institute of telecommunications

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Digitale Signalverarbeitung (LVA 389.066, 389.067)

Musterprogramme für den ADSP-21369

© M. Heinschink, M. Hofbauer, L. Osl, G. Doblinger

März 2013

Gerhard.Doblinger@tuwien.ac.at

Inhaltsverzeichnis

1 Kurzbeschreibung

Diese Unterlage umfasst die Beschreibung der Musterprogramme für das *ADSP-21369 EZ-KIT Lite Evaluation Kit* [1], die von den Studenten M. Heinschink, M. Hofbauer und L. Osl im Rahmen des Seminars „Digitale Signalverarbeitung“ erstellt wurden. Die Adaptionen betreffen insbesondere die Überarbeitung der von *Analog Devices Inc.* [2] zur Verfügung gestellten *Code Examples*, und zwar einerseits einem Block-Based Talk-Thru, welches in C geschrieben ist, und zweitens einem Single Sample Talk-Thru in Assembler. Diese Programme wurden soweit modifiziert, dass sie als Grundgerüst für viele weitere Anwendungen wie Filter, FFT und Ähnliches fungieren können. Besonderes Augenmerk wurde auf die leichte Veränderung der Abtastrate und die korrekte Umsetzung der ADC/DAC Daten gelegt, da die ursprünglichen Talk-Thru-Programme nur eine sehr schlichte Verarbeitungsmöglichkeit bieten.

Es wurden schlussendlich vier verschiedene Programmcodes erstellt, nämlich jeweils in C und Assembler ein Single sample und ein Block Sample Talk-Thru. Alle Programme lesen einen Wert bzw. einen Block von Werten vom ADC des Evaluation Kits ein, konvertieren die Werte in das float-Format und stellen diese zur Manipulation zur Verfügung. Anschließend werden die daraus neu errechneten Werte an den DACs des Boards ausgegeben. Für das Einlesen stehen beide Kanäle eines Stereo-Audio-Einganges zur Verfügung, für die Ausgabe enthält das Evaluation Kit vier DACs, welche jeweils einen Stereo-Audio-Ausgang ansteuern. Auch die Abtastrate von ADC und DAC können im Programm verändert werden.

Alle Programme wurden dem Musterprogramm nachempfunden, welches in den Seminarunterlagen [3] für den Prozessor *ADSP-21065L* vorgestellt wird. Daher wurden Programmteile dieses Musterprogramms entsprechend modifiziert verwendet. Im Folgenden werden jeweils getrennt für die einzelnen Programme die Codes und insbesondere die vorgenommenen Änderungen und Einstellmöglichkeiten im Programm behandelt.

Neu ab 2013 ist das Musterprogramm für die FFT-Filterbank von G. Doblinger, das die Lösung einer Vielzahl von Audioanwendungen vereinfacht, da nur mehr die spektralen Modifikationen zu programmieren sind. Der ADSP-21369 ist wesentlich schneller als der ADSP-21065L, so dass dieses Musterprogramm in C geschrieben und für die Verarbeitung von Stereosignalen ausgelegt ist.

2 Programme

Alle hier vorgestellten Programme wurden ursprünglich mit Hilfe der Entwicklungsumgebung *VisualDSP++* in der Version 4.5.5 von *Analog Devices Inc.*[2] erstellt. Dabei kam es immer wieder zu Problemen bei der Verwendung des Debuggers. Gesetzte Breakpoints konnten nicht mehr gelöscht werden oder der geschriebene Code wurde nicht mehr korrekt ausgeführt. In diesen Fällen konnte oft nur mehr ein Neustart der Entwicklungsumgebung Abhilfe leisten. Im Seminar wird die Version 5.x verwendet, bei der die Verwendung von Breakpoints ebenfalls problematisch sein kann. Bei hartnäckigen Fehlern sollte daher versucht werden, das betreffende Code-Segment zuerst im Simulator zu testen.

2.1 Assembler - single sample

Dieses Programm beruht auf dem *Code Example* von *Analog Devices Inc.* [2]. Das in Assembler geschriebene Programm wurde soweit ergänzt, dass es als vielseitige Grundlage für alle möglichen Anwendungen der Signalverarbeitung dienen kann.

Das *Code Example* SINGLE SAMPLE TALK-THRU liest einen Wert von dem am Signalprozessor *ADSP-21369* über SPI angehängten *AD1835A* [4] (2fach ADC, 8fach DAC, 96kHz, 24 Bit, mit integriertem $\Sigma - \Delta - Codec$) ein und gibt anschließend wieder einen Wert auf dem *AD1835A* aus. Dabei wird jeweils abwechselnd ein Wert vom ersten Analog-Digital-Konverter und anschließend vom zweiten ADC eingelesen, sodass im Endeffekt ein linker (weiße Buchse) und ein rechter (rote Buchse) Audio-Kanal mit einer maximalen Abtastrate von je 96kHz zur Verfügung stehen. Dasselbe gilt auch für die Digital-Analog-Verstärker, welche in vier Paaren mit je zwei Kanälen organisiert sind.

Das *Code Example* wurde insoweit ergänzt, als dass die beiden Kanäle getrennt als linker und rechter Kanal zu *einem* Zeitpunkt für die Verarbeitung zur Verfügung stehen. Dies hat den Vorteil, dass eine übersichtlichere und einfachere Verarbeitung der beiden Datenströme erreicht wird, jedoch mit dem Preis, dass für die Berechnung der Ausgangswerte nur die halbe Rechenzeit der DSPs zur Verfügung steht. Dies ist auf die Tatsache zurückzuführen, dass das gesamte Programm komplett INTERRUPT-gesteuert ist, weshalb die Daten der beiden Kanäle ebenfalls direkt im Interrupt der seriellen Schnittstelle, an der der ADC-DAC *AD1835A* angeschlossen ist, verarbeitet werden. Da diese Verarbeitung immer nur nach dem Empfang beider Kanäle erfolgen kann, jedoch abgeschlossen werden muss, bevor der nächste Wert des ersten ADC-Kanals entgegengenommen wird, reduziert sich die effektive Rechenzeit für die Manipulation der Daten auf die Hälfte.

