# Einleitende Worte

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.

# Einlesen von Signalen

Im Ersten Versuch wird der Umgang mit der Software VisualDSP++ erlernt. Dies geschieht mit einem Projekt, dass die Signale die der CODEC an den DSP liefert visualisiert werden, um sie so mit dem anliegenden Eingangssignal zu vergleichen.

Entsprechend der Aufgabenstellung wurde ein Projekt angelegt, der c-Kode kompiliert und auf das EVB (Evaluationsboard) gespielt. Wir legten entsprechend der Aufgabenstellung ein Signal an den rechten Kanal des ADC1 an.   
Aufgabe1Scope1.jpg

V-t Diagramm des ersten Versuchs

Dieses Signal wurde, wie alle weiteren mit dem VI „FFT & FRF & Scope“ aufgezeichnet.

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

#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;

}

}

Wie man leicht sieht werden die Werte die der CODEC liefert kopiert und in den iDMARxBuffer vortlaufend geschrieben, bis dieser voll ist. In dem Fall wird vorn angefangen und die Werte werden überschrieben.

Wird dieser Kode nun vom DSP ausgeführt ermöglicht die Software VisualDSP++ sie zu debuggen, was die Möglichkeit des Stoppens zu jeder Zeit bietet. Stoppt man nun den Durchlauf kann man, wie in der Aufgabenstellung beschrieben die aktuellen Werte des iDMARxBuffer ausleiten und mit MatLab visualisieren.

Aufgabe1Matlab1.jpg

# Auswertung 1.2

Für das geübte Auge ist in der Abbildung durchaus ein Sinussignal zu erkennen. Die Amplitude beträgt 1446560512 dies wurde via MatLab aus der Datenreihe über die Funktion max() ermittelt. Für die normierte Kreisfrequenz wird der Ansatz verwendet und führt bei 18 Messwerten in 2 Perioden (es folgt N=18/2) zu Omega0=8/9Pi.  
Da die Diskreten Amplitudenwerte im 32bit-Integer Format vorliegen, müssen sie in eine Spannung umgerechnet werden, hierzu sind die Daten des EVB erforderlich. Der Codec hat eine Auflösung von 24bit bei einer Spitze-Spitze-Spannung Vss=6,16V. Es gilt zur Berechnung der Amplitudenspannung Vamp=(Vadc\*Vss)/(2\*2^(Auflösung-1)). Es Ergibt sich für die Amplitude bei 1446560512 Vmax= wert aus matlab eintragen

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 zu f0=5,3kHz berechnen, da der Codec mit 48kHz abtastet. Aus der Messung ergibt sich grafisch eine Amplitude von Vss=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 Verschaltung des EVB, wie der Verstärker zwischen Eingang und ADC.

# Erzeugen von Signalen

In dieser Aufgabe war das Ziel den DSP ein Sinussignal ausgeben zu lassen und die Frequenz dieses Sinusses per Taster in 200Hz schritten zu verändern, wobei 2 LEDs die Frequenzen 200Hz, 400Hz und 4kHz signalisierten.

Zur Realisierung dieser Aufgabe wurde in der Vorbereitung ein Funktion genSinus erstellt, die bei jedem Aufruf den nächsten Wert eines Sinus ausgibt.  
#include <stdio.h>

#include <math.h>

#define PI 3.141592653 //Pi und die Abtastfrequenz werden definiert da sie unveränderlich sind

#define FABTAST 48000

float genSinus(float A, short Freq200) //beim Funktionsaufruf wird die Amplitude und die Frequenz übergeben

{

float sinusValue;

float omegaNormNeu;

static float omegaNorm = 0;

omegaNormNeu = 2 \* PI \* ((Freq200 \* 200.)/FABTAST); //die Zeigerstellung wird berechnet

omegaNorm += omegaNormNeu; // die neue Zeigerstellung wird ermittelt

sinusValue = A \* sin(omegaNorm);//der Sinuswert wird entsprechend der Zeigerstellung ermittelt

if(omegaNorm > 2 \* PI)//um Ueberlauf zu vermeiden wird ab 2Pi der Zeiger um 2Pi zurück gesetzt (eine Periodendauer

{

omegaNorm -= 2 \* PI;

