BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Analog-Digital-Umsetzer und digitale Signale

Laborbericht

angefertigt von

Robby Kozok, Nic Frank Siebenborn, Pascal Kahlert

in dem Fachbereich VII – Elektrotechnik - Mechatronik - Optometrie –
für das Modul Digitale Signalverarbeitung III
der Beuth Hochschule für Technik Berlin im Studiengang
Elektrotechnik - Schwerpunkt Elektronische Systeme

Datum 1. Dezember 2015

Lehrkraft
Prof. Dr.-Ing Marcus Purat Beuth Hochschule für Technik

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Dokument protokoliert die Ergebnisse und Herangehensweise der Laborübung im Rahmen der Veranstaltung Digitale Signalverarbeitung III Labor. Als Vorbereitung auf diese Übung wurde sich theoretisch mit dem ADSPBF561EZ-KIT Lite sowie dem darauf eingesetzten Signalprozessor ADSP-BF561 beschäftigt. Außerdem wurden Funktionen in der Programmiersprache C erstellt. Im Folgenden wird auf dieser Vorbereitung aufbauend anhand der Aufgabenstellung der Lösungsweg erörtert.

Kapitel 2

Einlesen von Signalen

2.1 Durchführung

Im ersten Versuch soll der Umgang mit der Software VisualDSP++ erlernt werden. Dies geschieht mit einem Projekt, welches die Signale am Eingang des Codec zum Ausgang des Codec durchreicht. Diese werden dann am PC visualisiert, um den Eingang und den Ausgang vergleichen zu können.

Entsprechend der Aufgabenstellung wurde ein Projekt angelegt und der kompilierte Code auf das Evaluationboard (EVB) übertragen. Wir legten entsprechend der Aufgabenstellung ein Signal an den rechten Kanal des ADC1 an.
Dieses Signal wurde, wie alle weiteren mit dem *VI FFT & FRF & Scope* aufgezeichnet.

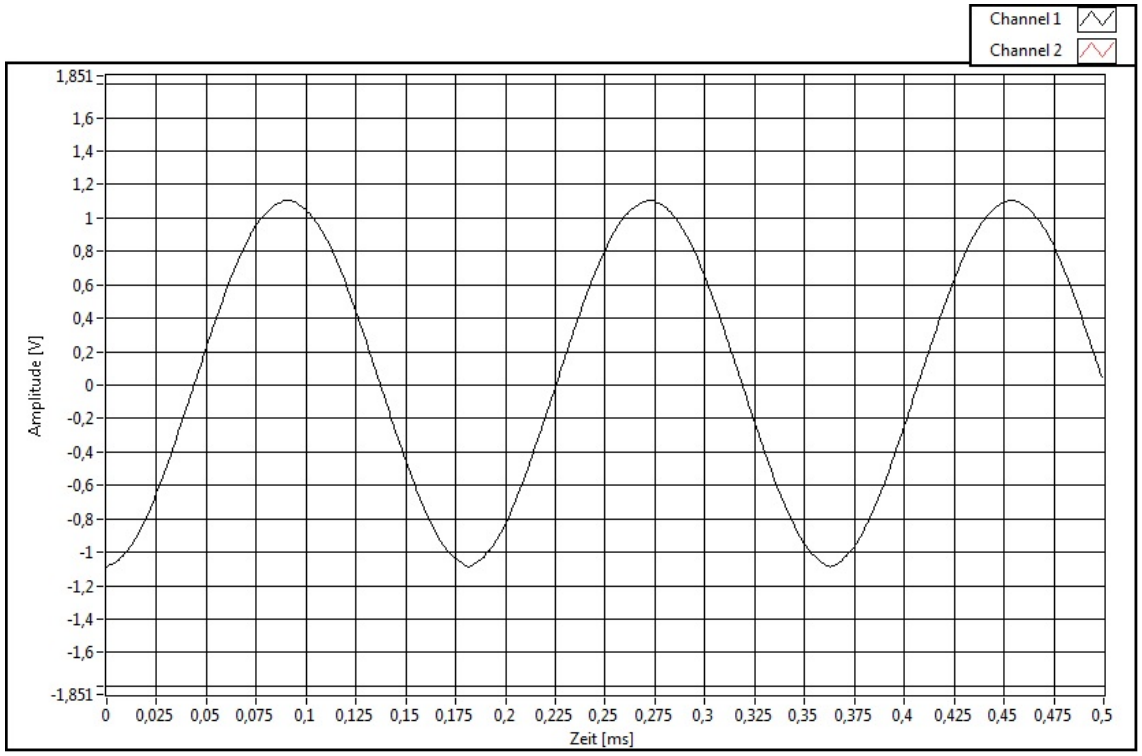

Abbildung 2.1: Darstellung des Signals am Eingang des Codec.

In der Vorbereitung wurde die Funktion copyData() erstellt, die sich wie folgt aufbaut:

```
1  #include "codeclib.h"
2  #include "copydata.h"
3
4
5  /*  @function   copyData
6  *   @brief      Kopiert die Audiodaten des Kanals "Internal ADC R0"
7  *               aus dem DMA-Lesepuffer
8  *               in den Speicherbereich iInput schreibt.
9  *   @param      input   Adresse der ersten Speicherstelle von input
10 *   @param      pWrite  Zeiger auf die Adresse von input
11 *   @param      size    Anzahl der übergebenen Werte
12 *   @return     void
13 */
14 void copyData(int *input, int **pWrite, int size) {
15
16     //Nimm den 5ten Wert aus iDMARxBuffer und schreibe diesen in den input.
17     **pWrite = iDMARxBuffer[4];
18
19     //Inkrementiere den Wert der zuletzt beschriebenen Adresse
20     (*pWrite)++;
21
22     // Wenn die obere Grenze size erreicht wurde müssen wir auf
23     // die Startadresse zurückspringen.
24     if(*pWrite == input + size)
25     {
26         *pWrite = input;
27     }
28
29 }
```

copydata.c

Wie man leicht sieht werden die Werte, die der Codec liefert, kopiert und in den iDMARx-Buffer fortlaufend geschrieben, bis dieser voll ist. In dem Fall wird an der ersten Position des Buffer erneut angefangen und die alten Werte werden überschrieben. Die Software VisualDSP++ biete die Möglichkeit den DSP zu debuggen.

Stoppt man nun den Durchlauf, können wie in der Aufgabenstellung beschrieben, die aktuellen Werte des iDMARxBuffer ausleitet werden und mit MatLab visualisieren werden.

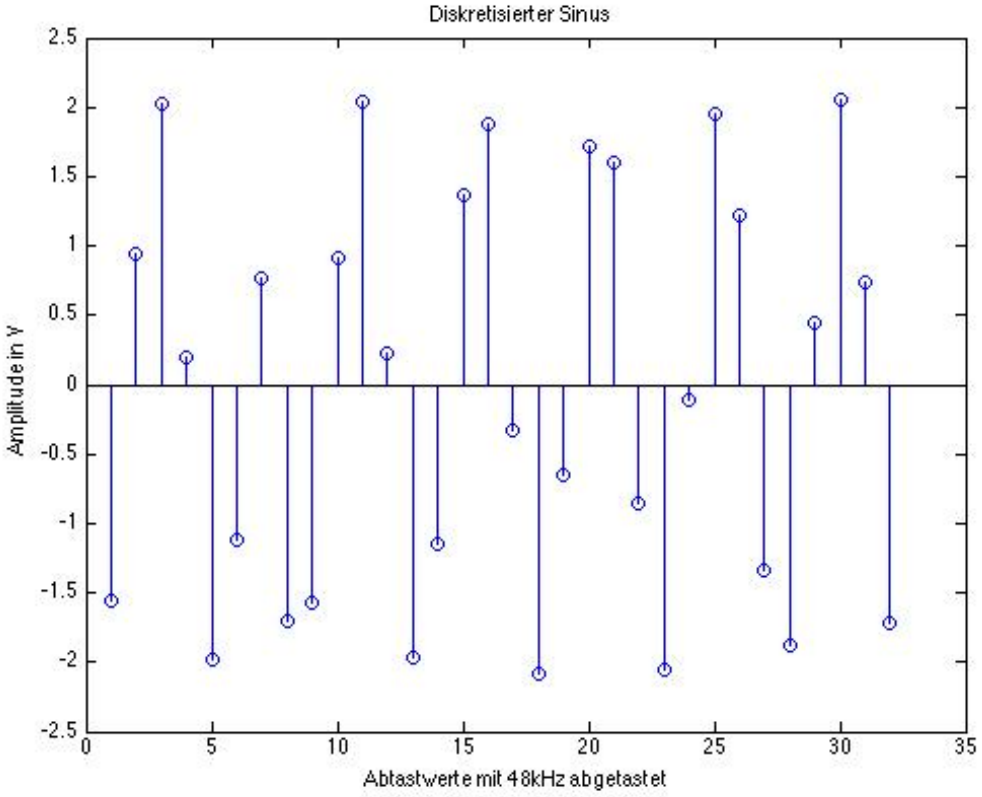

Abbildung 2.2: Darstellung der Datenreihe aus den Werten des DSP.