Für besonders zeitkritische Anwendungen muss deshalb das hier vorliegende Programm soweit verändert werden, dass die Verarbeitung der Daten getrennt für den linken und den rechten Kanal jeweils sofort nach Erhalt des Eingangswertes erfolgt, soweit das die Anwendung zulässt.

main.asm

```
1  _main:
2
3      call _initSDRAM;       // Initialize SDRAM for the correct SDRAM clock (
           SDCLK) frequency
4      call _initSRU;         // Initialize the SRU & DAI/DPI pins
5      call _initSPORT;       // Initialize the serial ports (SPORTS)
6      call _init1835viaSPI; // Initialize the ADC/DAC
7
8      call Init_variables;  //Initialize global variables
9
10     BIT SET MODE1 IRPTEN; // Enable interrupts (global)
11     BIT SET MODE1 CBUFEN; // Enable circular buffers (global)
12
13     LIRPTL = SP0IMSK;      // Unmask the SPORT0 ISR
14
15 _main.end:
16     jump (pc,0);     // Loop forever.  Work is driven by interrupts
```

Zu Beginn der MAIN-Routine wird das angeschlossene SDRAM mit der entsprechenden Taktrate initialisiert (_INITSDRAM). Diese Routine wurde ohne Änderungen direkt aus dem *Code Example* von *Analog Devices Inc.* übernommen, weshalb hier nicht näher auf sie eingegangen wird. Dasselbe gilt für _INITSRU, in der die *signal routing unit*, das *digital audio interface* und das *digital peripheral interface*, und _INITSPORT, in der die seriellen Schnittstellen des DSPs initialisiert werden. Anschließend werden der *AD1835A* über die SPI-Schnittstelle (_INIT1835VIASPI) sowie einige globale Variablen initialisiert. Weiters werden die notwendigen Interrupts freigeschaltet und die Verwendung von zyklischen Puffern ermöglicht. Zum Schluss der MAIN-Routine geht der DSP in eine Endlosschleife, da die gesamte Verarbeitung der Daten im SPORT0-INTERRUPT erfolgt.

init1835viaSPI.asm

```
1  .var config_tx_buf[]=          // Buffer of configuration data
2              WR | DACCTRL1 | DACI2S | DAC24BIT | DACFS,
3              WR | DACCTRL2,     // e.g.: | DACMUTE_R1 | DACMUTE_L2,
4              WR | DACVOL_L1 | DACVOL_MAX,
5              WR | DACVOL_R1 | DACVOL_MAX,
6              WR | DACVOL_L2 | DACVOL_MAX,
7              WR | DACVOL_R2 | DACVOL_MAX,
8              WR | DACVOL_L3 | DACVOL_MAX,
9              WR | DACVOL_R3 | DACVOL_MAX,
10             WR | DACVOL_L4 | DACVOL_MAX,
11             WR | DACVOL_R4 | DACVOL_MAX,
12             WR | ADCCTRL1 | ADCFS,
13             WR | ADCCTRL2 | ADCI2S | ADC24BIT,
14             WR | ADCCTRL3 | IMCLK | PEAKRDEN;
```

In dieser Routine wird der über die SPI-Schnittstelle an den Prozessor *ADSP21369*[4, 7] angebundene ADC/DAC *AD1835A* initialisiert. Hier wird also festgelegt, dass letzterer im *single sample* Betrieb arbeitet. Außerdem können auch Auflösung und Taktrate des

AD1835A festgelegt werden. Letztere wird über die Variablen DACFS, ADCFS und IMCLK festgelegt [4], womit insgesamt fünf unterschiedliche Abtastraten eingestellt werden können.

labor.h

```
1  #define SAMPLERATE 24   //possible rates: 16, 24, 32, 48, 96kHz
2
3  #if SAMPLERATE == 16
4      #define ADCFS ADCFS48
5      #define DACFS DACFS48
6      #define IMCLK IMCLKx23
7  #endif
8  #if SAMPLERATE == 24
9      #define ADCFS ADCFS48
10     #define DACFS DACFS48
11     #define IMCLK IMCLKx1
12 #endif
13 #if SAMPLERATE == 32
14     #define ADCFS ADCFS96
15     #define DACFS DACFS96
16     #define IMCLK IMCLKx23
17 #endif
18 #if SAMPLERATE == 48
19     #define ADCFS ADCFS48
20     #define DACFS DACFS48
21     #define IMCLK IMCLKx2
22 #endif
23 #if SAMPLERATE == 96
24     #define ADCFS ADCFS96
25     #define DACFS DACFS96
26     #define IMCLK IMCLKx2
27 #endif
```

Um ein einfacheres Einstellen der Taktrate zu ermöglichen, kann diese in der Header-Datei LABOR.H direkt über die Variable SAMPLERATE ausgewählt werden. Außerdem wurde die Header-Datei AD1835.H verändert, da in dieser die Werte der Variablen IMCLK falsch gesetzt waren.

ad1835.h

```
1 #define IMCLKx2     (0x0000)  // Internal MCLK = external MCLK x 2
2 #define IMCLKx1     (0x0040)  // Internal MCLK = external MCLK
3 #define IMCLKx23    (0x0080)  // Internal MCLK = external MCLK x 2/3
```

Hier wird die interne Taktrate des *AD1835A* festgelegt. Die Werte für IMCLKx1 und IMCLKx23 wurden korrigiert.

SPORTisr.asm

```
1  _talkThroughISR:
2
3      r10=dm(Toggle_Channel);
4      r10=-r10;
5      dm(Toggle_Channel)=r10;
6      if GT call Left_channel;
7      if LT call Right_channel;
8
9  _talkThroughISR.end: rti;
10
11
12 // ------------------------------------------------------------------ //
13
14 Left_channel:
15     r10=dm(RXSP0A);      // Read new left sample from ADC
16     dm(Left_Channel_In) = r10;
17
18     r10=dm(R10_Save);    //restore register 10 from previous pass
19     nop; nop; nop; nop;
20     call DSP_program;    //process new samples
21     nop; nop; nop; nop;
22     dm(R10_Save)=r10;    //save register 10 for next pass
23
24     r10 = dm(Left_Channel1_Out);     // Write to DAC1
25     dm(TXSP1A)=r10;
26     r10 = dm(Left_Channel2_Out);     // Write to DAC2
27     dm(TXSP1B)=r10;
28     r10 = dm(Left_Channel3_Out);     // Write to DAC3
29     dm(TXSP2A)=r10;
30     r10 = dm(Left_Channel4_Out);     // Write to DAC4
31     dm(TXSP2B)=r10;
32
33 Left_channel.end: rts;
34
35 Right_channel:
36     r10=dm(RXSP0A);      // Read new right sample from ADC
37     dm(Right_Channel_In)  = r10;
38
39     r10 = dm(Right_Channel1_Out);    // Write to DAC1
40     dm(TXSP1A)=r10;
41     r10 = dm(Right_Channel2_Out);    // Write to DAC2
42     dm(TXSP1B)=r10;
43     r10 = dm(Right_Channel3_Out);    // Write to DAC3
44     dm(TXSP2A)=r10;
45     r10 = dm(Right_Channel4_Out);    // Write to DAC4
46     dm(TXSP2B)=r10;
47
48 Right_channel.end: rts;
```

