



Digital Signal Processing Lab

---

# Digital Signal Processing

## Lab

April 17, 2023

Hochschule für Angewandte Wissenschaften Hamburg  
Hamburg University of Applied Sciences

© 2023 Copyright

ANDREA KUPKE, PROF. DR.-ING. ULRICH SAUVAGERD, PROF. DR.-ING. LUTZ LEUTELT  
Hochschule für Angewandte Wissenschaften Hamburg,

All rights reserved.

Alle Rechte, auch das des auszugsweisen Nachdrucks, der auszugsweisen oder vollständigen Wiedergabe, der Speicherung in Datenverarbeitungsanlagen und der Übersetzung, vorbehalten.

Dieses Dokument wurde mit Hilfe von KOMA-Script und L<sup>A</sup>T<sub>E</sub>X gesetzt.

# Contents

<b>1. Getting Started: TMS320C6713, CCSv5.5 and Audio Sampling</b>	<b>5</b>
1.1. Objectives of this first lab session . . . . .	5
1.2. Lab preparation . . . . .	5
1.2.1. Interrupt handler and bit manipulation . . . . .	6
1.2.2. Sampling and quantization . . . . .	6
1.2.3. Sampling frequency and aliasing . . . . .	7
1.3. A first DSP project with Code Composer Studio v5 . . . . .	7
1.3.1. Start of CCSv5 and import of a project . . . . .	7
1.3.2. Compile, link and load the project . . . . .	8
1.3.3. Debugging the DSP program . . . . .	10
1.3.4. Removing errors and restarting the program . . . . .	11
1.3.5. Using the CCS Debugging tool . . . . .	12
1.4. Interrupt-driven Programming for Digital Signal Processing . . . . .	14
1.4.1. Gateway for signals: CODEC Interface and McBSP . . . . .	15
1.4.2. Project get_started with signal input and output . . . . .	16
1.4.3. First test of the project . . . . .	18
1.4.4. Left-/Right Switching of CODEC Channels . . . . .	19
1.4.5. Quantization . . . . .	19
1.4.6. ADC- /DAC-Overflow . . . . .	20
1.4.7. Aliasing Effects . . . . .	21
1.4.8. Audio Reverberation System . . . . .	21
<b>2. Radix-2 FFT and Real-Time Spectrum Analyser</b>	<b>23</b>
2.1. Objectives of this lab second session . . . . .	23
2.2. Preparation of the lab . . . . .	23
2.2.1. Analysis of a Butterfly . . . . .	24
2.2.2. 8-point FFT (DIT) . . . . .	24
2.2.3. Familiarize yourself with the lab project . . . . .	26
2.3. Lab: Spectrum Analysis using FFT . . . . .	27
2.3.1. Getting started with the c project . . . . .	27
2.3.2. Extension of the FFT to 64 points . . . . .	27
2.3.3. Real-time spectrum analyser . . . . .	28
<b>A. Appendix: Hardware DSK6713</b>	<b>31</b>
A.1. The DSK6713 Board . . . . .	31
A.2. The Interrupt System of the C6000 DSPs . . . . .	31
A.2.1. Configuration of the interrupt system . . . . .	34
A.3. Multichannel Buffered Serial Ports McBSP . . . . .	35
A.3.1. McBSP Pins and Registers . . . . .	35
A.3.2. Configuration of the McBSP . . . . .	36
A.4. The AIC23 CODEC . . . . .	38





# Getting Started: TMS320C6713, CCSv5.5 and Audio Sampling

## 1.1. Objectives of this first lab session

The purpose of this first lab project is to give an introduction to the hardware and software of the Digital Signal Processor board, the C6713 DSK board [\[A.1\]](#), which is used in this and all following lab sessions. You will step by step

- create a Code Composer Studio (CCS) project for the DSK board, adding custom and DSP-lab specific project files,
- compile and link the project and execute your project on the DSP Client,
- use the CCS debugging tool and correct errors in the source code,
- implement an interrupt service routine,
- get to know the CODEC-Interface and the usage of hardware interrupts
- and develop simple DSP programs which read audio signals from an audio source and output them through a CODEC (directly or after processing).

## 1.2. Lab preparation

It is very important that you work through these lab instructions before the lab session and that you are familiar with the fundamentals of “Signals and Systems 1+2” and “Programming in C”. If you need to catch up, please make yourself familiar with these topics of the previous semesters. In particular, answer all the preparation tasks in the light blue boxes (“Prep task”).

### Prep task (for lab entry test)

Familiarize yourself with the concepts of the chapter “DP01: Digitization and Digital Signals”, particularly

- sampling, sampling frequency, aliasing and quantization,
- DSP system C6713DSK board, interrupt-based sample-by-sample processing in C
- rounding of fixed-point numbers and techniques in C to avoid overflows after arithmetic operations

These topics will be addressed by the lab entry test at the beginning of the lab session.

### 1.2.1. Interrupt handler and bit manipulation

In your microcontroller class, you have learned how to do bit manipulation of integer values with bit masks and bitwise-logic operators (e.g. and, or, xor). Let an interrupt handler, which is called with every new stereo sample, perform a bit manipulation.

Listing 1.1: bit-mask.c.

```
1 interrupt void intser_McBSP1() {  
    // read both audio channels from McBSP1  
3    AIC23_data.both = MCBSP_read(DSK6713_AIC23_DATAHANDLE);  
  
5    AIC23_data.both &= 0xAAAAAAAA;  
  
7    // write both audio channels to McBSP1  
    MCBSP_write(DSK6713_AIC23_DATAHANDLE, AIC23_data.both);  
9    return;  
}
```

where AIC23\_data is a union declared by

Listing 1.2: union-decl.c.

```
1 union {  
2    unsigned int both;  
    short int channel[2];  
4 } AIC23_data;
```

### Prep task 1: Interrupt handler and bit manipulation

- Which decimal values are output after bit manipulation to channel[0] and channel[1] of the DAC, if the hexadecimal value received from ADC AIC23\_data.both was 0x7FFFFFFC.

### 1.2.2. Sampling and quantization

Let an analog cosine signal  $x(t) = \cos(2\pi f_0 t)$  with  $f_0 = 4\text{kHz}$  be sampled at  $f_S = 32\text{kHz}$  (in the lab you later use a different sampling frequency). The sampled discrete-time signal  $x[n]$  is afterwards

quantized by a 3-bit quantizer with amplitude input range  $R_{ADC} = [-1, +1[$ .

#### Prep task 2: Sampling and quantization

- Determine the sampled discrete-time signal  $x[n]$  (without quantization).
- Determine the six signal values  $x[n], \hat{x}[n]$ ,  $n = -1, 0, +1, \dots, +4$  **before and after** 3-bit quantization with truncation.

### 1.2.3. Sampling frequency and aliasing

Assume the real-time DSP system consisting of Audio Codec and DSP has a non-ideal low-pass characteristic as depicted in Figure 1.1. An analog cosine signal  $x(t) = 1/\pi * \cos(2\pi f_0 t)$  is applied to the system input at different frequencies  $f_0$ . Let the sampling frequency  $f_S$  of the Audio Codec be 32 kHz (in the lab you later use a different sampling frequency).

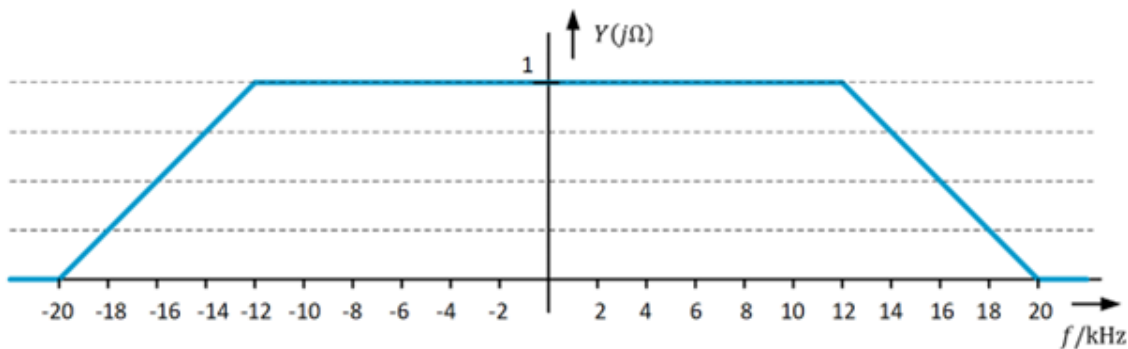


Figure 1.1.: Low-pass characteristic of a DSP system

#### Prep task 3: Sampling frequency and aliasing

- Explain why the DSP system has a low-pass filter characteristic.
- Sketch the spectrum  $Y(j\Omega)$  of the analog signal  $y(t)$  at the output of the DSP system with different colors for  $f_0 = 10$  kHz, 14 kHz, 16 kHz, 18 kHz and 20 kHz (in the same plot). It is *not* necessary to calculate the exact spectral amplitudes.

## 1.3. A first DSP project with Code Composer Studio v5

### 1.3.1. Start of CCSv5 and import of a project

Start CCS 5 by double clicking the corresponding icon on the Desktop.



Open an existing project via **Project**→**Import Existing Eclipse Project** and choose the directory get\_started D:\ti\_work5\DSK6713 (see Figure 1.2). Click **Finish** and you'll find the included files of the CCS project in the automatically opened Project Explorer.