}

return sinusValue; //der Sinuswert wird zurückgegeben

}

Die Funktion ist so Implementiert, dass sie den jeweiligen Zeitpunkten einen entsprechenden Sinuswert zuweist und diesen zurück weißt. Als übergabeparameter sind zum einen die Amplitude und zum anderen die Frequenz vorgesehen wobei die Frequenz in 200Hz Schritten übergeben wird. Dies entspricht der Aufgabe und hält den Übergabeparameter klein da ein Char ausreicht, hat aber die Folge dass die übergeben Zahl mit 200 multipliziert werden muss. Das Zeigerdelta also wie weit der Zeiger im Durchlauf weitergedreht werden muss, wird entsprechend Omega=2Pi\*f ermittelt und normiert. Der Aufruf durch den Timerinterrupt stellt dabei periodizität sicher. Der letzte Zeigerwert wird dann mit dem Zeigerdelta addiert und mit der Funktion sin aus math.h wird der entsprechende Sinuswert ermittelt. Die Skalierung erfolgt durch simple multiplikation mit A, da sin() einen Normierten Wert zurück gibt.

Um die LEDs und Buttons entsprechend zu modifizieren wurde die main.c angepasst indem folgender Kode eingefügt wurde:

\*pFIO2\_DIR = 0xFFFF; //setze alle LED als Ausgänge.

\*pFIO0\_DIR &= 0xFF0F; //setze die Buttons als input

\*pFIO0\_INEN |= 0x00F0; // erlaubt den Interrupt der Taster

Die Modifikation wurde vor der Dauerschleife eingefügt um die Taster als Eingang und die LEDs als Ausgang zu konfigurieren. Außerdem wurde über die programmable Flags der Interrupt der Taster erlaubt, sodass ein Tastendruck die isr aufruft. Des weiteren musste entsprechend die isr angepasst werden.

#define AMPLITUDE 0.5

#define FREQ\_MIN 0 // 0 Hz

#define FREQ\_MAX 50 // 10 kHz

if (Freq200 == 1)

\*pFIO2\_FLAG\_S = 0x0100;

// !! set control register so that LED 5 is on

else if (Freq200 == 2)

\*pFIO2\_FLAG\_S = 0x0200;

// !! set control register so that LED 6 is on

else if (Freq200 == 20)

\*pFIO2\_FLAG\_S = 0x0400;

// !! set control register so that LED 7 is on

else

\*pFIO2\_FLAG\_C = 0x0700;

// !! set control register so that no LED is on

ssync();

// copy sine value to dma output buffer

iDMATxBuffer[4] = genSinus(AMPLITUDE, Freq200) \* LONG\_MAX;//Sinussignal an den DAC geben

}

Entsprechend der Aufgabe wurden die LEDs gesetzt sobald die entsprechende Frequenz angewählt ist und andernfals zurück gesetzt, dies geschieht über die PFs mit \_S für setzen und \_C für rücksetzen (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.

# Durchführung

Das Board wurde in Betrieb genommen und für die Frequenzen 200Hz, 400Hz und 4kHz wurden sowohl Spannungs-Zeit-Verläufe als auch FFTs mit Hilfe des VIs aufgenommen. Auf Grund von Unachtsamkeit ist das Bild der FFT 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.

200Hz.jpg

V-t Diagramm 200Hz

400Hz.jpg

V-t Diagramm 400Hz

4kHz.jpg

V-t Diagramm 4kHz

FFT400Hz.jpg

FFT400Hz

FFT4kHz.jpg

FFT4kHz

# Auswertung

Für eine Vergleichbarkeit der Signale haben wir die Oszilloskopbilder der 3 Signal mit der Selben Amplitudenauflösung und demselben Triggerlevel 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 6,16V Spitze-Spitze des Codec, was zu einer maximalen Amplitude von 3,08V führt. Softwareseitig haben wir A=0,5 eingestellt was eine Amplitude von ca. 1,5V erzeugt, allerdings lassen sich nur 0,7V messen, da die ausgangsstufe des EVB einen Spannungsabfall hat. Die FFts sehen genau wie erwartet auf 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.

# Verarbeiten von signalen

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 1Vpp und 500Hz eingestellt, natürlich ergeben sich die üblichen abweichungen von der Tatsächlichen Messung.

