmbeispiele.htm, 2012.