Im SPORT0-Interrupt wird festgestellt, welcher Kanal gerade ausgelesen wurde (Variable Toggle_Channel) und das entsprechende Unterprogramm aufgerufen. In diesem wird der eingelesene Wert gespeichert und die vorher für diesen Kanal berechneten

Werte auf den DACs ausgegeben. Stehen bereits wieder für beide Kanäle ein neuer Wert für die Berechnung zur Verfügung wird außerdem das Unterprogramm DSP_PROGRAM aufgerufen (Zeile 20). In dieser Routine wird der anwendungsspezifische Algorithmus programmiert. Um zu gewährleisten, dass für die Programmierung desselben alle Register des Prozessors zur Verfügung stehen, wird das im Interrupt verwendete Register zwischengespeichert (Zeile 18 bzw. 22).

labor.asm

```
1  Init_variables:
2      bit set mode1 ALUSAT;    // enable ALU saturation (for FIX)
3      r10=1;
4      dm(Toggle_Channel)=r10; // first sample is from right channel
5  Init_variables.end: rts;
6
7
8  DSP_program:
9
10     dm(save_astat) = astat;      //not saved automatically at int.
11     r0 = dm(Right_Channel_In);   //right channel input sample
12     r1 = dm(Left_Channel_In);    //left channel input sample
13
14     //convert samples to floats
15     r0 = LSHIFT r0 BY 8;
16     r1 = LSHIFT r1 BY 8;
17
18     r2 = -31;
19     f0 = float r0 by r2;
20     f1 = float r1 by r2;
21
22     call Process_samples;        //processing of samples
23
24     //convert samples back to fractions
25     r2 = 31;
26     r0 = fix f0 by r2;
27     r1 = fix f1 by r2;
28
29     r0 = LSHIFT r0 BY -8;
30     r1 = LSHIFT r1 BY -8;
31
32
33     dm(Left_Channel1_Out) = r1;    //send to left  DAC1 channel
34     dm(Right_Channel1_Out) = r0;   //send to right DAC1 channel
35
36     dm(Left_Channel2_Out) = r1;    //send to left  DAC2 channel
37     dm(Right_Channel2_Out) = r0;   //send to right DAC2 channel
38
39     dm(Left_Channel3_Out) = r1;    //send to left  DAC3 channel
40     dm(Right_Channel3_Out) = r0;   //send to right DAC3 channel
41
42     dm(Left_Channel4_Out) = r1;    //send to left  DAC3 channel
43     dm(Right_Channel4_Out) = r0;   //send to right DAC3 channel
```

```
44
45      rts (db);
46          astat = dm(save_astat);    //restore ASTAT register
47          nop;
48
49  DSP_program.end:
50
51
52  Process_samples:
53
54      //place your code here
55
56      f0 = (f0+f1)/2;
57      f1 = f0;
58
59  Process_samples.end:     rts;
```

In der Routine INIT_VARIABLES (Zeilen 1-5) können wie bereits erwähnt globale Variable initialisiert werden. In der Routine DSP_PROGRAM werden zuerst die beiden von den ADCs eingelesenen Samples für die Bearbeitung ins FLOAT-Format konvertiert (Zeilen 17-22). Nun erfolgt der Aufruf des Unterprogramms PROCESS_SAMPLES, in dem der anwendungsspezifische Algorithmus programmiert wird. Zum Schluss werden die berechneten Werte wieder in ein für die Ausgabe an den DACs gültiges Format gebracht (Zeilen 27-32) und an diesen ausgegeben.

2.2 C-Programm - block sample

Dieses Programm basiert auf dem *Code Example* BLOCK-BASED TALK-THRU von *Analog Devices Inc.* und ist in der Programmiersprache C verfasst. Im Gegensatz zum oben beschriebenen Programm wird hier nicht mehr jeweils ein einzelner Wert vom ADC eingelesen und anschließend bearbeitet, sondern ein ganzer Block von Werten auf einmal übertragen. Dies ermöglicht eine ressourcensparendere Berechnung der Ausgangswerte und die Verwendung von blockbasierten Algorithmen wie etwa die FFT.

Durch die Verwendung von Blöcken können jedoch nicht nur deutlich effizientere Algorithmen verwendet werden, sondern auch die häufig auftretenden Interrupt-Aufrufe des SINGLE-SAMPLE-Programms vermieden werden. Außerdem wird nicht die halbe Rechenzeit lediglich durch das Warten auf den zweiten Kanal verschwendet.

Um dies zu gewährleisten kann jedoch nicht mehr mit einem ausschließlich INTERRUPT-gesteuerten Programm gearbeitet werden, weshalb der Aufruf des Unterprogramms zur Berechnung der Ausgangswerte in die Endlosschleife der MAIN-Routine verlegt wurde.

main.c

```
1     for(;;)
2     {
3      while(blockReady)
4           processBlock(src_pointer[int_cntr]);
5     }
```

Zusätzlich muss gewährleistet werden, dass die Berechnung abgeschlossen ist, bevor diese vom nächsten Interrupt unterbrochen wird. Andernfalls käme es zu einem Überschreiben der gerade verwendeten Daten (siehe weiter unten).