2.2 Auswertung

Aus Abbildung 2.1 war ein Sinussignal zu erwarten, in Abbildung 2.2 ist diese Sinussignal wieder zu erkennen.

Die Amplitude des Signals beträgt 1446560512 in der Registerdarstellung des DPS. Um diesen Wert exakt ermitteln zu können, wurde die Matlab-Funktion max() verwendet. Für die normierte Kreisfrequenz wird der Ansatz

$$\omega_0 = \frac{2\pi}{N} \tag{2.1}$$

verwendet und führt bei 18 Messwerten in 2 Perioden (es folgt $N = \frac{18}{2}$) zu

$$\omega_0 = \frac{2\pi}{9}$$

Da die Diskreten Amplitudenwerte im 32-bit-Integer Format vorliegen, müssen sie in eine Spannung umgerechnet werden. Hierzu sind die Daten des EVB erforderlich. Der Codec hat eine Auflösung von 24-bit bei einer Spitze-Spitze-Spannung $V_{ss} = 6,17V$.

Es gilt zur Berechnung der Amplitudenspannung V_{Amp}

$$V_{Amp} = \frac{V_{ADC} * V_{ss}}{2 * 2^{Registerbreite-1}} \tag{2.2}$$

Dies ergibt bei einem Registerwert von 1446560512 eine Spannung von $V_{ss_{max}} = 2,0579V$

Es ist an dieser Stelle nicht nachweisbar ob es sich bei dem gemessenen Maximalwert auch um das tatsächliche Maximum handelt, da der Abtastzeitpunkt nicht unbedingt der Zeitpunkt des lokalen Hochpunktes war. Auch ist die Frequenz des Ursprungssignals nur näherungsweise zu ermitteln, da die Abtastfrequenz kein ganzzahliges Vielfaches der Signalfrequenz ist. Um letztgenannten Fehler zu minimieren, wurden in erster Annäherung die Abtastwerte über 4 Perioden ermittelt. Die Frequenz des abgetasteten Sinus lässt sich mit

$$f_0 = \frac{f_s}{N} \tag{2.3}$$

zu $f_0 = 5,3kHz$ berechnen, da der Codec mit 48kHz abtastet.

Aus der Messung ergibt sich grafisch eine Amplitude von $V_{ss} = 1,1V$ und eine Frequenz von 5,5kHz. Die Abweichungen sind u.A. der Messung aus o.g. Gründen und des Ablesens geschuldet. Eine weitere Fehlerquelle bildet die interne Schaltung des EVB, so wie der Verstärker zwischen Eingang und ADC.

Kapitel 3

Ausgeben von Signalen

3.1 Durchführung

In dieser Aufgabe sollte der DSP ein Sinussignal generieren. Die Frequenz sollte dabei per Taster in 200Hz Schritten einstellbar sein. Die drei auf dem DSP angebrachten LEDs sollten die Frequenzen 200Hz, 400Hz und 4kHz darstellen.
Zur Realisierung dieser Aufgabe wurde in der Vorbereitung eine Funktion genSinus erstellt, die bei jedem Aufruf den nächsten Wert des Sinussignals ausgibt.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define PI 3.141592653 //Definition der Zahl Pi
5  #define FABTAST 48000  //Definition der Abtastrate des Codec
6
7  /*  @function genSinus
8  *   @brief  Diese Funktion generiert fortlaufend einen Sinussignal.
9  *
10 *   @param  A ist die Amplitude des gewünschten Sinussignal zwischen 0 und 1.
11 *   @param  Freq200 ist die gewünschte Frequenz des Sinussignal
12 *   in Schritten von 200Hz.
13 *   @return Ist der aktuell berechnete Wert des Sinussignals
14 */
15 float genSinus(float A, short Freq200)
16 {
17     float sinusValue;
18     float omegaNormNeu;
19     static float omegaNorm = 0;
20
21     //Die Stellung des Zeigers wird neu berechnet.
22     omegaNormNeu = 2 * PI * ((Freq200 * 200.)/FABTAST);
23     omegaNorm += omegaNormNeu;
24
25     //Ermittlung des Sinuswertes anhand des neuen Zeigers.
26     sinusValue = A * sin(omegaNorm);
27
28     //Bei durchschreiten einer Periode von omegaNorm wird das Signal
29     //um 2Pi zurückgesetzt
30     if(omegaNorm > 2 * PI)
31     {
32         omegaNorm -= 2 * PI;
33     }
34
35     return sinusValue;
36 }
```

gensinus.c

Die Funktion ist so Implementiert, dass sie den jeweiligen Zeitpunkten einen entsprechenden Sinuswert zuweist und diesen zurückweißt. Als Übergabeparameter sind zum einen die Amplitude und zum anderen die Frequenz vorgesehen, wobei die Frequenz in 200Hz Schritten übergeben wird.
Das neue ω_{Norm}, also wird entsprechend mit

$$\omega_{Norm} = 2\pi * f \tag{3.1}$$

ermittelt und normiert.

Der Aufruf durch den Timer-Interrupt stellt dabei die Periodizität sicher. Das alte ω_{Norm} wird dann mit dem neuen ω_{Norm} addiert und mit der Funktion sin aus math.h wird der entsprechende Sinuswert ermittelt. Die Skalierung erfolgt durch Multiplikation mit A, da sin() einen Normierten Wert zwischen 0 und 1 zurückgibt.
Um die LEDs und Taster entsprechend nutzen zu können wurde die main.c angepasst.

```
1  #include <ccblkfn.h>
2  #include "isr.h"
3  #include "codeclib.h"
4
5  void main(void)
6  {
7          // initialize AD1836
8          start_AD1836();
9
10         // !! set control register so that PF5 ...
11         // 8 are enabled as input, all LED PF are directed as output
12         // !! and all LED are switched off
13         ssync();
14
15         //Anfang Modifizierter Code
16         *pFIO2_DIR = 0xFFFF;    //Setze alle LED als Ausgaenge.
17         *pFIO0_DIR &= 0xFF0F;   //Setze die Taster als Eingang.
18         *pFIO0_INEN |= 0x00F0;  //Aktiviere den Input Buffer fuer die Taster.
19         //Ende Modifizierter Code
20
21         // loop forever
22         while(1) {
23                 idle(); // go asleep and wake up when external
24                                 //interrupt and ISR have been
25         }
26 }
```

main.c

Da auf die Eingabe der Taster reagiert werden soll, musste die Interrupt Service Routine (ISR) angepasst werden. Dazu wird in einer Switch-Case-Anweisung überprüft welche Frequenz eingestellt wurde.