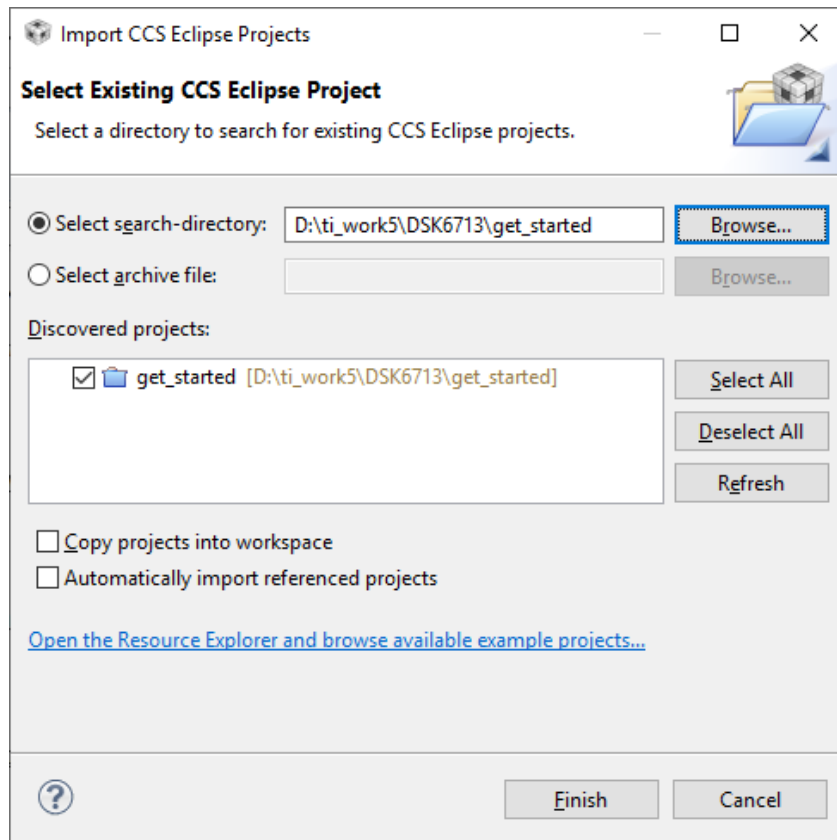


Figure 1.2.: Import a project

### 1.3.2. Compile, link and load the project

Connect the DSK6713 board via USB. Select now **Run** → **Debug** (< F11 >). By this command

- the inserted files will be **compiled**,
- **linked**, while the linker is interpreting the file C6713dsk\_AIC23.cmd
- and an **executable file** will be created.
- The target hardware (DSK board) will be **initialised and connected**,
- The executable file `..\get_started\Debug\get_started.out` will be **loaded** to the board.
- The program will be **started and stopped at the row main()**.

The „Problems“ window should not show any error or warning and should not be opened, respectively. After that, you are automatically switched to the Debug Perspective. You can return to the CCS Edit-Perspective via the buttons on the right top (Figure 1.3, encircled in red).

**Run** → **Resume** or <F8> or the corresponding button (Figure 1.3, green arrow on the left of the area circled in blue) starts the program on the DSP. You are asked for input in the „Console“ window.



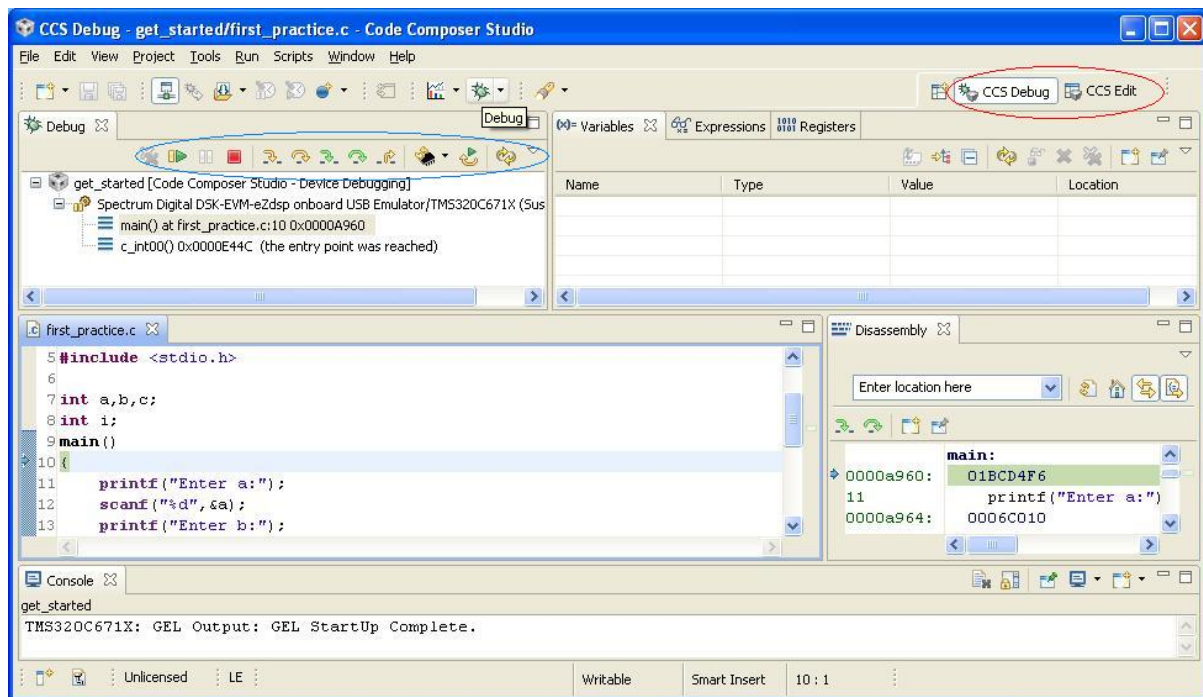


Figure 1.3.: View after Debug → Run

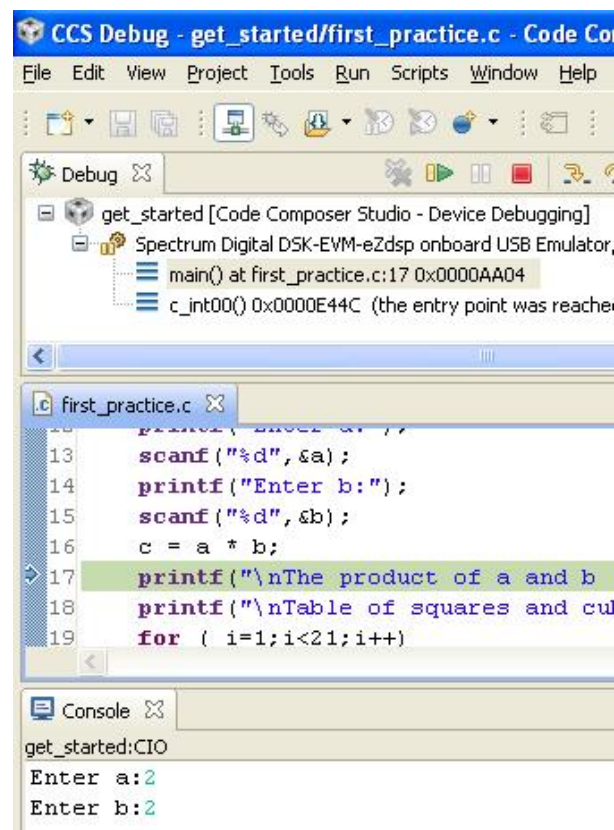


Figure 1.4.: Execute a project

Click into the Console window to input values, confirm with <Enter>. Output appears in this window as well. Figure 1.4 shows the window content prior to execution of the line highlighted in green (printf... ).

The program is not terminated automatically, but runs in an infinite loop while(1) until it is terminated manually.

### 1.3.3. Debugging the DSP program

**Run** → **Restart** restarts the program and waits at the first row of main().

**Run** → **Resume** executes the program, with the Single Step- functions stepwise.

**Run** → **Step Over** or <F6> executes the program line by line. The individual functions of the Run menu are listed in Figure 1.5.

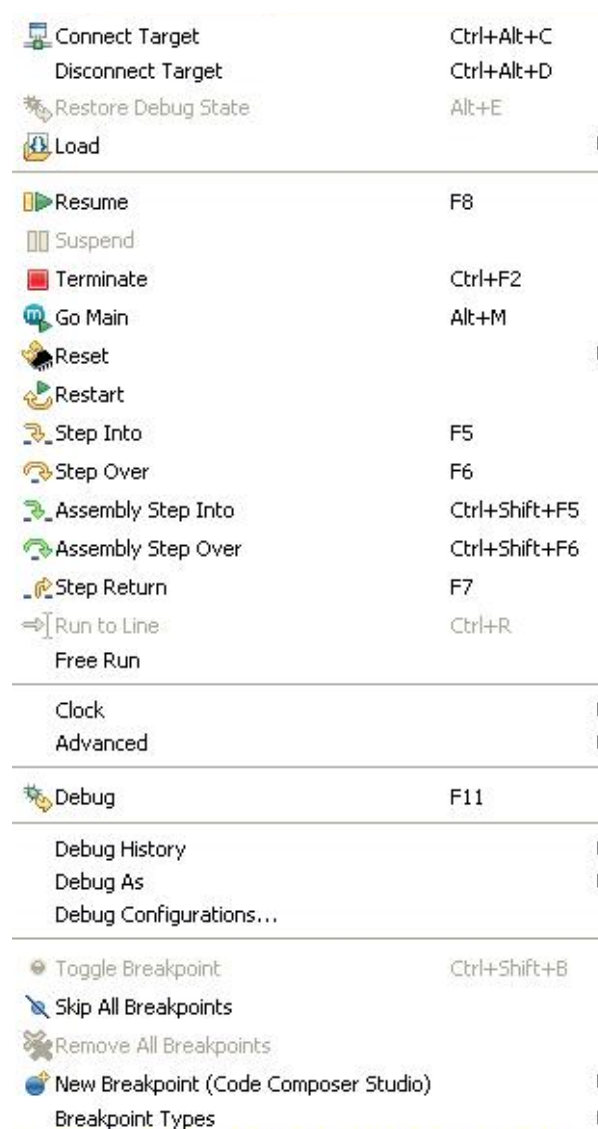


Figure 1.5.: Debug functions (Run menu)