initSPORT.c

```
1  unsigned int Block_A[NUM_SAMPLES] ;
2  unsigned int Block_B[NUM_SAMPLES] ;
3  unsigned int Block_C[NUM_SAMPLES] ;
4
5  //Set up the TCBs to rotate automatically
6  int TCB_Block_A[4] = { 0, sizeof(Block_A), 1, 0};
7  int TCB_Block_B[4] = { 0, sizeof(Block_B), 1, 0};
8  int TCB_Block_C[4] = { 0, sizeof(Block_C), 1, 0};
9
10 void InitSPORT()
11 {
12     //Proceed from Block A to Block C
13     TCB_Block_A[0] = (int) TCB_Block_C + 3 - OFFSET + PCI ;
14     TCB_Block_A[3] = (unsigned int) Block_A - OFFSET ;
15
16     //Proceed from Block B to Block A
17     TCB_Block_B[0] = (int) TCB_Block_A + 3 - OFFSET + PCI ;
18     TCB_Block_B[3] = (unsigned int) Block_B - OFFSET ;
19
20     //Proceed from Block C to Block B
21     TCB_Block_C[0] = (int) TCB_Block_B + 3 - OFFSET + PCI ;
22     TCB_Block_C[3] = (unsigned int) Block_C - OFFSET ;
23
24     //Clear the Mutlichannel control registers
25     *pSPMCTL0 = 0;
26     *pSPMCTL1 = 0;
27     *pSPMCTL2 = 0;
28     *pSPCTL0 = 0 ;
29     *pSPCTL1 = 0 ;
30     *pSPCTL2 = 0 ;
31
32     //=============================================================
33     //
34     // Configure SPORT0 for input from ADC
35     //
36     //-------------------------------------------------------------
37
38
39     *pSPCTL0 = (OPMODE | SLEN24 | SPEN_A | SCHEN_A | SDEN_A);
```

```
40
41      // Enabling Chaining
42      // Block A will be filled first
43      *pCPSP0A = (unsigned int) TCB_Block_A - OFFSET + 3 ;
44
45      //=============================================================
46      //
47      // Configure SPORTs 1 & 2 for output to DACs 1-4
48      //
49      //-------------------------------------------------------------
50
51      #ifdef DAC1
52      *pSPCTL1 = (SPTRAN | OPMODE | SLEN24 | SPEN_A | SCHEN_A | SDEN_A) ;
53      // write to DAC1
54      *pCPSP1A = (unsigned int) TCB_Block_C - OFFSET + 3 ;
55      #endif
56
57      #ifdef DAC2
58      *pSPCTL1 |= (SPTRAN | OPMODE | SLEN24 | SPEN_B | SCHEN_B | SDEN_B) ;
59      // write to DAC2
60      *pCPSP1B = (unsigned int) TCB_Block_C - OFFSET + 3 ;
61      #endif
62
63      #ifdef DAC3
64      *pSPCTL2 = (SPTRAN | OPMODE | SLEN24 | SPEN_A | SCHEN_A | SDEN_A) ;
65      // write to DAC3
66      *pCPSP2A = (unsigned int) TCB_Block_C - OFFSET + 3 ;
67      #endif
68
69      #ifdef DAC4
70      *pSPCTL2 |= (SPTRAN | OPMODE | SLEN24 | SPEN_B | SCHEN_B | SDEN_B) ;
71      // write to DAC4
72      *pCPSP2B = (unsigned int) TCB_Block_C - OFFSET + 3 ;
73      #endif
74  }
```

Um die blockweise Verarbeitung der Daten zu ermöglichen, müssen drei gleiche Blöcke mit der gewünschten Größe, d.h. mit der gewünschten Anzahl von Samples pro Block, angelegt werden (Zeile 1-3), und zwar

- zum Einlesen der Daten vom ADC,
- zur Berechnung der neuen Ausgangsdaten
- und zur Ausgabe der Daten an die DACs.

Außerdem müssen die seriellen Schnittstellen nun so konfiguriert werden, dass diese automatisch die Daten in die richtigen Speicherbereiche schreiben bzw. aus diesen auslesen. Dies erfolgt während der Initialisierung des DSPs in der Funktion InitSPORT. Da diese Routine nahezu unverändert aus dem *Code Example* übernommen wurde, wird für Näheres auf [4, 5] verwiesen. Nur das gezielte Ein- und Ausschalten der vier DAC-Paare über die Variablen DAC1-DAC4 wurde hinzugefügt (#ifdef DACx ... #endif am Ende der

Routine) und kann in der Header-Datei MAIN.H eingestellt werden. Um größere Änderungen am Code zu vermeiden, wird im Unterschied zum SINGLE-SAMPLE-Programm jedoch auf allen DACs dasselbe Signal ausgegeben.

SPORTisr.c

```
1  extern int Block_A[NUM_SAMPLES] ;
2  extern int Block_B[NUM_SAMPLES] ;
3  extern int Block_C[NUM_SAMPLES] ;
4  extern int OFFSET ;
5
6  int *src_pointer[3] = {Block_A, Block_C, Block_B};  //Pointer to the blocks
7
8  // Counter to choose which buffer to process
9  int int_cntr=2;
10 // Semaphore to indicate to main that a block is ready for proc.
11 volatile int blockReady=0;
12 // Semaphore to indicate to the isr that the processing has not
13 // completed before the buffer will be overwritten.
14 volatile int isProcessing=0;
15
16 //If the processing takes too long, the program will be stuck in this
       infinite loop.
17 void ProcessingTooLong(void)
18 {
19     while(1);  //deactivate for debugging
20 }
21
22 void TalkThroughISR(int sig_int)
23 {
24     if(isProcessing)
25         ProcessingTooLong();
26
27     //Increment the block pointer
28     int_cntr++;
29     int_cntr %= 3;
30
31     blockReady = 1;
32 }
```

Die Variable SRC_POINTER enthält die Startadressen der drei Datenblöcke. Über die Variable INT_CNTR wird der aktuell zu bearbeitende Block angezeigt und der Routine PROCESSBLOCK übergeben. Zu Beginn ist dies Block_B, während in Block_A neue Werte eingelesen werden und Block_C über die DACs ausgegeben wird.

Um, wie oben bereits erwähnt, zu überprüfen, ob die Verarbeitung des alten Blocks bereits abgeschlossen ist, wird die Variable ISPROCESSING verwendet, welche am Beginn der Berechnung gesetzt und beim Abschluss derselben wieder gelöscht wird. Ist die Berechnung beim Aufruf des Interrupts noch in Gang, geht der Prozessor in eine Endlosschleife (Zeile 19), welche, da der Aufruf direkt aus dem Interrupt erfolgt, nicht mehr verlassen wird. In der Funktion PROCESSINGTOOLONG kann auch eine allfällige Fehlerbehandlung durchgeführt werden.

blockProcess.c

```
1  void processBlock(int *block_ptr)
2  {
3      int i;
4      //float temp_out;
5      static float leftChannel[NUM_SAMPLES/2];
6      static float rightChannel[NUM_SAMPLES/2];
7      //Clear the Block Ready Semaphore
8      blockReady = 0;
9
10     //Set Processing Active Semaphore before starting processing
11     isProcessing = 1;
12
13     //convert to float
14     for(i=0;i<NUM_SAMPLES/2;i++)
15     {
16         rightChannel[i]  = (float) (((*(block_ptr+(2*i)))<<8)*ADC_SCALE);
17         leftChannel[i] = (float) (((*(block_ptr+(2*i)+1))<<8)*ADC_SCALE);
18     }
19
20     DSP_program(leftChannel, rightChannel);
21
22     //convert back
23     for(i=0;i<NUM_SAMPLES/2;i++)
24     {
25         *(block_ptr+(2*i)) = (((int) ((rightChannel[i])*DAC_SCALE))>>8)&0
               x00FFFFFF;
26         *(block_ptr+(2*i)+1) = (((int) ((leftChannel[i])*DAC_SCALE))>>8)&0
               x00FFFFFF;
27     }
28
29     //Clear Processing Active Semaphore after processing's complete
30     isProcessing = 0;
31 }
```

Die Routine processBlock ist analog zur Routine DSP_program im oben vorgestellten Assembler-Programm. Jedoch wird die Konvertierung ins float-Format bzw. die Rücktransformierung in ein passendes Fixpoint-Format nun für alle Samples des aktuellen Blockes durchgeführt.