1V500Hz\_ProgrammTest.bmp

1V 500Hz in C

1V500Hz\_AssemblerTest.bmp

1V 500Hz in Assembler

Auch wurden die Frequenzgänge und Phasengänge mittels VI visualisiert

Frequenzgang\_Gesamtsystem.bmp

Frequenzgang des Gesamtsystems in C

Frequenzgang\_Gesamtsystem\_asm.bmp

Frequenzgang des Gesamtsystems in Assembler

Phasengang\_Gesamtsystem.bmp

Phasengang des Gesamtsystems in C

Phasengang\_Gesamtsystem\_asm.bmp

Phasengang des Gesamtsystem in Assembler

Es ist kein deutlicher Unterschied fest zu stellen. Alle Ergebnisse entsprachen den Erwartungen.

Im nächsten Schritt wurde das Programm assemblerseitig auf 16bit umgestellt, da dies der optimierte Arbeitsbereich des DSP ist. Hierzu wurde die isr.asm zur isr\_s.asm mit folgender Änderung entsprechend der Aufgabenstellung:

P1.H = \_iDMARxBuffer;in Zeile 45 zugefügt

P1.H = \_iDMATxBuffer;in Zeile 56 zugefügt

Ein entsprechender Test zeigt, dass keine Veränderung sichtbar wurde.

1V500Hz\_AssemblerTest\_16Bit.bmp

1V bei 500Hz in Assembler bei 16 bit Auflösung

Der letzte Abschnitt der Aufgabe beinhaltete eine Amplitudenveränderung des Signals, dies wurde in process\_data\_s.asm bearbeitet.

\_process\_data:

P0.L = \_sADC2L; P0.H = \_sADC2L;

R0.L = W[P0];

P1.L = \_sDAC1L; P1.H = \_sDAC1L;

W[P1] = R0.L;

P2.L = \_sADC2R; P2.H = \_sADC2R;

R0.L = W[P2];

// !! Add the processing R0.L -> R1.L here \*/

//Multiplikation mit 4

//R1.L = R1.L << 2;

//Multiplikation mit 4 und Sättigung

//R1.L = R0.L << 2(S);

//Logische Division

//R1.L = R0.L >> 2;

//Arithemtische Division

R1.L = R0.L >>> 2;

P3.L = \_sDAC1R; P3.H = \_sDAC1R;

W[P3] = R1.L;

RTS;

.\_process\_data.end:

Für die einzelnen Tests wurden die entsprechenden Zeilen die nicht gefragt waren auskommentiert, so sieht man in diesem Beispiel die Variante der Arithmetischen Division durch vier.

In den Volgenden Bildern ist das Originalsignal in schwarz und das EVB-Ausgangssignal mit Verzögerung und Amplitudenänderung zu erkennen.  
  
Mul4\_ohneS.bmp

Multiplikation mit Vier ohne Sättigung  
Man sieht eindeutig das Überlaufverhalten des Zahlenbereichs welches zu entsprechenden Fehlern führt. Außerdem ist eine Art Rauschen auf dem Signal.  
  
Mul4\_mitS.bmp

Multiplikation mit 4 mit Sättigung

Im Unterschied zum Vorherigen Bild sieht man, 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.

divlog.png

logische Division durch vier

Das 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 dies einem Gleichstromofset gleich kommt und das EVB eine Gleichspannungsentkopplung am Codec besitzt. Die verringerung der Amplitude ist trotzdem zu sehen.

divarith.png

Arithmetische Division durch Vier

Das arithmetische shift behält stets sein Vorzeichen und ermöglicht daher die Signal beizubehalten und lediglich die Änderung der Amplitude macht sich bemerkbar.

# Auswertung

Die Signallaufzeit lässt sich mit Hilfe des Phasenganges Graphisch ermitteln. Man sieht bei 14kHz -8000° Phasenverschiebung, das sind etwa 22 Perioden. Gemäß 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.

Die andern kids haben iwie denADC noch ausgewertet vgl behrens-kasper-otto und duemontschlentherstein ich sehenaber laut seiner aufgabe keinen grund dazu

Bitte alle code dateien die wir bearbeitet ahben anhängen

Bitte auf inhalt orthographie genderneutralität grammatik politischerNeutralität etc prüfen und korrigieren

Das hier liest eh niemand Pascal Aragon Kahlert Robärt Kozok