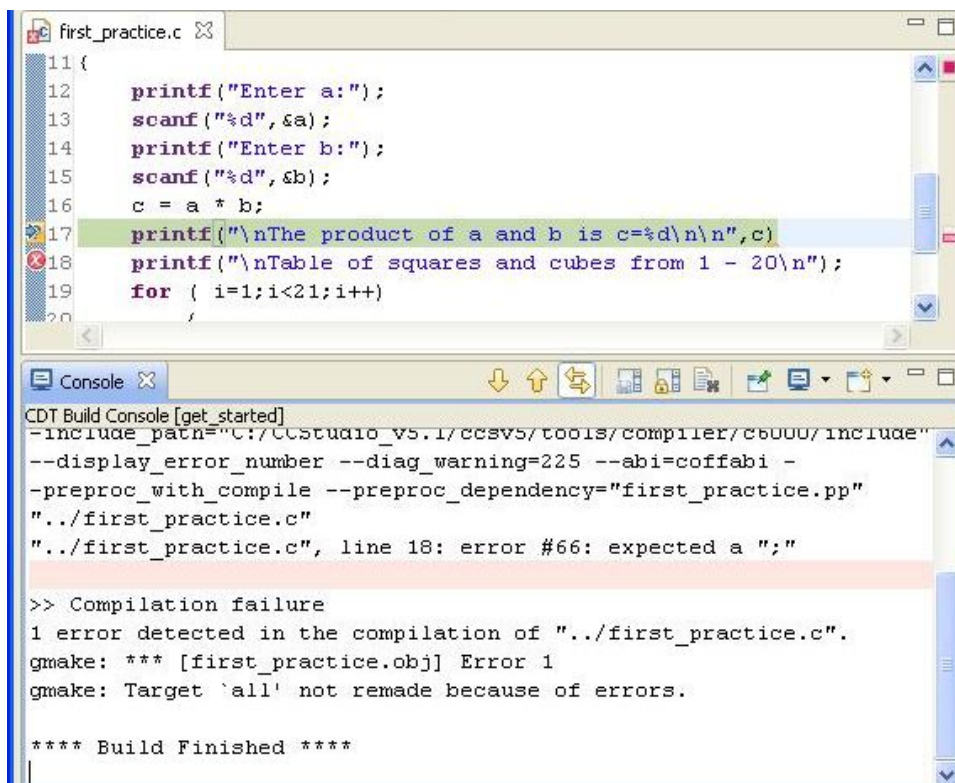
**Remark:** The provided sample program shows simple console input and output using the C functions `scanf` and `printf` and calculates the squares and cubes for numbers between 1 and 20. For functions `scanf` and `printf` so called I/O streams are used. These streams are buffered and sent from the DSK board to the host PC via serial interface. Hence, you **cannot expect real-time behaviour** with these functions. For real-time input/output you need to use other C functions, based on libraries provided by Texas Instruments.

### 1.3.4. Removing errors and restarting the program

In order to practice working with the debugger, we want to insert an error in our first project `first_practice.c` e.g. by removing the ";" behind the first `printf` command.

**Project**→**Build All** lists error messages in the console, see Figure 1.6. You can call **Run**→**Debug** as well, then you'll get a message in a separate window, that no executable file is available. In this case, error messages are listed in a new Problems window, see Figure 1.7.

A doubleclick on the error message places the cursor in the line behind the incorrect line. Correct the mistake and repeat the procedure. The error message should have disappeared again. **Run**→**Debug** is required to execute the program.



```
first_practice.c
11 {
12     printf("Enter a:");
13     scanf("%d", &a);
14     printf("Enter b:");
15     scanf("%d", &b);
16     c = a * b;
17     printf("\n\nThe product of a and b is c=%d\n\n", c)
18     printf("\n\nTable of squares and cubes from 1 - 20\n");
19     for ( i=1; i<21; i++)
20         ;

Console
CDT Build Console [get_started]
-include_path="C:/CCstudio_v5.1/ccsv5/tools/compiler/c60000/include"
--display_error_number --diag_warning=225 --abi=coffabi -
-preproc_with_compile --preproc_dependency="first_practice.pp"
"../first_practice.c"
"../first_practice.c", line 18: error #66: expected a ";"

>> Compilation failure
1 error detected in the compilation of "../first_practice.c".
gmake: *** [first_practice.obj] Error 1
gmake: Target 'all' not remade because of errors.

**** Build Finished ****
```

Figure 1.6.: Error message after Build All

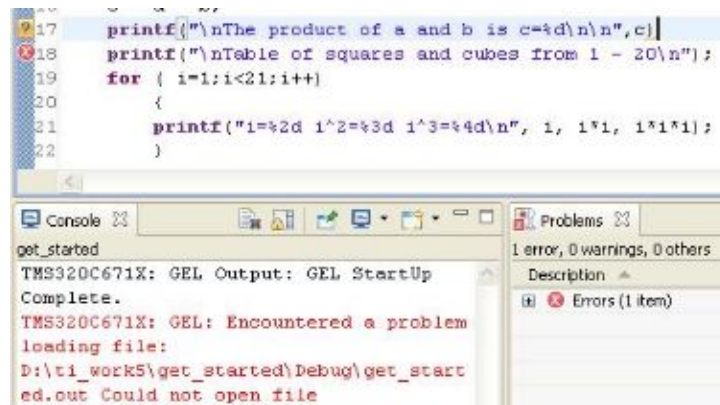


Figure 1.7.: Error message after Run→Debug

### 1.3.5. Using the CCS Debugging tool

CCS offers excellent debugging possibilities, which will be introduced step by step. Among others you can:

- set and remove breakpoints in the source code
- run the program in single-step mode
- view mixed source and assembly code
- monitor variables
- view the content of the DSP registers (DSP core registers, peripheral registers)
- view the content of any memory location.

Let us set a breakpoint before the product calculation by placing the cursor in front of the line 16:  $c = a * b$ ; and double-clicking. Alternatively, we can mark the line, right mouse click and select *Breakpoint→Breakpoint*. A blue dot shows the breakpoint. The program in Figure 1.8 runs up to the breakpoint. **View→Breakpoints** shows all breakpoints in a window.

You can set any number of breakpoints in any C or assembler code. Remember that the current position of program execution (i.e. the address of the program counter) is indicated by a blue arrow. In this way you can run the program piecewise. Individual breakpoints can be removed by double-clicking on the blue dot. *Toggle Breakpoint* in the menu (open with right mouse click), sets or removes the breakpoint, *Disable/Enable Breakpoint* can temporarily deactivate a breakpoint without removing it for later use.

Note that with the command *Assembly Single Step*, you can view the assembly code lines in the **Disassembly**-window. One or more (if more statements are executed in parallel) green arrows show the current position of the program counter. For inspecting C-code, the *Source Step Over* is more suitable than the *Source Single Step*.

You can see the Disassembly window (**View→Disassembly**) in the lower right part of Figure 1.8. In this window, you can observe assembly single steps. One or more (in case of parallel processing of statements) blue arrows show the current position of the program counter.

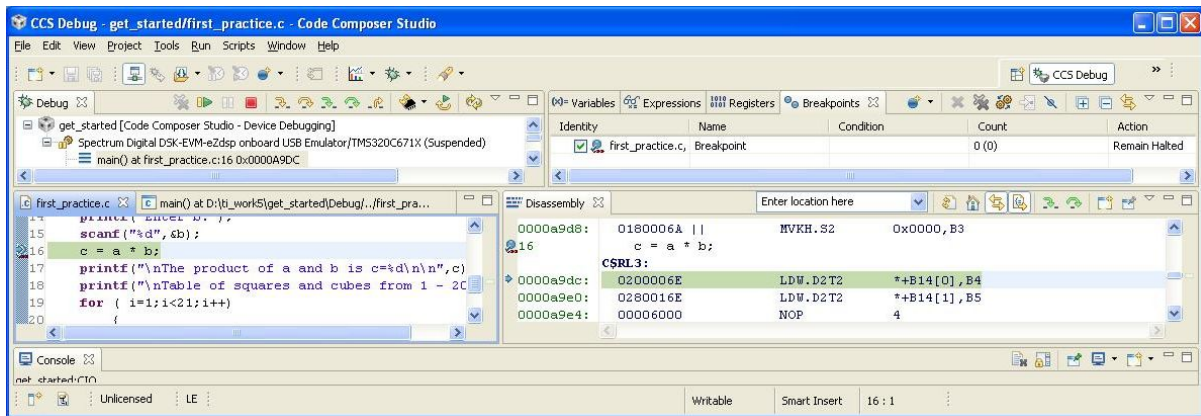


Figure 1.8.: Halt at a breakpoint

To **monitor a variable**, mark it, right mouse click, then select *<Add Watch Expression...>*. Alternatively call *View→Expressions*, add the variable in the opening window, see. Figure 1.9. You can also check the value of a variable without using the Expressions window: simply place the mouse pointer over the value of a variable, and its value (or the start address, if it is a function identifier) is displayed, if the program execution is currently halted.