DSP_program.c

Die Berechnungen der Ausgangsdaten wird in der Funktion DSP_PROGRAM durchgeführt.

```
1  void DSP_program(float *leftChannel, float *rightChannel)
2  {
3      int i;
4
5      for (i=0;i<NUM_SAMPLES/2;i++)
6      {
7          //Place your code here
8          leftChannel[i]=leftChannel[i];
9          rightChannel[i]=-rightChannel[i];
10     }
11 }
```

2.3 C-Programm - single sample

Das in C geschriebene SINGLE SAMPLE Programm ist eins zu eins dem in Assembler geschriebenen, oben vorgestellten Musterprogramm nachempfunden, weshalb sich eine genauere Betrachtung erübrigt. Allein die Konvertierung in das für die Berechnung zu bevorzugende FLOAT-Zahlenformat wird wie im oben vorgestellten BLOCK-BASED Programm, welches ebenfalls in C verfasst wurde, durchgeführt. Außerdem wird analog auch die Programmierung des anwendungsspezifischen Algorithmus nun in der Funktion DSP_PROGRAM vorgenommen.

2.4 Assembler - block sample

Wie für das in C geschriebene SINGLE SAMPLE Programm gilt auch für dieses Musterprogramm, dass es eine nahezu exakte Übersetzung des in Abschnitt 2.2 vorgestellten BLOCK-BASED Programms darstellt.

Im Gegensatz zum ersten Assembler-Programm müssen aufgrund der höheren Komplexität vor dem Aufruf des Signalverarbeitungsalgorithmus jedoch mehr Register gesichert werden, wozu geeignete Unterprogramme (RESTORE_REG, SAVE_REG) geschrieben wurden. Außerdem wurde die Endlosschleife im Falle einer zu langsamen Verarbeitung des Datenblocks entfernt. Diese kann jedoch ohne großen Aufwand analog zum C-Programm wieder hinzugefügt werden.

3 C-Programm für die FFT-Filterbank

Die Funktionsweise und eine Anwendung der FFT-Filterbank wird in den Seminarunterlagen erklärt [3]. Da die FFT eine Blockverarbeitung benötigt, baut das Musterprogramm auf dem in Abschnitt 2.2 vorgestellten Rumpfprogramm mit blockweiser Verarbeitung auf. Das Problem dabei ist jedoch die benötigte zeitliche Überlappung der Blöcke (Frames), die im Rumpfprogramm mit den 3 Puffern nicht vorgesehen ist. Für die FFT-Filterbank wird nämlich alle M Abtastintervalle ein Block von N Abtastwerten benötigt ($M < N$). Aufeinanderfolgende Blöcke haben damit eine Überlappung von $N - M$ Speicherwerten (siehe Abb. 9 in Abschnitt 4 von [3]).

Wir können diese Überlappung jedoch sehr einfach mit einem Zwischenspeicher realisieren, wenn die Blocklänge N ein Vielfaches der Blockverschiebung M ist. Dazu sind in

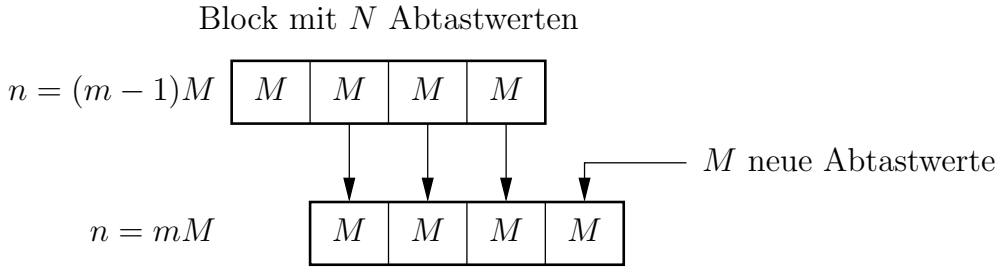

Abbildung 1: Laden des Zwischenspeichers für zwei zeitlich aufeinanderfolgende Eingangssignalblöcke (Zeitindex n, Frameindex m, $M = N/4$)

Abb. 1 zwei zeitlich aufeinanderfolgende Eingangssignalblöcke darstellt. Vor dem Laden von M neuen Abtastwerten in den Zwischenspeicher werden die alten Teilblöcke im Zwischenspeicher um M Plätze nach links verschoben. Im frei werdenden rechten Teilblock erfolgt dann das Speichern der neuen Abtastwerte.

Die Overlap-Add Operation am Ausgang der Filterbank wird, wie beim Musterprogramm der FFT-Filterbank für den ADSP-21065L, mit einem zyklischen Puffer realisiert.

Im Hauptteil der folgenden Programmausdrucke werden neben der Initialisierung der DSP-Funktionseinheiten und der Interrupts die benötigten Puffer angelegt. Danach wartet der Prozessor bis ein Block mit M neuen Abtastwerten verfügbar ist. Der Zwischenspeicher `xbuf` ist vom Typ COMPLEX, wobei der linke Eingangskanal im Realteil und der rechte Kanal im Imaginärteil gespeichert werden.