```
1      if (Freq200 == 1)
2          *pFIO2_FLAG_S = 0x0100;
3      // !! set control register so that LED 5 is on
4      else if (Freq200 == 2)
5          *pFIO2_FLAG_S = 0x0200;
6      // !! set control register so that LED 6 is on
7      else if (Freq200 == 20)
8          *pFIO2_FLAG_S = 0x0400;
9      // !! set control register so that LED 7 is on
10     else
11         *pFIO2_FLAG_C = 0x0700;
12     // !! set control register so that no LED is on
13
14      ssync();
15
16     // copy sine value to dma output buffer
17     iDMATxBuffer[4] = genSinus(AMPLITUDE, Freq200) * LONG_MAX;
```

isr.c

Entsprechend der Aufgabe wurden die LEDs gesetzt sobald die entsprechende Frequenz ausgewählt ist. Dies wird mit der Programmable Flag (PF)_S realisiert. _S steht in diesem Fall für set. Ist keine der im Switch-Case angegebenen Frequenzen ausgewählt, so wird die PF mit dem Prefix _C zurückgesetzt, _C steht für clear.
Im Array iDMATxBuffer steht am Index 4 der 32bit Ausgabewert des DAC, da die Funktion genSin() nur Werte zwischen -1 und 1 zurück gibt wird mit dem maximalen Wert des 32-bit Formats multipliziert(definiert als LONG_MAX). Dies ermöglicht die Ausnutzung des gesamten Spannungsbereiches des DAC.
Das Board wurde in Betrieb genommen und für die Frequenzen 200Hz, 400Hz und 4kHz wurden sowohl Spannungs-Zeit Verläufe als auch Spektren mit Hilfe des Virtual Instrument (VI) aufgenommen. Auf Grund von Unachtsamkeit ist das Bild des Spektrums des 200Hz Signals verloren gegangen und kann hier daher nicht gezeigt werden.
Wir rekonstruieren aus unseren Unterlagen, dass es wie erwartet aussah und auf kein besonderes Verhalten hinwies.

3.2 Auswertung