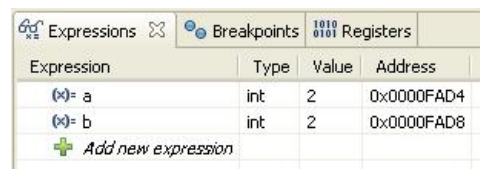


Figure 1.9.: Monitoring variables

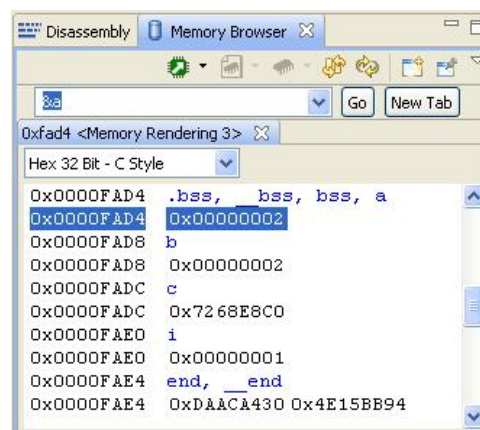


Figure 1.10.: View memory in Memory Browser

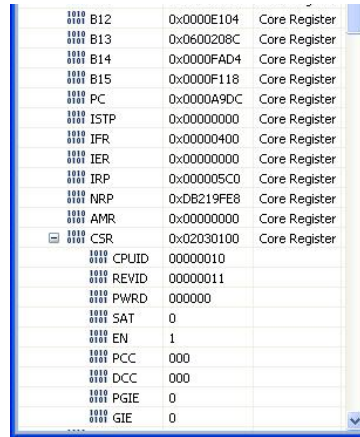
View the **content of a memory location** with *View→Memory Browser*. The **Memory Browser** opens. Here you can enter the address like „&a“ in the example in figure 1.10, click on *<Go>* and choose the data format, e.g. „Hex 32 Bit – C Style“.

*View→Registers* opens a new window to observe the **registers**. Choosing **Core Registers (internal CPU registers)**, you can monitor and change the content of the 32 A and B registers and of the „program counter“ PC. Further important registers are the interrupt registers ISTR, IFR, IER



and IRP and the interrupt flag GIE (bit no. 0 in the CSR, see Figure 1.11). Registers which content has been changed recently are highlighted in yellow. To change register content, select it with a mouse click.

Further registers can be monitored directly in this window: **Peripheral Registers** (for Timer, Interrupt and EMIF) as well as registers of the serial interfaces McBSP0 and McBSP1, controlling the CODEC.



0100 0100	B12	0x0000E104	Core Register
0100 0101	B13	0x0600208C	Core Register
0100 0102	B14	0x0000FAD4	Core Register
0100 0103	B15	0x0000F118	Core Register
0100 0104	PC	0x0000A9DC	Core Register
0100 0105	ISTP	0x00000000	Core Register
0100 0106	IFR	0x00000400	Core Register
0100 0107	IER	0x00000000	Core Register
0100 0108	IRP	0x000005C0	Core Register
0100 0109	NRP	0xDB219FE8	Core Register
0100 010A	AMR	0x00000000	Core Register
0100 010B	CSR	0x02030100	Core Register
0100 010C	CPUID	00000010	
0100 010D	REVID	00000011	
0100 010E	PWRD	00000000	
0100 010F	SAT	0	
0100 0110	EN	1	
0100 0111	PCC	000	
0100 0112	DCC	000	
0100 0113	PGIE	0	
0100 0114	GIE	0	

Figure 1.11.: View→Registers

**Hint:** With a *right* mouse click on a tab, e.g. the Register tab, and choosing **Detached** you can release the window, so you can move it freely and group it. Removing the checkmark in front of **Detached**, attaches the window again to the main CCS user interface. By drag and drop with the mouse, you can easily move a window (tab) to another location in the CCS user interface.

## 1.4. Interrupt-driven Programming for Digital Signal Processing

The interrupt system of the DSP is described in the Appendix. In our lab we use the Chip System Library CSL from TI, which offers a programming interface (APIs) for the interrupt system and a number of real-time functions for the C6xxx signal processor. You can find the CSL documentation in the SPRU401 document from TI [4].

Programs which use interrupts typically consist of:

1. Reference to include files
2. Interrupt system initialisation
3. Interrupt vector table
4. Interrupt service routine
5. Main program

**Attention:** To meet real-time conditions, it is strongly advised **not to use printf("...") statements** in an interrupt-driven program! As explained above, you can display output values with *View→Memory Browser*, entering the variable names prefixed with the address operator (&) in the

address field. Alternatively, you can watch variables in the Expressions Window, as shown above. You can view memory sections, e.g. input buffers inBuf\_L and inBuf\_R with the „CCS Graphical Display“ (see separate manual [6]).

### 1.4.1. Gateway for signals: CODEC Interface and McBSP

For input and output of analog signals, an audio codec (simply referred to as CODEC in the following) is required. To communicate with the CODEC, the DSP has an interface which allows for receiving and transmitting digitized data as a serial bitstream. These DSP interfaces, named McBSP0 and McBSP1, have to be configured and the CODEC has to be initialised. Figure 1.12 shows a block diagram of these interfaces.

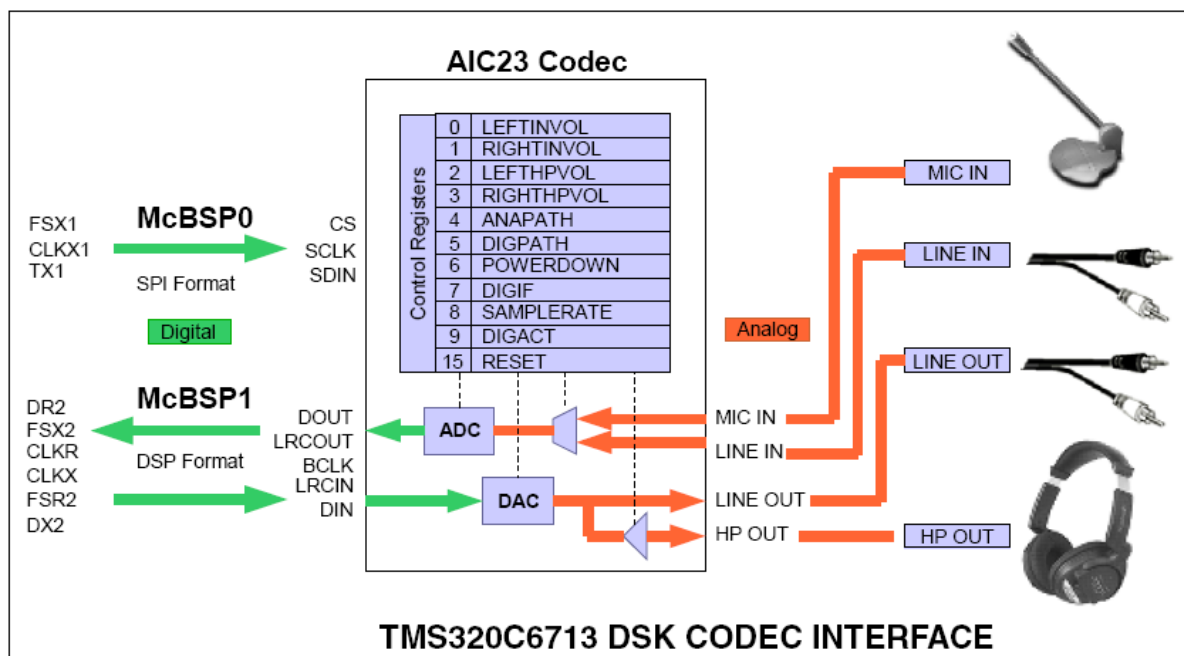


Figure 1.12.: Block Diagram of the interfaces (from TI's Techn. Reference C6713 DSK [2]).

**McBSP:** A description of the Multichannel Buffered Serial Ports McBSP0 and McBSP1 of the DSP can be taken from Appendix A.3, as well as the registers for its configuration. McBSP0 is used for Codec configuration, McBSP1 for serial data exchange, sending and receiving audio data.

**CODEC AIC23:** The 16-Bit-Stereo-Audio-CODEC, AIC23 is an onboard CODEC, connected to the DSP via two serial ports. So first both McBSP ports of the DSP have to be configured, before the CODEC can be used. A block diagram of the AIC23 can be found in the Appendix A.4.

### 1.4.2. Project get\_started with signal input and output

Replace the file *first\_project.c* in your project with the file *get\_started.c*. To do this, right-click on these two files and the file *DSK\_vectors\_AIC23.asm*, all available in the *Project Explorer* in CCS, and toggle the option *Exclude from Build* in each case. In Figure 1.13 the state before (left) and after (right) is shown.

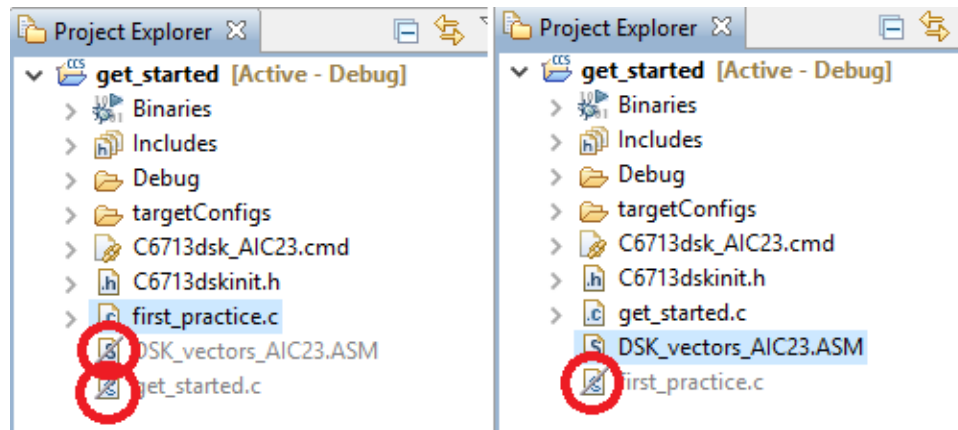


Figure 1.13.: Selection of files for the interrupt-driven program

Using this simple demo program *get\_started.c* (see Listing 1.3), you will now program the DSP reading a signal via the CODEC and output the signal directly or after modifications. The configuration of CODEC and McBSP takes place in *main()* using the *c6713dskinit.h* file present in the project directory and the *dsk6713\_aic23.h* and *dsk6713.h* system files, respectively. The listed functions in *dsk6713\_aic23.h* are implemented in the Board-System Library *dsk6713bsl.lib*, which will be added automatically while linking the project.