3.1 Beschreibung des Musterprogramms der FFT-Filterbank

```
1 /* FFT filter bank (C version, stereo signals)
2 based on MATLAB test program cfft_fib1.m
3 Fs = 16 kHz, select optimization in project options
4
5 DAC4 (= phone jack) volume can be increased with PB1, and decreased with
      PB2
6
```

```
7  if gain_flag = 1, then spectra are multiplied by factor 0.5
8
9  Note: C memory map offers 16K floats in dm, 8K floats in pm, 8K floats in
       heap, and 8K floats in stack, arrays must be distributed among these
       memory sections to avoid out of memory message in case of FFT length =
       2048
10
11 G. Doblinger, TU-Wien, 10.03.03, 1-2013
12 */
13
14 #include <stdlib.h>
15 #include <math.h>
16 #include <complex.h>
17 #include <filter.h>
18
19 #include "tt.h"
20
21 void modify_spectra(complex_float dm *X);
22
23 int gain_flag = 1;      // apply gain factor to spectra, if set to 1
24
25 static complex_float dm  xbuf[N];
26 static float pm          h[N];
27 static float             *ybuf_l, *ybuf_r;
28 static unsigned int      mp;
29
30 void main(void)
31 {
32     int   n;
33     float arg, tmp;
34
35     //Initialize PLL to run at CCLK= 331.776 MHz & SDCLK= 165.888 MHz.
36     //SDRAM is setup for use, but cannot be accessed until MSEN bit is
           enabled
37
38     InitPLL_SDRAM();
39
40     // Setting up IRQ0 and IRQ1
41
42     SetupIRQ01() ;
43
44     // Need to initialize DAI because the sport signals need to be routed
45
46     InitSRU();
47
48     // This function will configure the codec on the kit
49
50     Init1835viaSPI();
51
52     // Finally setup the sport to receive / transmit the data
53
54     InitSPORT();
55
56     interrupt (SIG_SP0,TalkThroughISR);
```

```
57      interrupt (SIG_IRQ0, Irq0ISR) ;
58      interrupt (SIG_IRQ1, Irq1ISR) ;
59
60      // init. buffers, and create time window function (hanning)
61      // Note: use calloc to place ybuf_l,r in heap memory (to save dm data)
62
63      ybuf_l = (float *) calloc(N, sizeof(float));
64      ybuf_r = (float *) calloc(N, sizeof(float));
65
66      arg = 2.0*PI/(N-1);
67      tmp = 0.5763*sqrtf((float) M/ (float) N);  // needed to obtain overall
            gain = 1
68      for (n = 0; n < N; n++)
69          {
70             xbuf[n].re = 0.0;
71             xbuf[n].im = 0.0;
72             h[n] = tmp*(1.0-cosf(arg*n));
73          }
74
75      mp = 0;                 // init. index used to address cyclic OLA buffer
76
77      // Be in infinite loop and do nothing until done.
78
79      for(;;)
80        {
81           while(blockReady)
82              processBlock(src_pointer[int_cntr]);
83        }
84
85  }
```

Wenn mit DMA ein Block mit M Abtastwerten geladen ist, wird das folgende Programm aufgerufen, das die Analyse- und Syntheseseite der FFT-Filterbank und die Funktion `modify_spectra()` zur Modifikation der Signalspektren enthält. Am Beginn dieser Funktion werden die Zwischenspeicherwerte wie in Abb. 1 zuerst nach links verschoben und danach die M neuen Abtastwerte in die freien Speicherplätze von `xbuf` geladen. Der gesamte Zwischenspeicherinhalt wird mit der Fensterfunktion $h[n]$ multipliziert und im FFT-Eingangspuffer `Xbuf` gespeichert. Die FFT-Funktion überschreibt `Xbuf` mit dem Spektrum des komplexwertigen Signals. Die Trennung der Spektren des linken und des rechten Eingangssignals wird weiter unten beschrieben. Nach der spektralen Modifikation wird im Syntheseteil der FFT-Filterbank das Ausgangssignal durch die Overlap-Add Operation mit Hilfe der zyklischen Puffer `ybuf_l` und `ybuf_r` berechnet.

```
1   void DSP_program(float *leftChannel, float *rightChannel)
2   {
3      int   n,k,l;
4      complex_float dm Xbuf[N];          // locate Xbuf is stack memory (to save
           dm data)
5
6      /* shift samples of input buffer by M samples to the left to store
7         new M input samples in last subblock
```

```
8         e.g. M = N/4, i.e. 4 subblocks per input buffer: |M|M|M|M|
9      */
10
11     for (n = 0; n < L-1; n++)
12        for (k = 0; k < M; k++)
13           {
14              l = k+n*M;
15              xbuf[l].re = xbuf[l+M].re;
16              xbuf[l].im = xbuf[l+M].im;
17           }
18
19     // store new M input samples in last subblock of input buffer
20
21     for (n = 0; n < M; n++)
22        {
23           xbuf[N-M+n].re = leftChannel[n];
24           xbuf[N-M+n].im = -rightChannel[n];
25        }
26
27     // apply time window and FFT, copy buffer to Xbuf since cfft changes
          input buffer
28
29     for (n = 0; n < N; n++)
30        {
31           Xbuf[n].re = h[n]*xbuf[n].re;
32           Xbuf[n].im = h[n]*xbuf[n].im;
33        }
34
35     // use inplace FFT to save memory space
36
37  #ifdef FFT2048
38     cfft2048(Xbuf, Xbuf);
39  #elif FFT1024
40     cfft1024(Xbuf, Xbuf);
41  #else
42     cfft512(Xbuf, Xbuf);
43  #endif
44
45     // modify spectra of left and right channel
46
47     modify_spectra(Xbuf);
48
49     // synthesis stage of FFT filter bank
50
51  #ifdef FFT2048
52     ifft2048(Xbuf, Xbuf);
53  #elif FFT1024
54     ifft1024(Xbuf, Xbuf);
55  #else
56     ifft512(Xbuf, Xbuf);
57  #endif
58
59     /* carry out overlap-add of successive IFFT outputs
60        (overlap-add is done with a cyclic buffer ybuf in order to store only
```

```
            N samples,
61          note that N is an integer multiple of M)
62      */
63
64      // overlap-add N-M samples in cyclic buffer ybuf
65
66      for (n = 0; n < N-M; n++)
67          {
68              ybuf_l[mp] += h[n]*Xbuf[n].re;
69              ybuf_r[mp] += h[n]*Xbuf[n].im;
70              mp++;
71              mp %= N;
72          }
73
74      // copy last M samples to buffer ybuf
75
76      for (; n < N; n++)
77          {
78              ybuf_l[mp] = h[n]*Xbuf[n].re;
79              ybuf_r[mp] = h[n]*Xbuf[n].im;
80              mp++;
81              mp %= N;
82          }
83
84      // copy M samples of output buffer to output signal
85
86      for (n = 0; n < M; n++)
87          {
88              leftChannel[n] = ybuf_l[mp];
89              rightChannel[n] = ybuf_r[mp];
90              mp++;
91              mp %= N;
92          }
93
94  }
```