Für eine Vergleichbarkeit der Signale haben wir alle Bilder der Signale (Abbildung 3.1 bis 3.3) der gleichen Auflösung für Amplitude und demselben Level für den Trigger erzeugt. Die Zeitbasis wurde entsprechend angepasst, sodass vier Perioden zu sehen sind.

Die Resultate entsprechen den Erwartungen, wobei die Amplitude von 0,7V anfangs überraschte. Die Amplitude ergibt sich aus dem Hardwareaufbau des EVB, so liegt die maximale Aussteuerung bei 6,17V Spitze-Spitze des Codec, was zu einer maximalen Amplitude von 3,08V führt. Softwareseitig haben wir die Amplitude so eingestellt, dass sie das Signal genau auf die Hälfte skaliert, dies erzeugt eine Amplitude von ca. 1,5V. Allerdings lassen sich nur 0,7V messen, da die Ausgangsstufe des EVB einen Spannungsabfall hat.

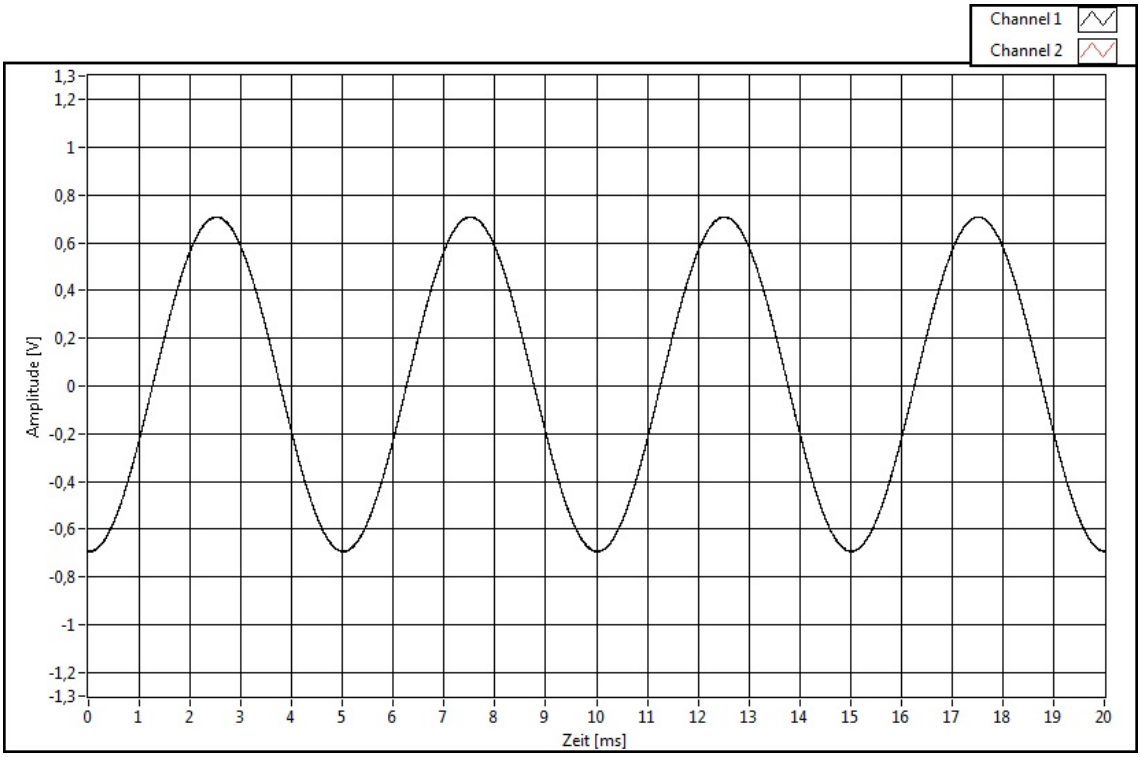

Abbildung 3.1: Spannungs-Zeit Verlauf 200Hz

Die Spektren sehen genau wie erwartet aus und zeigen bei der entsprechend eingestellten Frequenz einen entsprechenden Peak. Einige kleine Peaks sind bei anderen Frequenzen zu ermitteln, ihre Höhe ist aber im Verhältnis so gering, dass sie vernachlässigt werden können.

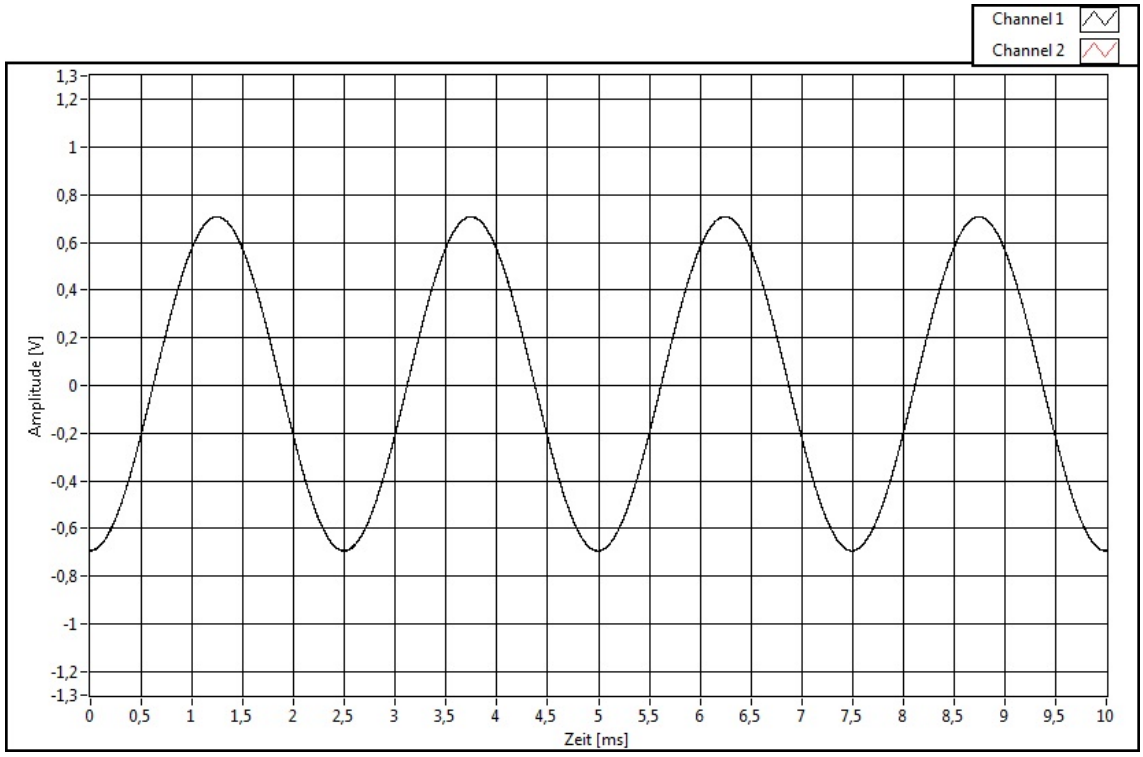

Abbildung 3.2: Spannungs-Zeit Verlauf 400Hz

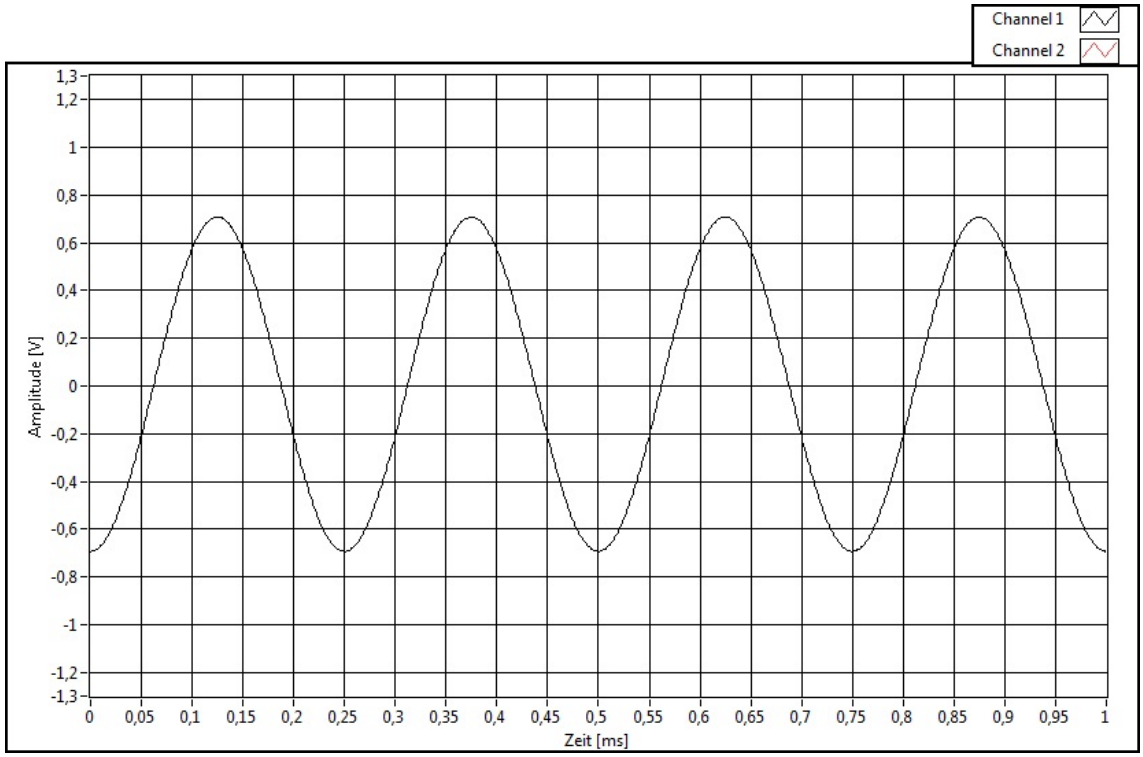

Abbildung 3.3: Spannungs-Zeit Verlauf 4kHz

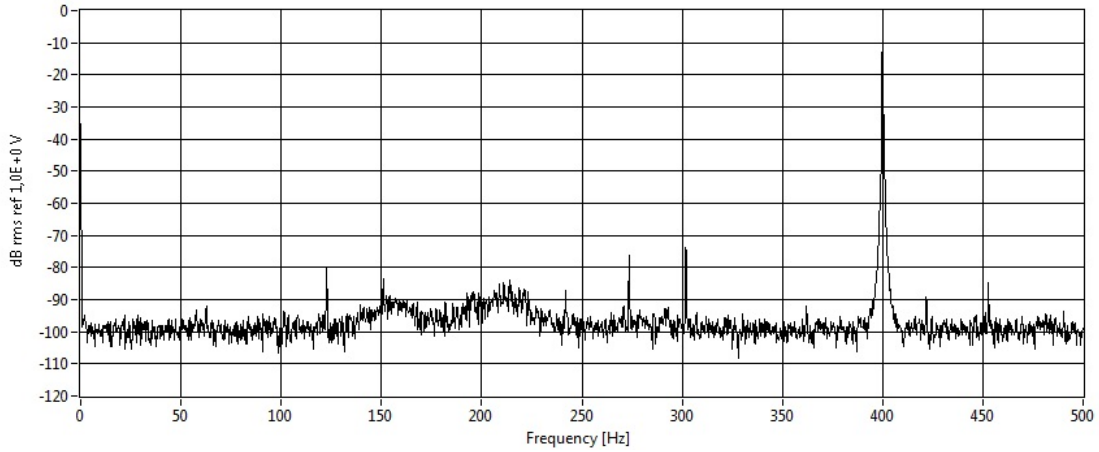

Abbildung 3.4: Amplitude(dB)-Frequenz Verlauf 400Hz

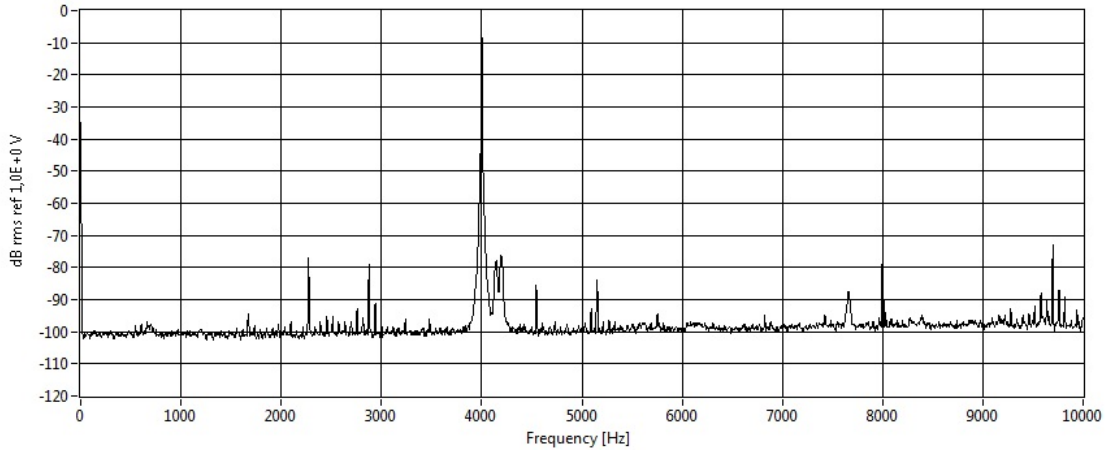

Abbildung 3.5: Amplitude(dB)-Frequenz Verlauf 4kHz

Kapitel 4

Verarbeiten von Signalen

4.1 Durchführung