In the lab it's not needed to change one of these files!

The required samplerate has to be set in the variable *fs*. The following values can be set:

required samplerate	value of fs
8 kHz	DSK6713_AIC23_FREQ_8KHZ
16 kHz	DSK6713_AIC23_FREQ_16KHZ
24 kHz	DSK6713_AIC23_FREQ_24KHZ
32 kHz	DSK6713_AIC23_FREQ_32KHZ
44,1 kHz	DSK6713_AIC23_FREQ_44KHZ
48 kHz	DSK6713_AIC23_FREQ_48KHZ

The sampling rate set in *get\_started.c* is  $F_s=44,1$  kHz.

Listing 1.3: *get\_started.c*.

```
1 //-----  
2 // Digital Signal Processing Lab  
3 // Testprogram read/write  
4 //  
5 // AIC23 version  
6 //  
7 // Filename : get_started.c  
8 //
```



```

// Author Svg 8-Jan-07
10 // last modified: March 22, Kpke

12 #include "c6713dskinit.h"      //codec-DSK support file
#include "dsk6713.h"
14 #include <math.h>              //math library

16 #define LEFT 1
#define RIGHT 0
18 #define BUFLEN 1000

20 // external at DSK-Board
extern MCBSP_Handle DSK6713_AIC23_DATAHANDLE;

22
static Uint32 CODECEventId;
24 Uint32 fs=DSK6713_AIC23_FREQ_44KHZ;    //for sampling frequency
//Uint32 fs;                            //for sampling frequency

26
// two buffers for input and output samples with two counters
28 short int inBuf_L[BUFLEN];
short int inBuf_R[BUFLEN];
30 short int count_INT=0;

32 union {
    Uint32 both;
34     short channel[2];
} AIC23_data;

36
interrupt void intser_McBSP1()
38 {
    AIC23_data.both = MCBSP_read(DSK6713_AIC23_DATAHANDLE); //input data

40
    // buffer monitoring input signal, reset count if BUFLEN is reached,
42 // then input buffer is full
    inBuf_L[count_INT] = AIC23_data.channel[LEFT];
44     inBuf_R[count_INT] = AIC23_data.channel[RIGHT];

46 // buffer full ??
    count_INT++;
48     if (count_INT >= BUFLEN)
        count_INT = 0;

50
    MCBSP_write(DSK6713_AIC23_DATAHANDLE, AIC23_data.both); //output 32 bit
        data, LEFT and RIGHT

52
    return;
54 }
////////////////////////////////////

56
void main()
58 {
    IRQ_globalDisable();                //disable interrupts

60
    DSK6713_init();                      //call BSL to init DSK-EMIF,PLL)

62
    hAIC23_handle=DSK6713_AIC23_openCodec(0, &config); //handle(pointer) to
        codec
64     DSK6713_AIC23_setFreq(hAIC23_handle, fs); //set sample rate

66     MCBSP_config(DSK6713_AIC23_DATAHANDLE,&AIC23CfgData); //interface 32 bits to
        AIC23

```

```

68  MCBSP_start(DSK6713_AIC23_DATAHANDLE, MCBSP_XMIT_START | MCBSP_RCV_START |
    MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220); //start data channel again
70
72  CODECEventId= MCBSP_getXmtEventId(DSK6713_AIC23_DATAHANDLE); //McBSP1 Xmit
74  IRQ_map(CODECEventId, 5);           //map McBSP1 Xmit to INT5
76  IRQ_reset(CODECEventId);           //reset codec INT5
78  IRQ_globalEnable();                //globally enable interrupts
76  IRQ_nmiEnable();                   //enable NMI interrupt
78  IRQ_enable(CODECEventId);          //enable CODEC eventXmit INT5
78  IRQ_set(CODECEventId);             //manually start the first interrupt
80  while(1);                          //infinite loop
}

```

### 1.4.3. First test of the project

The demo program `get_started.c` reads input data from the CODEC and outputs these data to the CODEC again, without any changes: The function `MCBSP_read` reads 32 bit integer from the ADC, so both channels together. These are buffered in an union, as `AIC23_data.both`. An access to each single channel is possible by `AIC23_data.channel[ ]`. `AIC23_data.channel[1]` contains the left audio channel data and the data from the right audio channel are available in `AIC23_data.channel[0]`. Correspondingly, function `MCBSP_write` outputs 32 bit integer to the DAC of the Codec.

#### Lab task 1: Understanding the audio input / output operation

In this first task, you will apply a sinusoidal voltage to the CODEC and use the program `get_started.c` to read this signal into the DSP and output the signal unaltered. You will use the union and understand how a bitmask works. -

- Please use the HAMEG HMF2525 function generator to apply the sinusoidal voltage to the DSK 6713 board. Mind that you have to **terminate the input cable from the function generator with a 50  $\Omega$  resistor** to avoid overvoltages that might destroy electrically the CODEC input.
- Feed a sine wave from the function generator with  $V_{pp} = 1\text{ V}$  into both audio channels at the input of the board one after the other and connect an oscilloscope to both output channels. Output should be nearly equal to input signal for each channel.
- Mask the right channel (set to zero the lower 16 bit) by inserting the following row between reading and writing data:  
`AIC23_data.both = AIC23_data.both & 0xffff0000;`
- Call *Run*→*Debug* and check the program: The right channel should be “quiet” now.
- Comment out the mask after this task.
- Insert the following line before outputting the data:  
`AIC23_data.channel[RIGHT] = AIC23_data.channel[LEFT];`

- The left channel data is then copied to the right channel and sent to the DAC. Call *Run→Debug* and check the program again.
- Comment out this line as well.

#### 1.4.4. Left-/Right Switching of CODEC Channels

For the following task, feed an audio file into the input of the DSK board.

Audio files can be found in M:\DV\_LAB\wavefiles. Please play them back with Audacity on the PC.

The output of the sound card of the PC (headphone audio jack) must be connected to the input of the board via a audio cable.

**Attention:** Never feed input signals from different sources at the same time!

Both output channels can now be connected to the scope via BNC cables and displayed on it. Here you can also connect an audio cable to the loudspeakers at the same time.

##### Lab task 2: Switching of the channels in real-time

Modify the interrupt service routine (ISR) to switch channels in real-time. The selection of the channel has to be achieved with a new variable *short int select\_channels*, which is displayed and changed in the CCS Expressions window. For the changing, the program has to be suspended.

The content of *select\_channels* should have the following effect:

- *select\_channels* = 0 ⇒ left output only
- *select\_channels* = 1 ⇒ right output only
- *select\_channels* = 2 ⇒ play both channels, this time interchanged
- all other values ⇒ output as input

Verify the functionality of your program with the following audio file as input:  
*500\_1200Hz.wav*

#### 1.4.5. Quantization

Based on the *get.started.c* program it should be determined how many bits are needed to produce understandable speech. Use for instance *THEFORCE.wav* as signal input. Do the quantisation by masking the lower order bits of both channels in the interrupt routine.

Use a variable as mask, so that you can change the value while the program is running in the Expressions window. (Choose presentation in hex via right mouse click on the variables type.)

**Hint:** To obtain meaningful results, also the most significant bits of the sampled audio signal should change correspondingly. Begin this study (with all 16 bits) at an input level high enough to cause the output to nearly saturate on peaks.

### Lab task 3: Quantization

Quantize each of the signals of both channels with the prepared bitmasks. Give a subjective evaluation for quantization with 16, 4 and 1 bit respectively:

- How understandable is the speech?
- Can the speaker be recognized?
- How intense is the noise?
- What is your overall conclusion?

### 1.4.6. ADC- /DAC-Overflow

**Hint:** At the beginning of this task, you will ideally receive direct instruction in the audio analyzer UPV. If this is not convenient for organizational reasons, follow the instructions for the alternative measurement without UPV.

Add a factor (variable with start value: 1) to the `get_started.c` program to increase the output amplitude of the right channel internally.

Feed the right channel with a sine wave of 300 Hz from the UPV audioanalyser.

**Attention:** Max.  $4 V_{pp}$  at the DSK board input!

The voltage at the R&S UPV is set as RMS value (effective)!

$$U_{eff} = U_{pp}/2/\sqrt{2} \dots \Rightarrow \text{Max. } 1,4 V_{eff}$$

### Lab task 4: Overflow

#### ADC overflow:

- Increase the input voltage starting at  $U_{eff} = 0.2 V$  (for a measurement with function generator select  $U_{pp} = 0.5V$ ).
- Determine the maximum input voltage for the CODEC ADC: Explain your decision on the basis of a screenshot from the oscilloscope or from the waveform window (UPV) and a FFT measurement with the UPV (use `setfile FFT_up-to-24kHz.set`, set a adequate voltage (in RMS!) and frequency in the Generator Function window).
- How does the spectrum change when you further increase the input voltage (in respect of the maximum allowed value)?
- *Alternative measurement without UPV: Use the CCS graphical display (in frequency domain), a short manual is provided in the bibliography under 6. You have to add a buffer for the output values.*

#### DAC overflow:

- Drive the input with  $U_{eff} = 0.1 \text{ V}$  (function gen.  $U_{pp} = 0.2 \text{ V}$ ).
- Add the implemented factor to the Expressions window.
- Increase the **DAC** input in single steps using this factor until you notice an overdrive.
- What is the corresponding gain factor and voltage?
- Show a screenshot of the spectrum (FFT) for a slight and for a heavy overdrive of the DAC.