In der folgenden Funktion `modify_spectra()` wird zunächst die FFT des linken Signals $x_l[n]$ und des rechten Signals $x_r[n]$ aus der FFT $X[k]$ des komplexwertigen Eingangssignals $x[n] = x_l[n] + jx_r[n]$ berechnet. Dazu verwenden wir die folgenden Eigenschaften der Fouriertransformation [8]:

$$x_l[n] = \Re e\{x[n]\} \quad \Longleftrightarrow \quad X_l[k] = \frac{1}{2}(X[k] + X^*[N-k]), \quad k = 0, 1, \ldots, N-1$$

$$x_r[n] = \Im m\{x[n]\} \quad \Longleftrightarrow \quad X_r[k] = \frac{1}{2j}(X[k] - X^*[N-k]), \quad k = 0, 1, \ldots, N-1.$$

Die Werte an den Frequenzen $f = 0$ $(k = 0)$ und $f = F_s/2$ $(k = N/2)$ werden nicht modifiziert, da diese Werte durch die analogen Vor- und Nachfilter des DSP-Entwicklungssystems unterdrückt werden. Wie im Kommentar des Programms angege-

ben, wird $X[k]$ durch $X_l[k]$ und $X_r[k]$ überschrieben:

$$\begin{aligned} X[k] &= X_l[k], \quad k = 1, 2, \ldots, N/2 - 1 \\ X[N-k] &= X_r[k], \quad k = 1, 2, \ldots, N/2 - 1. \end{aligned}$$

Zu beachten ist, dass $X_r[k]$ mit umgekehrter Indexreihenfolge gespeichert wird, um zusätzliche Speicheroperationen zu vermeiden. Die Modifikation der spektralen Werte erfolgt mit der Funktion `scale_FFT_bins()`, die als Beispiel nur eine Amplitudenänderung bewirkt. Diese Funktion wird für die einzelnen Aufgaben (Pitch Scaling, Speech Enhancement, Channel Vocoder, etc,) angepasst. Zum Schluss wird mit den modifizierten Spektren das Spektrum des komplexwertigen Signals gebildet, das für die Aktualisierung der Overlap-Add Puffer im Filterbanksyntheseteil benötigt wird:

$$\begin{aligned} X[k] &= X_l[k] + jX_r[k], \quad k = 1, 2, \ldots, N/2 - 1 \\ X[N-k] &= X_l[N-k] + jX_r[N-k] \\ &= X_l^*[k] + jX_r^*[k], \quad k = 1, 2, \ldots, N/2 - 1. \end{aligned}$$

Dabei haben wir die hermitesche Symmetrie der Fouriertransformationen der reellwertigen Signale $x_l[n]$ und $x_r[n]$ verwendet [8]. Die umgekehrte Indexreihenfolge von $X_r[k]$ wird in der letzten FOR-Schleife automatisch korrigiert. Damit das funktioniert, darf in der Funktion `scale_FFT_bins()` die Indexreihenfolge nicht verändert werden!

```
1  /* spectral modification functions for FFT filter bank
2     G. Doblinger, 03-03, 08-03, 1-2013 */
3
4  #include <stdlib.h>
5  #include <math.h>
6  #include <complex.h>
7  #include "tt.h"
8
9  static void scale_FFT_bins(complex_float dm *X, float);
10 extern int gain_flag;
11
12 void modify_spectra(complex_float dm *X)
13
14    /* modify spectra of left and right channel signals (or first,
15       second signal block in case of monaural signals
16       (real-valued signals, use symmetry of spectra, inplace storage)
17       Note: frequency points 0 and Fs/2 are not modified since they are
             suppressed by input/output anti-aliasing filters */
18 {
19    int      k, l;
20    float    X1, X2, X3, X4;
21
22    /* separate spectra Xl, Xr of left and right channels from spectrum X =
         Xre + jXim using
23
24       Xl[k] = 0.5*(X[k] + conj(X[N-k]))
25       Xr[k] = -0.5*j*(X[k] - conj(X[N-k]))
26
```

```
27          Xl stored in Xre[k]+jXim[k]   k = 0...N/2-1
28          Xr stored in Xre[k]+jXim[k]   k = N-1...N/2 (reversed order!)
29          (compute only half of frequency points (k = 0...N/2) by using
                symmetry)
30          Note: 0.5 is omitted to avoid multiplications (corrected with window
                h[n]) */
31
32       for (k = 1, l = N-1; k < N/2; k++, l--)
33          {
34             X1 = X[k].re;
35             X2 = X[k].im;
36             X3 = X[l].re;
37             X4 = X[l].im;
38             X[k].re = X1 + X3;
39             X[k].im = X2 - X4;
40             X[l].re = X2 + X4;
41             X[l].im = X3 - X1;
42          }
43
44       /* modify N/2+1 points of spectra Xl, Xr (if wanted)
45          Xl stored in Xre[k]+jXim[k]   k = 0...N/2-1
46          Xr stored in Xre[k]+jXim[k]   k = N-1...N/2 (reversed order!)
47          (compute only half of frequency points (k = 0...N/2) by using
                symmetry) */
48
49       if (gain_flag) /* apply gain factor to spectra */
50          {
51             scale_FFT_bins(X, 0.5);                  /* left channel spectrum */
52             scale_FFT_bins(&X[N/2], 0.5);            /* right channel spectrum */
53          }
54
55       /* combine spectra Xl, Xr back to spectrum X = Xre+jXim
56          (Note: reverse storage of Xr is automatically corrected!)
57
58          X[k] = Xl[k] + j*Xr[k]                   k = 1...N/2-1
59          X[N-k] = conj(Xl[k]) + j*conj(Xr[k])     k = 1...N/2-1 */
60
61       for (k = 1, l = N-1; k < N/2; k++, l--)
62          {
63             X1 = X[k].re;                          /* Re{Xl[k]} */
64             X2 = X[k].im;                          /* Im{Xl[k]} */
65             X3 = X[l].re;                          /* Re{Xr[k]} */
66             X4 = X[l].im;                          /* Im{Xr[k]} */
67
68             X[k].re = X1 - X4;
69             X[k].im = X2 + X3;
70             X[l].re = X1 + X4;
71             X[l].im = X3 - X2;
72          }
73    }
74
75    /* ------------------------------------------------------------------
          */
76
```

```
77 static void scale_FFT_bins(complex_float dm *X, float gain)
78
79 /* compute  modify FFT real Xre[] and imaginary parts Xim[] according
80    to spectral amplitude modification
81
82    gain gain factor for frequency bin modification
83 */
84
85 {
86    int k;
87
88    for (k = 0; k < N/2; k++)
89       {
90          X[k].re = gain*X[k].re;
91          X[k].im = gain*X[k].im;
92       }
93 }
```

3.2 Übersicht über mögliche Anwendungen

Die folgenden Anwendungen können im Rahmen der DSP-Seminare unter Verwendung des ADSP-21369 Entwicklungssystems programmiert und untersucht werden. Unterlagen und Referenzen dazu werden individuell den Teilnehmern zur Verfügung gestellt.