Die Aufgabe besteht nun darin, die Signale zu verarbeiten, was hier durch einfaches Verstärken oder Dämpfen realisiert wird. In einem ersten Schritt wird das eingelesene Signal unverändert über den Codec zurückgegeben. Dies kann mittels VI sichtbar gemacht werden. Dazu wurde der Funktionsgenerator auf $V_{ss} = 1V$ und 500Hz eingestellt, natürlich ergeben sich die üblichen Abweichungen von der tatsächlichen Messung.

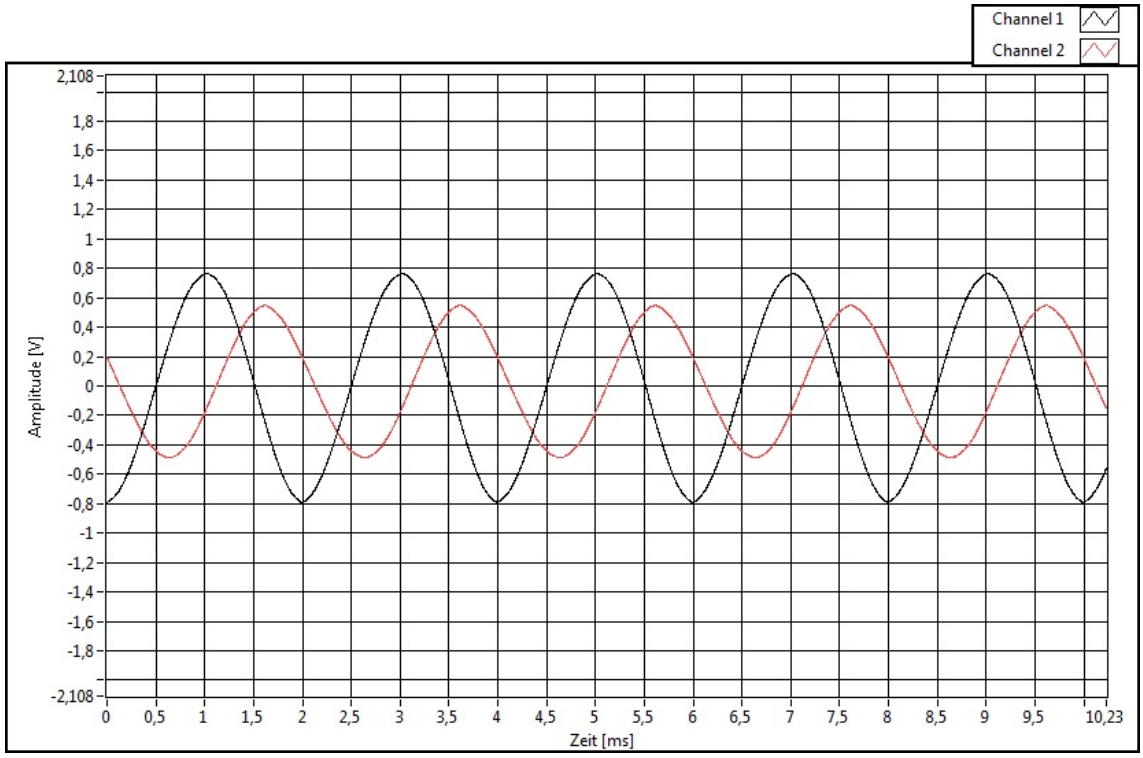

Abbildung 4.1: Vergleich Eingang und Ausgang in C - 500Hz bei 1V

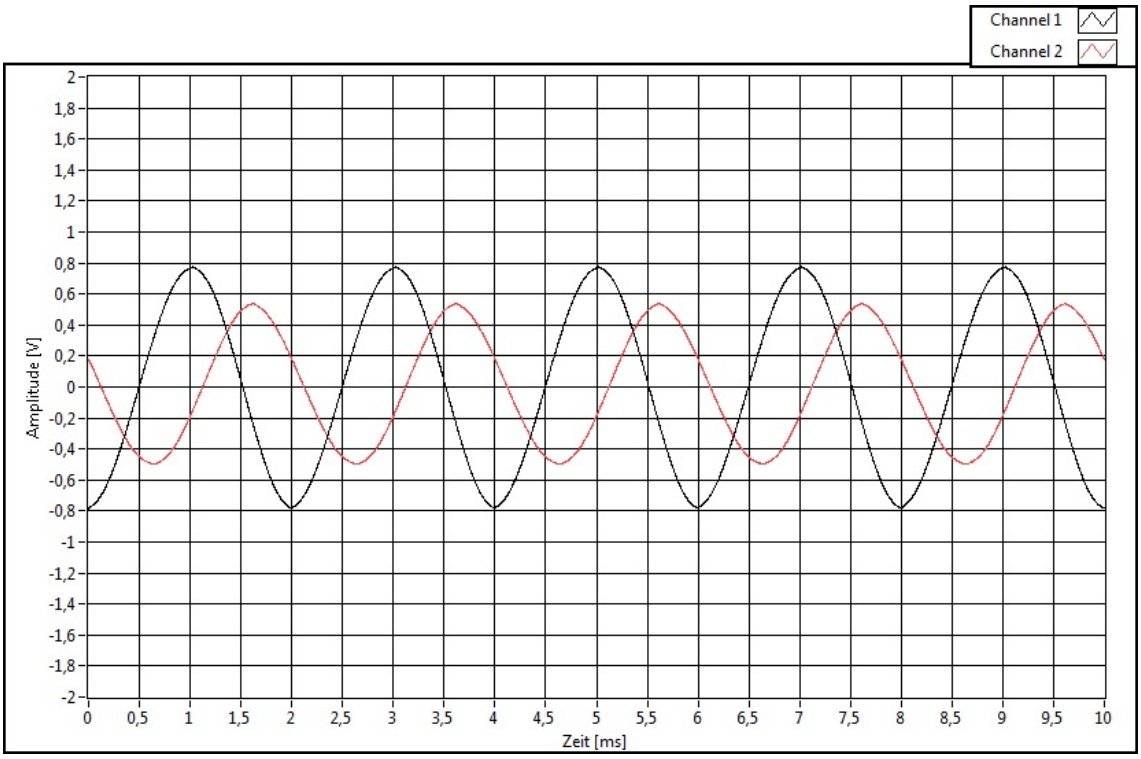

Abbildung 4.2: Vergleich Eingang und Ausgang in Assembler - 500Hz bei 1V

Auch wurden die Frequenzgänge und Phasengänge mittels VI visualisiert.

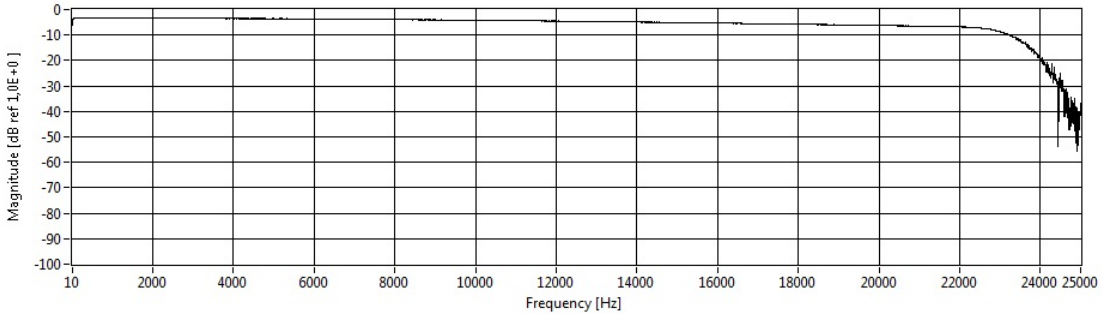

Abbildung 4.3: Frequenzgang Gesamtsystem in C

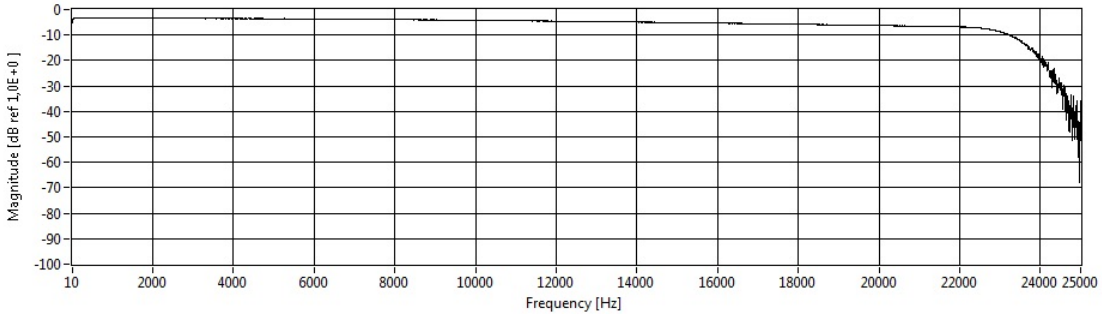

Abbildung 4.4: Frequenzgang Gesamtsystem in Assembler

In Abbildung 4.1 und Abbildung 4.2 ist kein Unterschied sichtbar. Dies entsprach den Erwartungen.

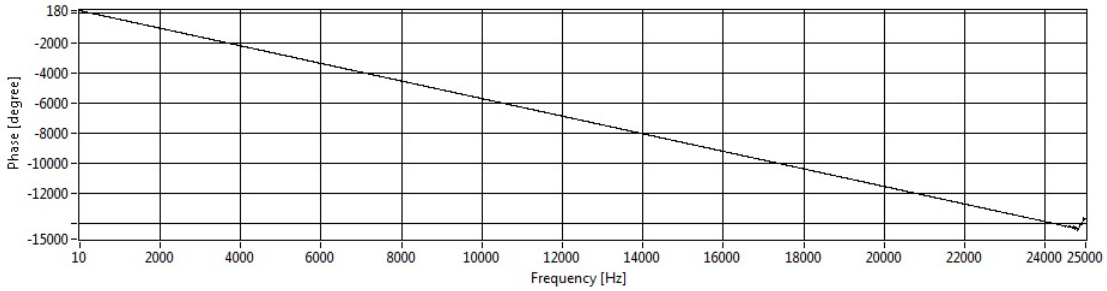

Abbildung 4.5: Phasengang Gesamtsystem in C

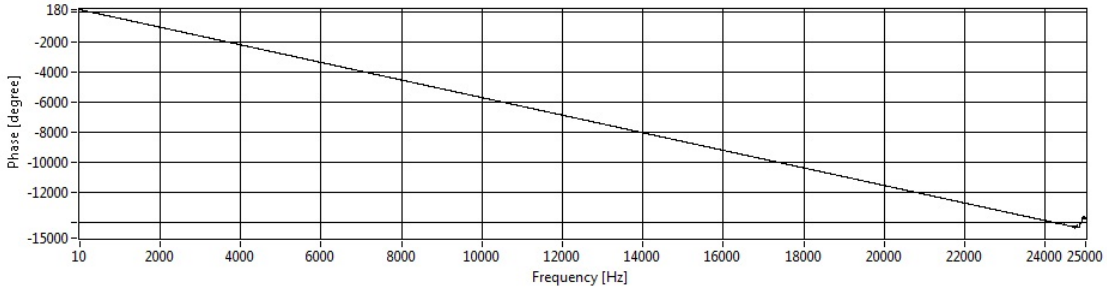

Abbildung 4.6: Phasengang Gesamtsystem in Assembler

Wir haben die Umwandlung der ADC-Werte vom Hexadezimalen in das Signed Integer Format überprüft indem wir im Debugmodus das Programm anhielten und uns die Werte notierten.

iADC2R	
Hexadezimal	Dezimal
00d2d300	13.881.856,00
0038d700	14.104.576,00
0047d600	14.042.880,00
00e4d300	13.886.464,00

Es ist zu erkennen, dass das niederwertige Byte stets mit 00h gefüllt ist, dies begründet sich darin, dass die 24-bit des Codec stets in die höherwertigen Stellen geschrieben werden. Im nächsten Schritt wurde das Programm in Assembler auf 16-bit umgestellt, da dies der optimierte Arbeitsbereich des DSP ist. Hierzu wurde die isr.asm zur isr_s.asm mit folgender Änderung entsprechend der Aufgabenstellung:

```
1          P1.L = _iDMARxBuffer;
2          P1.H = _iDMARxBuffer;
3
4          R1 = [P1+INTERNAL_ADC_L1*4];
5          P2.L = _sADC2L; P2.H = _sADC2L;
6          W[P2] = R1.H;//W hinzugefügt um nur die oberen 16-bit zu wählen.
7
8          R2 = [P1+INTERNAL_ADC_R1*4];
9          P2.L = _sADC2R; P2.H = _sADC2R;
10         W[P2] = R2.H;//W hinzugefügt um nur die oberen 16-bit zu wählen.
11
12         P1.L = _iDMATxBuffer;
13         P1.H = _iDMATxBuffer;
14
```

```
15          P2.L = _sDAC1L; P2.H = _sDAC1L;
16          R1.H = W[P2];//W hinzugefügt um nur die oberen 16-bit zu wählen.
17          [P1+INTERNAL_DAC_L0*4] = R1;
18
19          P2.L = _sDAC1R; P2.H = _sDAC1R;
20          R1.H = W[P2];//W hinzugefügt um nur die oberen 16-bit zu wählen.
21          [P1+INTERNAL_DAC_R0*4] = R1;
```

isr_s.asm

Ein entsprechender Test zeigt, dass keine Veränderung sichtbar wurde.

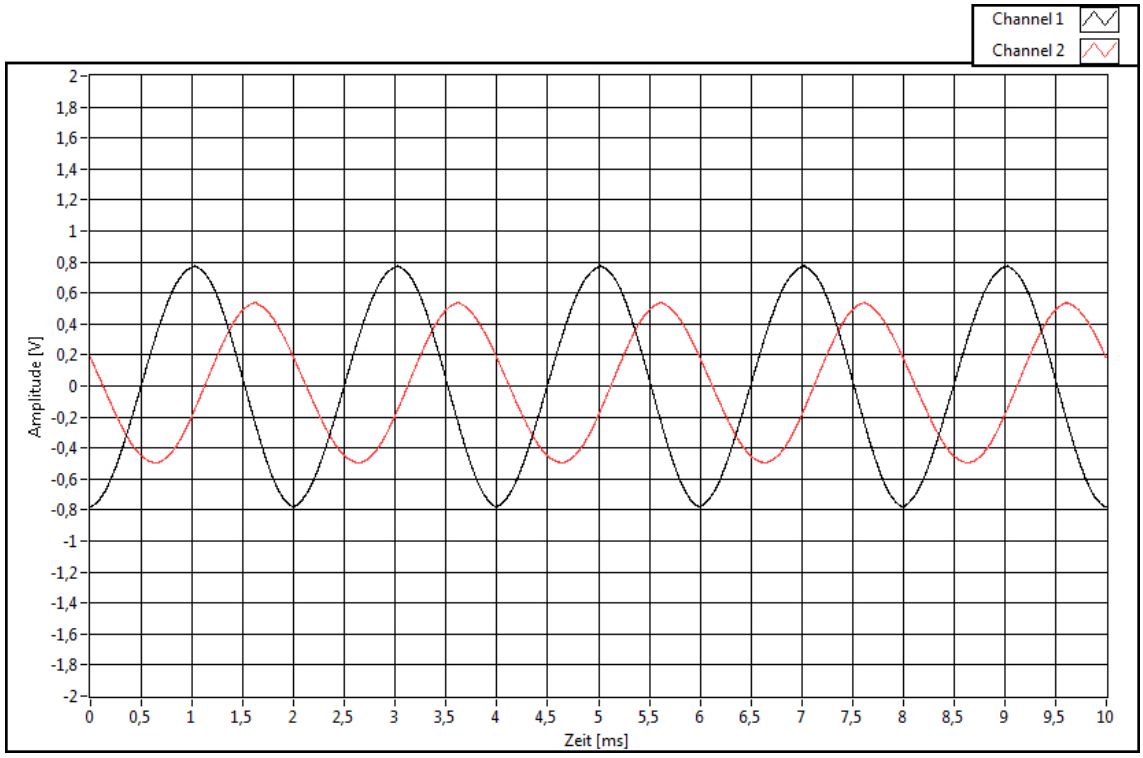

Abbildung 4.7: Vergleich Eingang und Ausgang in C und 16-bit - 500Hz bei 1V

Der letzte Abschnitt der Aufgabe beinhaltete eine Amplitudenveränderung des Signals. Dies wurde in process_data_s.asm bearbeitet.