Describe your observations:

- In what way are the ADC and DAC overflows different?

### 1.4.7. Aliasing Effects

Violating the sampling theorem, aliasing occurs. These are to be prevented by anti-alias filters.

#### Lab task 5: Aliasing

Input a sine signal to the board.

- Adjust the frequency sequentially so that (a) no aliasing occurs and (b) an explicit aliasing can be observed.
- Why did you choose these signal frequencies for (a) and (b)?
- Show the output signals using oscillogram or waveform window (UPV), as well as the screenshots of the output spectra (UPV) and explain your observations.

### 1.4.8. Audio Reverberation System

The final task is to implement a reverberation system.

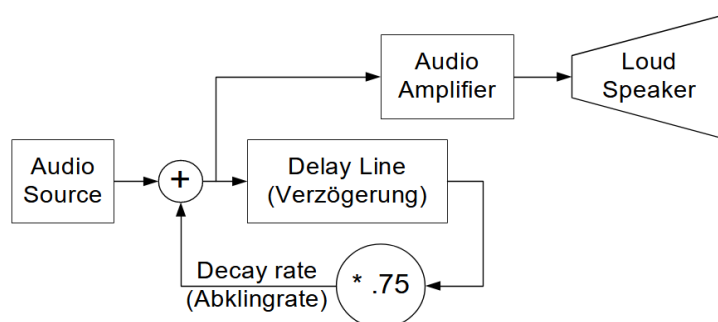


Figure 1.14.: Reverberation System

### Lab task 6: Reverberation System

In your *get\_started.c* program, implement the audio echo system from Figure 1.14 with the following parameters and following the detailed instructions below:

- sampling rate of 8 kHz
- 0.5 seconds delay
- decay rate of 0.75

Test your program.

It might be necessary to decrease the amplitude of the PC audio output, if overflows occur during the summation.

Use the file *CLAP.wav* to test the functionality of the program. An impulse can be heard in the speakers right away. After 0.5 seconds you can hear an impulse reduced to 75 % as the first echo. The second echo follows after another 0.5 seconds, itself reduced to 75 % of the first echo, and so on.

#### Hints:

Use a globally defined one-dimensional integer array *delay[ ]* as digital “circular delay buffer”, which can be accessed from the Interrupt Routine.

Set the sampling rate to 8 kHz. You will need a buffer length of  $N = 0.5 / (1/8000) = 4000$ .

Use a global (!!) index variable called *index*.

To avoid the appearance of noise, all elements of *delay[ ]* must be set to zero **before the enabling of the interrupts**, so *delay[ ]* must be explicitly cleared within a loop! (Initializing to zero when declared has no effect.)

In the interrupt routine, the output of the delay line is read at position “index” in the delay array.  
**delay\_line\_output = delay[index];**

Now the calculated delay line input value (according to Figure 1.14, this is the current input sample + 0.75 \* Delay\_line\_output) is written at this position *delay[index]*. To continue working with integers, we can no longer multiply by 0.75, instead we must deduct one quarter from *delay\_line\_output*, ending up with the original value shifted twice to the right. Pay attention to the brackets, since the shift operator has a lower priority than plus/minus.

**delay[index] = present\_input\_sample + delay\_line\_output - (delay\_line\_output >> 2);**

At the end of the ISR, *index* is incremented so it points to the next oldest value in *delay[ ]*; this is output at the next sample-interrupt. *Index* must be set back to 0 when the end of the array has been reached (*index = N*).



## Radix-2 FFT and Real-Time Spectrum Analyser

### 2.1. Objectives of this lab second session

In this lab, you will implement a 64-point Radix-2 FFT on the TMS320C67xx floating-point signal processor based on a given 8-point FFT. Eventually, you will develop a real-time spectrum analyzer using this FFT implementation. After this lab you should

- better understand the Radix-2 FFT algorithm,
- be able to understand how to implement and execute an FFT on a DSP under real-time constraints,
- be able to implement a framework around an existing FFT algorithms in assembly language in order to perform a frequency analysis of a signal.
- be able to apply a Hamming window to a block of N samples stored in a corresponding buffer

### 2.2. Preparation of the lab

Prepare well the fundamentals presented in the lecture on DFT and FFT and the preparation tasks in this lab assignment.

#### Prep task (for short test)

Familiarize yourself with the concepts of

- Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT), including
- DFT theorems,
- DFT symmetries, and

- effects of windowing.

These topics will be addressed by the short test at the beginning of the lab session.

### 2.2.1. Analysis of a Butterfly

In Prep Task 1, we analyze the butterfly of the 2-point FFT which is depicted in Figure 2.1.

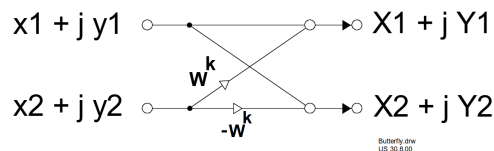


Figure 2.1.: Butterfly

#### Prep task 1

- The relation between the (generally complex) time-domain values

$$z_1 = x_1 + jy_1 \quad \text{and} \quad z_2 = x_2 + jy_2$$

on the left side of Figure 2.1 and the corresponding values

$$Z_1 = X_1 + jY_1 \quad \text{and} \quad Z_2 = X_2 + jY_2$$

of the DFT spectrum on the right side shall be found. Before doing so, please mind:

- Four equations are wanted: two for the real-parts  $X_1, X_2$  and two for the imaginary-parts  $Y_1, Y_2$ .
- The twiddle factor is given by  $w^k = e^{-j2\pi k/N}$  and the DFT length is  $N = 2$ . What is the value of  $k$  needed here? Determine the value(s) of the twiddle factor(s).
- Give now the four equations for  $X_1, Y_1, X_2, Y_2$ .
- Rewrite the equations for  $X_2, Y_2$  using only  $x_1, X_1, y_1, Y_1$

### 2.2.2. 8-point FFT (DIT)

An 8-point FFT (DIT) is illustrated in Figure 2.2. Analyse this signal-flow diagram by solving the prep tasks.

The input sequences  $x_1(n)$ ,  $x_2(n)$  (not  $x_{in}(n)$  !!) consist each of the following **8 real decimal** values, which we assume to be stored as 16 Bit (short int):



$$x_1[n] = \{500, 0, -500, 0, 500, 0, -500, 0\}, N = 0, \dots, 7$$

$$x_2[n] = \{15000, 0, -15000, 0, 15000, 0, -15000, 0\}, N = 0, \dots, 7$$

### Prep task 2

1. Put the values of  $x_1[n]$  in the correct order according to Figure 2.2. Calculate (e.g. by hand) the output values of the first, second and last stage according to Figure 2.2 and assign the values to the nodes in the graph.
2. Write a MATLAB script `FFT_a.m` which calculates the output signal  $X_8[k], k = 0, \dots, 7$  directly (i.e. internal node values not required) using MATLAB's FFT function. Compare your results from above with the result of MATLAB.
3. Do overflows occur?
4. Now repeat the handwritten calculation of the output values of all three stages for  $x_2[n]$ .
5. Extend your script `FFT_a.m` to calculate the FFT of  $x_2[n]$  and again compare your calculation with the one from MATLAB.
6. Do overflows occur (values larger than can be represented with signed 16 bit)? If so, explain why!
7. By which factor do we need to scale the input values  $x[n]$  that never an overflow can occur at the output of the 8-point FFT when all values are of type short int?
8. Find a method that has a smaller loss in precision as the previous one. Hint: consider a scaling of values at nodes *inside* the FFT algorithm. Explain e.g. with an example why the latter method outperforms method where we scale the input values only?

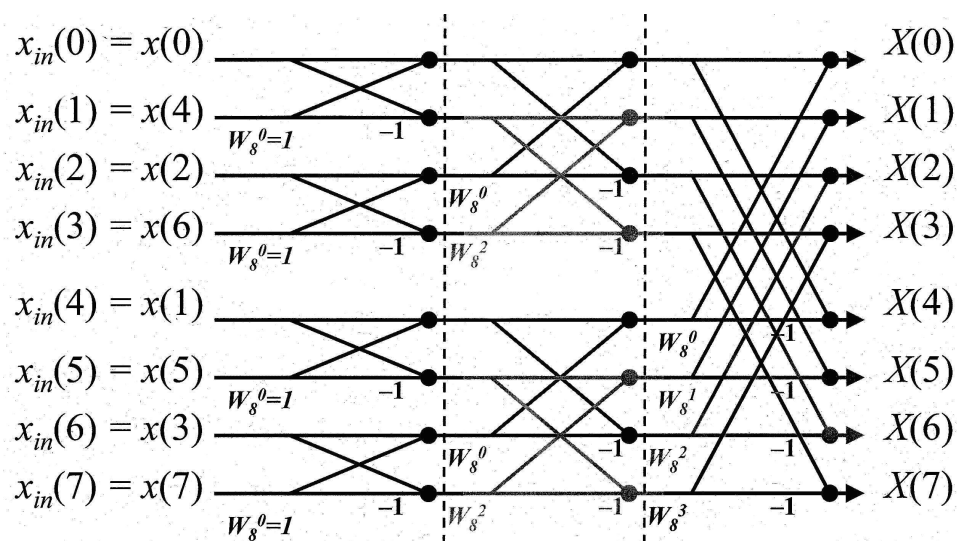


Figure 2.2.: 8-point FFT (3 stages)

**Hint:** Begin each MATLAB script with “clear all”. This clears the internal Workspace and if necessary resets ‘i’ and ‘j’ (previously defined as index variables) back to imaginary numbers, i.e.  $i^2 = -1$ ,  $j^2 = -1$ .