3.2.1 Frequenzabhängige Dynamikkompression

Bei der frequenzabhängigen Dynamikkompression wird die Dynamik des Kurzzeitspektrums in individuellen Frequenzbändern verändert. Damit kann z.B. die frequenzabhängige Schwerhörigkeit simuliert werden. Umgekehrt kann durch eine Dynamikexpansion ein solcher Hördefekt kompensiert werden. Die Kurzzeitspektralanalyse wird mit der FFT-Filterbank realisiert, wobei die Beträge der FFT in vorgegebenen Frequenzbereichen durch nichtlineare Kennlinien modifiziert werden. Diese Kennlinien ergeben sich z.B. dadurch, dass die Amplitude - gemessen mit logarithmischer Amplitudenskala (dB-Skala) - durch stückweise lineare frequenzabhängige Verläufe verändert wird. Damit lassen sich Einbrüche im Dynamikbereich in bestimmten Frequenzbereichen einfach nachbilden.

3.2.2 Veränderung der Tonhöhe akustischer Signale

Die Veränderung der Tonhöhe akustischer Signale (Pitch Scaling) mit der FFT-Filterbank erfolgt durch multiplikative Veränderung der Frequenzskala mit Faktoren im Bereich $[0.5, 2]$. Die Dauer des Signals wird im Gegensatz zur Frequenzänderung durch Abtastratenänderung nicht beeinflusst. Die Funktionsweise des Algorithmus für Pitch Scaling ist in [3] beschrieben.

3.2.3 Noise Gate

Ein effektives Verfahren zur Unterdrückung additiver, zeitinvarianter Störungen bei Audiosignalen ist die Verwendung eines Noise Gate im Frequenzbereich. Damit kann bei-

spielsweise breitbandiges Rauschen in Tonaufzeichnungen einfach unterdrückt werden, ohne das Nutzsignal signifikant hörbar zu verformen. Bei Anwendung der FFT-Filterbank wird ein Schwellwert definiert. Spektrale Amplituden, deren Werte unterhalb der Schwelle liegen werden auf Null gesetzt. Mit einer Hysterese können kleine Amplitudenschwankungen um diesen Schwellwert vermieden werden. Das Verfahren ist auf einen Signal-Rausch-Abstand am Eingang von SNR > 20 dB beschränkt. Ein kleineres SNR bewirkt ein unangenehmes Restrauschen (Musical Noise) im entstörten Signal.

3.2.4 Entstörung von Sprachsignalen

Für starke, nichtweiße und zeitvariante Störsignale wird die FFT-Filterbank nicht mit harten Schwellwerten (Hard Thresholding) sondern mit adaptiv gesteuerten Kennlinien (Soft Thresholding) eingesetzt. Die Beeinflussung der spektralen Amplituden erfolgt dabei in Einklang mit Schätzwerten des momentanen SNR der einzelnen Spektralkomponenten. Dazu ist eine laufende Schätzung des Störsignalspektrums notwendig, die bei Sprachsignalen in den Sprachpausen durchgeführt wird. Alternativ kann das Störsignalspektrum auch durch Verfolgung der spektralen Minima des gestörten Sprachsignals geschätzt werden.

3.2.5 Simulation bewegter akustischer Quellen

Um den Höreindruck einer langsam bewegten Signalquelle zu simulieren, verwenden wir die ortsabhängigen Impulsantworten, die vom Ort der Schallquelle bis zum Gehöreingang gemessen werden. Diese Impulsantworten werden als Head-Related Impulse Responses (HRIRs) bezeichnet. Für die Simulation liegen HRIRs für alle Winkel im 5°-Abstand in einer horizontalen Ebene rund um den Kopf vor. Bei der Bewegungssimulation der Schallquelle rund um den Kopf werden die HRIRs schrittweise ausgewählt. Das Kopfhörersignal wird durch Filterung mit den zeitvarianten HRIRs gebildet. Eine einfache Filterung (Faltungsoperation) ist nicht sinnvoll, da das Umschalten der Impulsantworten Knackgeräusche erzeugt, die durch die abrupte Veränderung der Filterkoeffizienten entstehen. Hier kann die Filteroperation mit der FFT-Filterbank erfolgen, bei der durch die gewichtete Blocküberlappung der Analysestufe und durch die Overlap-Add Operation der Synthesestufe der Einfluss des Umschaltens der Impulsantworten (bzw. deren Übertragungsfunktionen) unterdrückt wird.

3.2.6 Richtungsschätzung mit zwei Mikrofonen

Die Analysestufe der FFT-Filterbank kann in Kombination mit der Berechnung der spektralen Kreuzleistung zweier Eingangssignal zur Richtungsschätzung verwendet werden. Die Mikrofonsignale einer Schallquelle haben je nach Richtung (Azimuth) unterschiedliche Laufzeiten. Die Laufzeitdifferenz kann robust durch Kreuzkorrelation der beiden Signalspektren gemessen werden und aus dem Ergebnis der Winkel (Azimuth) zur Quelle bestimmt werden. Durch Bildung des Phasenspektrums wird die Messung unabhängig von den spektralen Amplituden und ist relativ robust bzgl. rauschartiger Störungen und Nachhall.

Literatur

[1] Analog Devices Inc. *ADSP-21369 EZ-KIT Lite® Evaluation System Manual*, Revision 2.1, August 2006.

[2] **www.analog.com**

[3] Gerhard Doblinger, „Digitale Signalverarbeitung," Skriptum zur LVA 389.066 und 389.067, TU-Wien, Februar 2011

[4] Analog Devices Inc. *AD1835A 2 ADC, 8 DAC, 96 kHz, 24-Bit Sigma Delta Codec Data Sheet*, Revision A, December 2003.

[5] Analog Devices Inc. *ADSP-21368 SHARC® Processor Hardware Reference Includes ADSP- 21367, ADSP-21369, ADSP-21371, ADSP-21375*, Revision 1.0, September 2006.

[6] Analog Devices Inc. *ADSP-2136x SHARC® Processor Programming Reference*, Revision 1.1, March 2007

[7] Analog Devices Inc. *ADSP-21367/ADSP-21368/ADSP-21369 SHARC Processors Data Sheet*, Revision C, January 2008

[8] Gerhard Doblinger, „Zeitdiskrete Signale und Systeme," J. Schlembach Fachverlag, 2010