```
1  .section/DOUBLE32 data1;
2  .extern _sDAC1L;
3  .extern _sDAC1R;
4  .extern _sADC2L;
5  .extern _sADC2R;
6
7  .section/DOUBLE32 program;
8  .global _process_data;
9
10 _process_data:
11
12                 P0.L = _sADC2L; P0.H = _sADC2L;
13                 R0.L = W[P0];
14                 P1.L = _sDAC1L; P1.H = _sDAC1L;
15                 W[P1] = R0.L;
16
17                 P2.L = _sADC2R; P2.H = _sADC2R;
18                 R0.L = W[P2];
19
20 // !! Add the processing R0.L -> R1.L here                */
21
22                 //Multiplikation mit 4
23                 //R1.L = R1.L << 2;
24
25                 //Multiplikation mit 4 und Saettigung
26                 //R1.L = R0.L << 2(S);
27
28                 //Logische Division
29                 //R1.L = R0.L >> 2;
30
31                 //Arithemtische Division
32                 R1.L = R0.L >>> 2;
33
34
35
36                 P3.L = _sDAC1R; P3.H = _sDAC1R;
37                 W[P3] = R1.L;
38
39                 RTS;
40
41 ._process_data.end:
```

processdatas.asm

Für die einzelnen Tests wurden die entsprechenden Zeilen die nicht genutzt wurden kommentiert. So ist in diesem Beispiel die Variante der Arithmetischen Division durch vier zu sehen.

4.2 Auswertung

In den folgenden Bildern ist das Originalsignal in schwarz und das EVB-Ausgangssignal mit Verzögerung und Amplitudenveränderung zu erkennen.

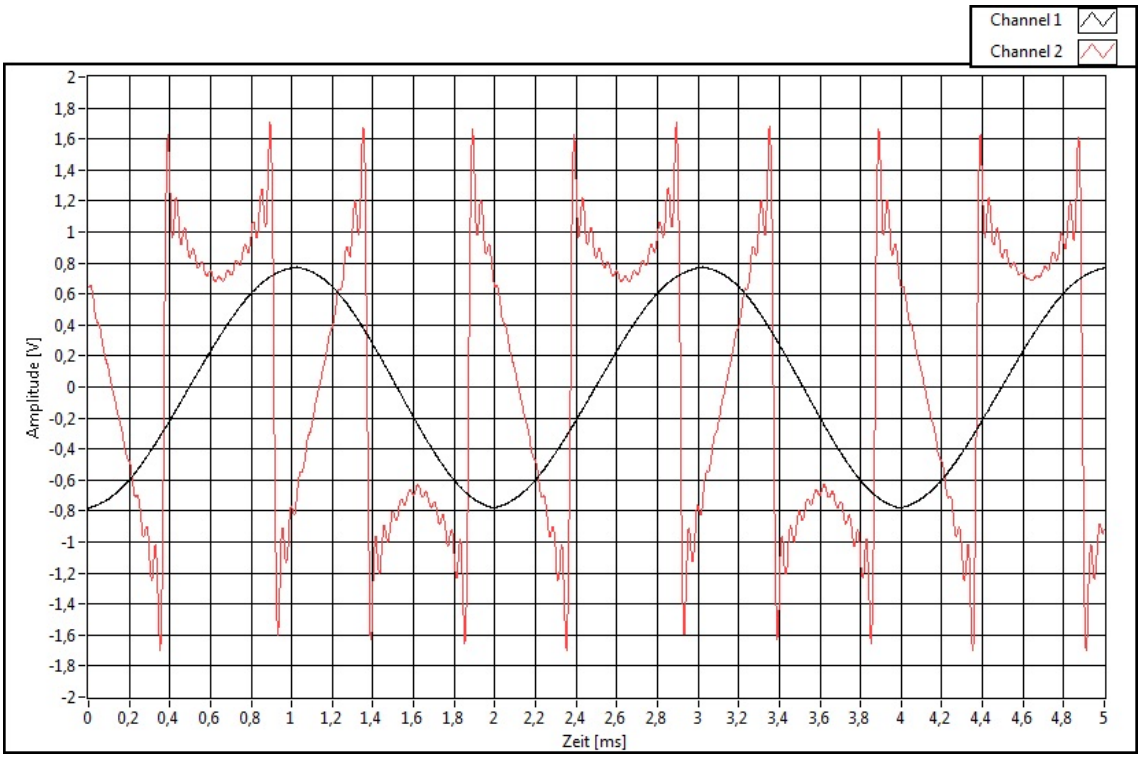

Abbildung 4.8: Multiplikation mit 4 ohne Sättigung

In Abbildung 4.8 ist ein Überlaufverhalten zu erkennen, welches zu entsprechenden Fehlern führt. Außerdem ist dem Signal ein Rauschen überlagert.

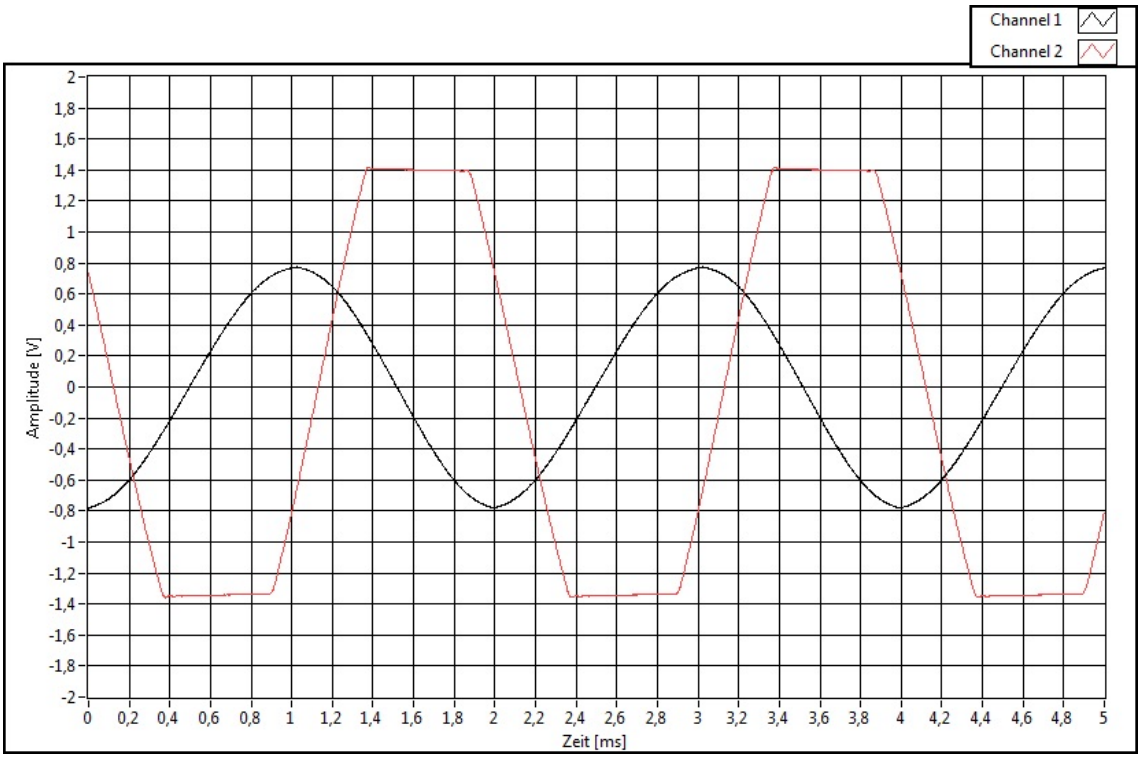

Abbildung 4.9: Multiplikation mit 4 mit Sättigung

Im Unterschied zur Abbildung 4.8 ist in Abbildung 4.9 zu sehen, dass es zu keinen reinen Überläufen kommt, sondern der maximale Aussteuerungsbereich verwendet wird. Dies

führt zu geringeren Fehlern, allerdings ist der originale Signalverlauf nicht rekonstruierbar, da die Maxima nicht mehr sichtbar sind. Der logische Shift sorgt für den Verlust des Vorzeichens, was zu rein positiven Werten führt, da immer eine 0 nachgeschoben wird. Es sind nicht nur positive Werte zu sehen, da dieser Gleichanteil durch die Gleichspanungsentkopplung des Codes gefiltert wird. Die Verringerung der Amplitude ist trotzdem zu sehen. Der arithmetische Shift verändert das Vorzeichen des Signals nicht und ermöglicht es den Signalverlauf beizubehalten. Lediglich die Änderung der Amplitude macht sich bemerkbar.

Der arithmetische Shift behält stets sein Vorzeichen und ermöglicht daher das Signal beizubehalten und lediglich die Änderung der Amplitude macht sich bemerkbar.

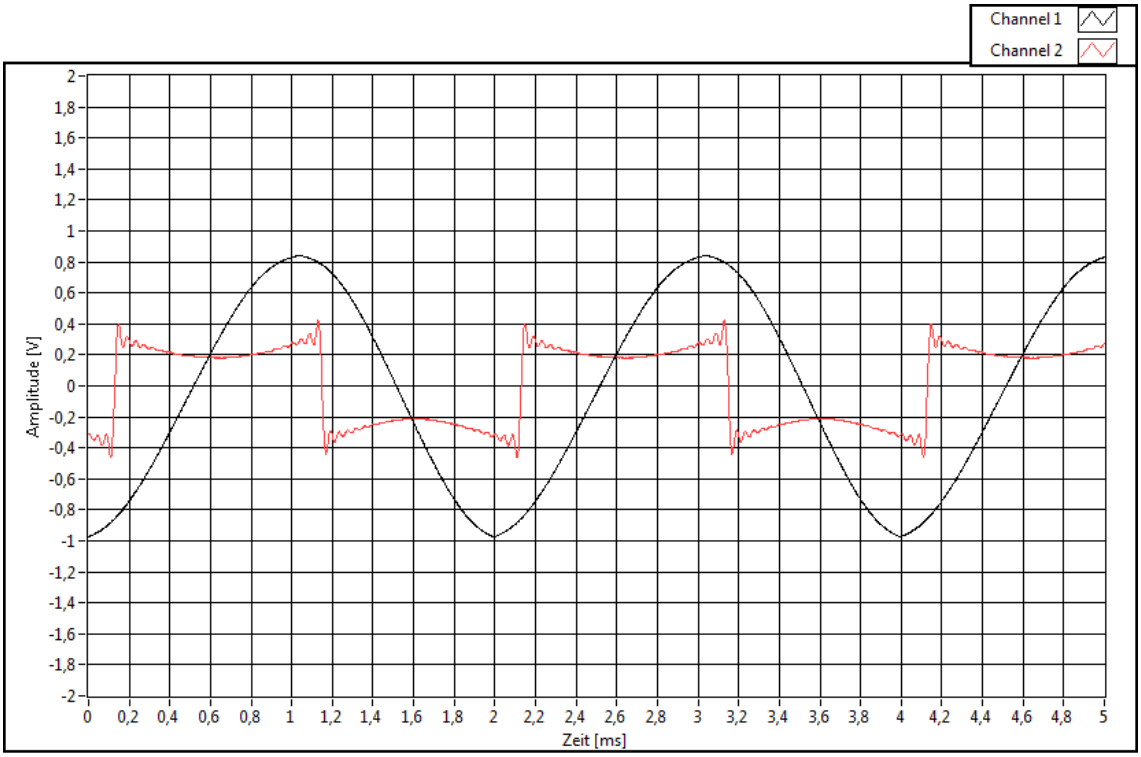

Abbildung 4.10: Logische Division mit 4

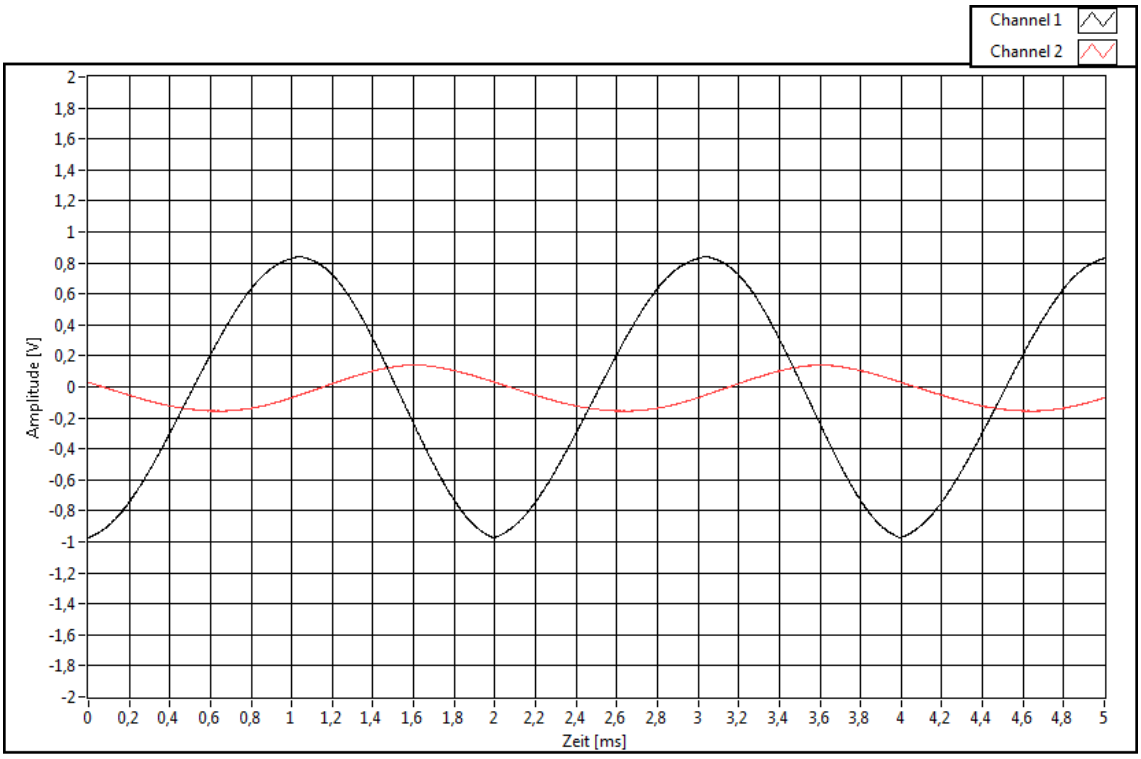

Abbildung 4.11: Arithmetische Division mit 4

Die Signallaufzeit lässt sich mit Hilfe des Phasenganges Graphisch ermitteln. In Abbildung 4.6 und Abbildung 4.5 ist eine Phasenverschiebung -8000° bei 14kHz zu erkennen. Dies entspricht etwa 22 Perioden.

Gemäß $\frac{22}{14kHz} = 1,6ms$ ergibt sich eine Laufzeit von 1,6ms. Diese Zeit resultiert zum einen aus der Verarbeitung durch die Software, aber auch durch die Wandlungen des Codec, wobei die Wandlungen den größten Anteil haben dürften. Ein weiterer Teil der Verzögerung wird dadurch verursacht, dass zwischen Ermitteln der Werte des ADC und Ablegen dieser im Speicher ein Taktzyklus liegt.

Anhang A

Quelltext-Dateien