**Complex-valued input signal:**  $x_3[n]$  is a complex-valued test signal with frequency 3 and amplitude 0.125. In MATLAB notation:

$$x_3 = 0.25 \cdot \cos(2\pi \cdot 2 \cdot (0:7)/8) + j \cdot 0.25 \cdot \sin(2\pi \cdot 2 \cdot (0:7)/8)$$

### Prep task 3

Extend your MATLAB script as follows:

- Plot the magnitude spectrum  $|X[k]|$  of  $x_3[n]$ . Label the frequency axis  $k$  correctly.
- Does the magnitude spectrum show symmetries? Explain your answer.

### 2.2.3. Familiarize yourself with the lab project

In D:\ti.work5 or in EMIL, you find a complete TI-DSP sample project for calculating an 8-point FFT. In `main()`, the FFT is calculated once before entering the infinite `while(1)`-loop. The program provides already an interrupt routine which however just realizes a simple echo program (output left, right = input left, right), i. e., the FFT routine is not called again.

Please make sure that you understand the program files of the project, particularly...

- how the input signal is generated,
- how twiddle factors are calculated and how they are arranged in bit-reversed order,
- how the FFT function is called including of bit-reversal of the samples in the FFT buffer in `main()` once.

The names of the files containing the FFT calculation are **butterfly.c** and **radix2.c**. The function call in the C code is:

```
// carry out the N-point FFT on array x[2*N] IN PLACE
radix2(N_FFT, x, w_r, w_i);
```

- This algorithm expects the (real and imaginary) samples in  $x[2 * N_{FFT}]$  in bit-reversed order, while the coefficients  $w[N_{FFT}]$  have to be stored in normal order.
- The real part of the twiddle factors is stored on even addresses of the buffer  $w[N_{FFT}]$ , the imaginary samples on the odd addresses.

- A block of  $N_{FFT}$  samples of the real-valued part of the input signal  $inbuf[ ]$  is stored bit reversed on even addresses of the FFT buffer  $x[ ]$ . The imaginary parts on the odd addresses are set to zero, since for a real-valued signal the imaginary part is necessarily equal to zero.
- Optional: A Hamming window shall be applied to the samples stored in  $inbuf[ ]$ . A variable `do_Hamming` shall be used to turn the window on or off.
- After execution of the FFT, the FFT result is stored in the  $x[2*N_{FFT}]$  buffer. The calculation is done “in-place”, i.e., the same memory is used for FFT input and output data.
- An ANSI C function `short bitrev(short in, short stages)` for bit-reversal is also provided. The second parameter of this function is referring to the number of FFT stages, not to the FFT length.

## 2.3. Lab: Spectrum Analysis using FFT

### 2.3.1. Getting started with the c project

The given program correctly calculates the Radix-2 8-point FFT for an input sequence. If necessary, adjust the input values to the already examined input sequence:

$$x_1(n) = \{500 \quad 0 \quad -500 \quad 0 \quad 500 \quad 0 \quad -500 \quad 0\}$$

Load the program into CCS and check this first.

#### Lab task 1

1. Enter the input sequence  $x_2(n)$  from prep task and check the result. Do overflows occur? Comment on this and explain the values that occur.
2. Correct the “error” just determined in the program `butterfly.c`, so that overflows are avoided. Check the functionality.
3. Why were the cosine values implemented with negative sign in the program?  
*Hint:* think of the representable number range with two’s complement representation.
4. In `butterfly.c` replace the equations for X2 and Y2 with the equations from the first preparation task. Check that the results remain identical.

### 2.3.2. Extension of the FFT to 64 points

Your project should now be extended to a 64-point FFT.

First make a copy of the file `FFT8.Radix2.c` in the project folder and rename it to `FFT64.Radix2.c`. After that deactivate `FFT8.Radix2.c` via *Exclude from Build*.

In the program you have to adjust not only the FFT length but also the number of stages (in `radix2.c`).

### Lab task 2: 64 point FFT

- Test the 64-point FFT with the following signal written directly to *inbuf[ ]* and compare the result with that from MATLAB.  
 $x4 = 16384 * \sin(2 * \pi * 5 * (0 : 63)/64);$
- Use the graphical display in CCS via *Tools*→*Graph* (separate instructions in the bibliography under 6) to plot the result against a MATLAB plot.

### 2.3.3. Real-time spectrum analyser

A continuous FFT analysis of N samples of a real signal is to be performed. The input signal is a sine signal coming from a function generator, the output is displayed on an oscilloscope. The sampling frequency is 8 kHz.

In the project folder, make a copy of the file *FFT64\_Radix2.c* again and rename it to *FFT64\_Analyser.c*. Then disable *FFT64\_Radix2.c* via *Exclude from Build*.

The algorithm is to be implemented as follows:

#### 1. Reading samples

The steps listed here are to be implemented in the ISR.

- The samples from the ADC are stored in a *short int* input buffer *in\_buf[N]* in the interrupt routine. The 0th sample value is saved in *in\_buf[0]*, the 1st in *in\_buf[1]* and so on.
- A global counter variable *samplecount* holds the number of samples already read from the A/D converter.
- After reading N samples (during N interrupts) the input buffer is filled. If (*samplecount*  $\geq$  N),
  - a) the interrupt is disabled (via *IRQ\_disable(CODECEventId);*)
  - b) *samplecount* is reset

#### 2. Calculation of the magnitudes of the spectrum

The whole calculation is done in the *while(1)* loop in *main()* and starts after reading N samples and disabling the interrupts. I.e. the following steps are only executed if (*samplecount* == 0).

- First each element of the input buffer *in\_buf[N]* is copied (bit reversed) to *x[2 \* N]*, but only to those array elements with even numbered indexes. All array elements with odd index (imaginary parts) have to be explicitly set to zero after calculating a 64-point FFT, since after the calculation *x[2 \* N]* is complex!!
- Function *radix2( )* is called and computes the FFT of the last N read samples, stored in *x[2 \* N]*.

Before calculating the FFT,  $x[2 * N]$  contains the values for the FFT (*short int*); after the FFT, it contains the (complex) values of the spectrum.

- After that, the magnitudes of the spectrum are calculated from  $x[2 * N]$  and saved in the output buffer  $out\_buf[N]$ .  $out\_buf[N]$  now contains the 32 Bit *int* results of the last read samples!
- The interrupt is enabled.

### 3. Output of results

The output of the values and the trigger pulse to the DAC is of course also done in the ISR.

- During each cycle, the interrupt routine sends one sample from  $out\_buf[ ]$  to the left channel of the D/A converter. So while reading N new samples, the result consisting of N squared magnitudes of the computed FFT is sent to the DAC.

**Hint:** To save time of taking the square roots in the calculation of the magnitudes, it is sufficient to send the *squares* of the magnitudes of the spectrum, i.e.  $|X_k|^2$  instead of  $X_k$  to the DAC.

- Breakpoint settings for continuous update:  
Right mouse click on the breakpoint, then *Breakpoint Properties* opens a new window. You can choose as **Action**, whether the program should stop (*Remain Halted*) or e.g. run to the breakpoint again and again and refresh the windows choosing *Update View* or *Refresh All Windows*.
- Trigger for the presentation on the scope:  
Furthermore, if  $(samplecount \leq 2)$ , a trigger impulse 32767 is sent to the right channel of the DAC; otherwise the output is '0'.

### Lab task 3: Real-time spectrum analyser

Implement the analyzer according to the description of the algorithm above.

Verify that the FFT64\_Analyser.c functions correctly:

Connect the signal generator to the DSK board and select "Waveform Sinus". Modify the frequency between 100 Hz and 4 kHz in steps of 100 Hz (amplitude:  $2 V_{pp}$ ).

1. Use the CCS 'graphical display' to monitor the results of the FFT. Set a breakpoint at the line where samplecount is set to zero. Start the program, updating the "graph display" at the breakpoint.  
Take a screenshot for  $f_{in} = 2 \text{ kHz}$ .
2. In a next step, display the result on the oscilloscope (connect the right DSK output channel to the 2nd channel of the scope and use this as triggersource). Take Screenshots of the scope for  $f_{in} = 1 \text{ kHz}$  and  $f_{in} = 3 \text{ kHz}$   
*Hint:* You'll see an inverted signal on the scope because of the inverting DAC.
3. *Optional:* Compute in MATLAB a 64-point Hamming-window and scale it to a *short int* variable  $hamm\_wind[64]$ . Multiply  $in\_buf[ ]$  with this window before the buffer  $in\_buf[N]$  is copied to  $x[2 * N]$ . Create a variable *do\_hamming* to switch the windowing on and off.