```h
extern void copyData(int *input, int **pWrite, int Size);
```

copydata.h

```c
#include "codeclib.h"
#include "copydata.h"


/*  @function    copyData
*   @brief       Kopiert die Audiodaten des Kanals "Internal ADC R0"
*                aus dem DMA-Lesepuffer
*                in den Speicherbereich iInput schreibt.
*   @param       input   Adresse der ersten Speicherstelle von input
*   @param       pWrite  Zeiger auf die Adresse von input
*   @param       size    Anzahl der übergebenen Werte
*   @return      void
*/
void copyData(int *input, int **pWrite, int size) {

    //Nimm den 5ten Wert aus iDMARxBuffer und schreibe diesen in den input.
    **pWrite = iDMARxBuffer[4];

    //Inkrementiere den Wert der zuletzt beschriebenen Adresse
    (*pWrite)++;

    // Wenn die obere Grenze size erreicht wurde müssen wir auf
    // die Startadresse zurückspringen.
    if(*pWrite == input + size)
    {
        *pWrite = input;
    }

}
```

copydata.c

```h
float genSinus(float, short);
```

gensinus.h

```c
#include <stdio.h>
#include <math.h>

#define PI 3.141592653 //Definition der Zahl Pi
```

```
5  #define FABTAST 48000  //Definition der Abtastrate des Codec
6
7  /*  @function genSinus
8  *   @brief  Diese Funktion generiert fortlaufend einen Sinussignal.
9  *
10 *   @param  A ist die Amplitude des gewünschten Sinussignal zwischen 0 und 1.
11 *   @param  Freq200 ist die gewünschte Frequenz des Sinussignal
12 *   in Schritten von 200Hz.
13 *   @return Ist der aktuell berechnete Wert des Sinussignals
14 */
15 float genSinus(float A, short Freq200)
16 {
17     float sinusValue;
18     float omegaNormNeu;
19     static float omegaNorm = 0;
20
21     //Die Stellung des Zeigers wird neu berechnet.
22     omegaNormNeu = 2 * PI * ((Freq200 * 200.)/FABTAST);
23     omegaNorm += omegaNormNeu;
24
25     //Ermittlung des Sinuswertes anhand des neuen Zeigers.
26     sinusValue = A * sin(omegaNorm);
27
28     //Bei durchschreiten einer Periode von omegaNorm wird das Signal
29     //um 2Pi zurückgesetzt
30     if(omegaNorm > 2 * PI)
31     {
32         omegaNorm -= 2 * PI;
33     }
34
35     return sinusValue;
36 }
```

gensinus.c

```
1  #include <sys\exception.h>
2  #include <cdefBF561.h>
3
4  //extern int iDAC1L,iDAC1R,iADC2L,iADC2R;
5  extern short sDAC1L,sDAC1R,sADC2L,sADC2R;
6  extern char cNewSample;
7
8  EX_INTERRUPT_HANDLER(Sport0_RX_ISR);
```

isr_s.h

```
1  #include <ccblkfn.h>
2  #include <sys\exception.h>
3  #include <cdefBF561.h>
4  #include "codeclib.h"
5  #include "gensinus.h"
6  #include "limits.h"
7
8  #define INTERNAL_DAC_R0 0x4
9
10 #define AMPLITUDE 0.5
11 #define FREQ_MIN 0      // 0 Hz
12 #define FREQ_MAX 50     // 10 kHz
13
```

```
14 short Freq200=1;         // 200 Hz
15
16 #define SW6_BIT 0x0020
17 #define SW7_BIT 0x0040
18
19 typedef enum {OFF, ON} Switch_States;
20 Switch_States state_sw6=OFF, state_sw7=OFF;
21                                 // state variables for switch 6 and 7
22
23 EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
24 {
25     // confirm interrupt handling
26     *pDMA2_0_IRQ_STATUS = 0x0001;
27
28     switch (state_sw6) {                    // State of SW6
29         case OFF:                           // is off
30         if (*pFIO0_FLAG_D & SW6_BIT) {      // current state = on
31             state_sw6 = ON;                 // set State of SW6 to on
32             if (Freq200 < FREQ_MAX)
33                 Freq200++;                  // Increase Frequency
34         }
35         break;
36         case ON:                            // is on
37         if (!(*pFIO0_FLAG_D & SW6_BIT)) {   // current state = off
38             state_sw6 = OFF;                // set State of SW6 to off
39         }
40         break;
41     }
42
43     switch (state_sw7) {                    // State of SW7
44         case OFF:                           // is off
45         if (*pFIO0_FLAG_D & SW7_BIT) {      // current state = on
46             state_sw7 = ON;                 // set State of SW7 to on
47             if (Freq200 > FREQ_MIN)
48                 Freq200--;                  // Decrease Frequency
49         }
50         break;
51         case ON:                            // is on
52         if (!(*pFIO0_FLAG_D & SW7_BIT)) {   // current state = off
53             state_sw7 = OFF;                // set State of SW7 to off
54         }
55         break;
56     }
57
58     if (Freq200 == 1)
59         *pFIO2_FLAG_S = 0x0100;
60     // !! set control register so that LED 5 is on
61     else if (Freq200 == 2)
62         *pFIO2_FLAG_S = 0x0200;
63     // !! set control register so that LED 6 is on
64     else if (Freq200 == 20)
65         *pFIO2_FLAG_S = 0x0400;
66     // !! set control register so that LED 7 is on
67     else
68         *pFIO2_FLAG_C = 0x0700;
69         // !! set control register so that no LED is on
70     ssync();
71
```

```
72      // copy sine value to dma output buffer
73      iDMATxBuffer[4] = genSinus(AMPLITUDE, Freq200) * LONG_MAX;
74  }
```

isr2.c

```
1   #include <defBF561.h>
2
3   #define INTERNAL_ADC_L1 0x1
4   #define INTERNAL_ADC_R1 0x5
5
6   #define INTERNAL_DAC_L0 0x0
7   #define INTERNAL_DAC_R0 0x4
8
9   .section/DOUBLE32 data1;
10  .align 1;
11  .byte _cNewSample[1];
12  .global _cNewSample;
13
14  .align 4;
15  .byte _sDAC1L[4];
16  .global _sDAC1L;
17  .byte _sDAC1R[4];
18  .global _sDAC1R;
19  .byte _sADC2L[4];
20  .global _sADC2L;
21  .byte _sADC2R[4];
22  .global _sADC2R;
23
24  .extern _iDMATxBuffer;
25  .extern _iDMARxBuffer;
26
27  .section/DOUBLE32 program;
28  .global _Sport0_RX_ISR;
29
30  _Sport0_RX_ISR:
31
32  // push DSP status and regs on stack
33                  [--SP] = ASTAT;
34                  [--SP] = (R7:0,P5:0);
35                  [--SP] = FP;
36                  [--SP] = LC0;
37                  [--SP] = LC1;
38
39                  P0.L =  LO(DMA2_0_IRQ_STATUS); P0.H =  HI(DMA2_0_IRQ_STATUS);
40                  R0 =   1;
41                  W[P0] = R0;
42
43
44                  P1.L = _iDMARxBuffer;
45                  P1.H = _iDMARxBuffer;
46
47                  R1 = [P1+INTERNAL_ADC_L1*4];
48                  P2.L = _sADC2L; P2.H = _sADC2L;
49                  W[P2] = R1.H;
50
51                  R2 = [P1+INTERNAL_ADC_R1*4];
52                  P2.L = _sADC2R; P2.H = _sADC2R;
```

```
53                  W[P2] = R2.H;
54
55                  P1.L = _iDMATxBuffer;
56                  P1.H = _iDMATxBuffer;
57
58                  P2.L = _sDAC1L; P2.H = _sDAC1L;
59                  R1.H = W[P2];
60                  [P1+INTERNAL_DAC_L0*4] = R1;
61
62                  P2.L = _sDAC1R; P2.H = _sDAC1R;
63                  R1.H = W[P2];
64                  [P1+INTERNAL_DAC_R0*4] = R1;
65
66                  R0 =   1;
67                  P0.L = _cNewSample; P0.H = _cNewSample;
68                  B[P0] = R0;
69
70  // pop DSP status and regs from stack
71
72                  LC1 = [SP++];
73                  LC0 = [SP++];
74                  FP = [SP++];
75                  (R7:0,P5:0) = [SP++];
76                  ASTAT = [SP++];
77
78                  RTI;
79
80  ._Sport0_RX_ISR.end:
```

isr_s.asm

```
1   .section/DOUBLE32 data1;
2   .extern _sDAC1L;
3   .extern _sDAC1R;
4   .extern _sADC2L;
5   .extern _sADC2R;
6
7   .section/DOUBLE32 program;
8   .global _process_data;
9
10  _process_data:
11
12                  P0.L = _sADC2L; P0.H = _sADC2L;
13                  R0.L = W[P0];
14                  P1.L = _sDAC1L; P1.H = _sDAC1L;
15                  W[P1] = R0.L;
16
17                  P2.L = _sADC2R; P2.H = _sADC2R;
18                  R0.L = W[P2];
19
20  // !! Add the processing R0.L -> R1.L here                */
21
22                  //Multiplikation mit 4
23                  //R1.L = R1.L << 2;
24
25                  //Multiplikation mit 4 und Saettigung
26                  //R1.L = R0.L << 2(S);
27
```

```
28                  //Logische Division
29                  //R1.L = R0.L >> 2;
30
31                  //Arithemtische Division
32                  R1.L = R0.L >>> 2;
33
34
35
36                  P3.L = _sDAC1R; P3.H = _sDAC1R;
37                  W[P3] = R1.L;
38
39                  RTS;
40
41  ._process_data.end:
```

process_data_s.asm

Abbildungsverzeichnis

Abkürzungsverzeichnis

EVB Evaluationboard. 3, 5, 6, 10, 17

ISR Interrupt Service Routine. 9

PF Programmable Flag. 9

VI Virtual Instrument. 9, 13, 14