Connect a sine signal of amplitude of  $2 V_{pp}$  and frequency 500 Hz to the input of the DSK board. Display the output buffer in the CCS „graph display“. Set a breakpoint within the *while(1)*-loop, at the line where samplecount is set to zero. Start the program, updating the “graph display” at the breakpoint. Display the variable *do\_hamming* in the CCS “Expressions Window” and switch *do\_hamming* on and off. Comment on the effect of the Hamming-window on the FFT output in *out\_buf[ ]* (magnitude spectrum displayed logarithmically in a CCS ”graph display\”)



## Appendix: Hardware DSK6713

### A.1. The DSK6713 Board

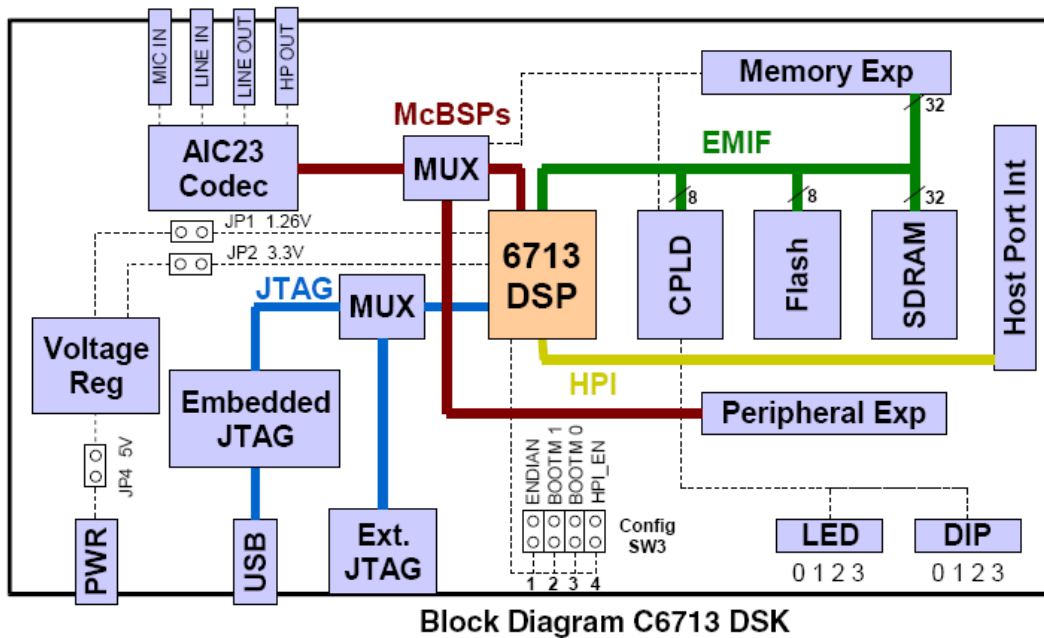


Figure A.1.: Block diagram of the C6713 DSK

### A.2. The Interrupt System of the C6000 DSPs

The C67xx DSPs have 16 interrupt sources each. However the CPU is able to handle only 12 interrupts at a time. These are INT4 ... INT15. To handle all 16 interrupts, an interrupt-multiplexer is used to map the external interrupts (see table A.1) to the internal ones INT4 ... INT15. The mapping is obtained with two memory DSP registers:

MUXL = 0x019C0004 and

MUXH = 0x019C0000.

These registers are programmed using the CSL-Routine **IRQ\_map**. In case of an interrupt, the processor jumps to a specific location in the interrupt vector table, which is normally located in the address range 0x0000 ... 0x0200. Using the ISTP-register (Interrupt Service Table Pointer) you can map the vector table to a different address range, but this feature will not be used in our lab.

CSL name	Description
IRQ_EVT_DSPINT	Host processor to DSP interrupt
IRQ_EVT_TINT0	Timer-0 interrupt
IRQ_EVT_TINT1	Timer-1 interrupt
IRQ_EVT_SDINT	EMIF SDRAM timer interrupt
IRQ_EVT_EXTINT4	External interrupt pin 4
IRQ_EVT_EXTINT5	External interrupt pin 5
IRQ_EVT_EXTINT6	External interrupt pin 6
IRQ_EVT_EXTINT7	External interrupt pin 7
IRQ_EVT_DMAINT0	DMA channel 0 interrupt
IRQ_EVT_DMAINT1	DMA channel 1 interrupt
IRQ_EVT_DMAINT2	DMA channel 2 interrupt
IRQ_EVT_DMAINT3	DMA channel 3 interrupt
IRQ_EVT_XINT0	McBSP0 transmit interrupt
IRQ_EVT_RINT0	McBSP0 receive interrupt
IRQ_EVT_XINT1	McBSP1 transmit interrupt
IRQ_EVT_RINT1	McBSP1 receive interrupt

Table A.1.: List of CSL naming for the interrupts

The correspondence between internal interrupts and the vector addresses is displayed in Table A.2. Note that the priority of the interrupts decreases from INT4 to INT15. In Table A.2 you can see that the Reset-vector has the address 0x0000, and the only interrupt of the C6000 which cannot be masked (NMI) has the vector address 0x020 (addresses 0x40 and 0x60 are reserved for later use). In the lab, we recommend the use of assembler code for the interrupt vector table. An example is given in DSK\_vectors\_AIC23.asm (see listing A.1), which you can find in D:\ti\_work5\DSK6713\Lab.support\_DSK6713. The assembler code must contain jumps to the RESET-Routine of C-Runtime-Systems `_c_int00` (defined in CCS) and to the Interrupt-Service-Routine (ISR). In the vector table, the names of the ISR are prefixed with an underscore „\_' (e.g. „`_intser_McBSP1`”). All the names must be listed in an assembler file,\*.ref’. The linker saves the code in the compiler section „vectors“, located at the address 0x0000 (see C6713dsk\_AIC23.cmd Link-File in D:\ti\_work5\DSK6713\Lab.support\_DSK6713).

Listing A.1: DSK\_vectors\_AIC23.asm contains the assembler code of the vector table (incomplete list!).

---

```

1      .sect ".vectors"
2      .ref  _c_int00
      .ref  _intser_McBSP1 ;
4 SP .set  B15 ; rts6700.lib uses B15 as stack pointer
RESET_V: ;
```



```

6     MVKL    _c_int00, B3
      MVKH    _c_int00, B3
8     B    B3
      MVC     PCE1, B0;
10    MVC     B0, ISTD;
      NOP
12    NOP
      NOP
14 NMI_RST: .loop 8
      NOP    ; fill with 8 NOPS
16      .endloop
      RESV1:  .loop 8
18      NOP
      .endloop
20 RESV2:  .loop 8
      NOP
22      .endloop
      INT4_V:  .loop 8
24      NOP
      .endloop
26 INT5_V:  STW    B0,*SP--[2] ; save B0 temp.
      || MVKL    _intser_McBSP1, B0
28      MVKH    _intser_McBSP1, B0
      B    B0    ; execute ISR
30      LDW     *++SP[2],B0 ; restore B0
      NOP     2
32      NOP
      NOP
34 ...

```

---

Note that between any interrupt vector pairs there is an interval of 0x20 in memory (one fetch each), which results in 8 NOPs for every unused vector. When a vector is used by a program, this is done with five statements: First B0 is saved temporarily to the stack. Two statements load the 32-bit-address of the ISR in parallel in the B0 register, and the fourth is the jump to the content of this register. Lastly, B0 is restored from the stack. This means that the gap to the next vector must also be filled with 4 NOPs. Register B15 is from TI's C-compiler always used as a stackpointer (SP). In order to execute one of the INT4...INT15 interrupts, the following additional requirements must be set:

- The NMI-Flag must be set using the CSL routine **IRQ\_nmiEnable**.
- The GIE-Flag (Global Interrupt Enable) must be set using the routine **IRQ\_globalEnable**.
- The individual interrupt(s) must be enabled using the routine **IRQ\_enable( <Int.Source> )**, which sets the corresponding bit in „Interrupt Enable Register“ **IER** (see Table A.2).
- Using **IRQ\_disable( <Int.Source> )** the interrupts can also be temporarily disabled.
- A branch to the ISR must exist at the appropriate vector table address.

When an interrupt is executed, the C6000 Interrupt-System clears the GIE-flag automatically. Also, when the DSP returns from the ISR, the flag is set again. (Actually it is copied to the PGIE-flag before it is cleared.)

To set up nested interrupts, the GIE-flag must be individually set in the ISR.

When the CPU recognises a hardware interrupt, the corresponding bit is set in the Interrupt-Flag-

Int. Vect. No	Bit pos. on Interrupt registers	Vector Address
Reset	0 (LSB)	0x000
NMI	1	0x020
reserved	2	0x040
reserved	3	0x060
4	4	0x080
5	5	0x0A0
6	6	0x0C0
7	7	0x0E0
8	8	0x100
9	9	0x120
10	10	0x140
11	11	0x160
12	12	0x180
13	13	0x1A0
14	14	0x1C0
15	15 (MSB)	0x1E0

Table A.2.: List of the interrupt vector numbers and vector addresses

Register **IFR**. If problems occur in interrupt-driven programs these flags should be checked. The status of the flags can be checked in CCS via *View→Registers→Core Registers*

Changes in the IFR should be made using those ISR (Interrupt Set Register) or ICR (Interrupt Clear Register) which are not displayed in CCS. All attempts to modify the IFR directly from CCS are ignored by the hardware!

Instead of accessing directly the IFR, the interrupts can be forced or cleared for instance by calling the CSL-routines **IRQ\_set** (<Int.Source>) or **IRQ\_clear** (<Int.Source>). These routines write directly to the ISR or ICR register.

### A.2.1. Configuration of the interrupt system

Listing A.2: Configuration of codec interrupts

```

1 ...
2 IRQ_map(CODECEventId, 5); //map McBSP1 Xmit to INT5
  IRQ_reset(CODECEventId); //reset codec INT5
4 IRQ_globalEnable();      //globally enable interrupts
  IRQ_nmiEnable();         //enable NMI interrupt
6 IRQ_enable(CODECEventId); //enable CODEC eventXmit INT5
  IRQ_set(CODECEventId);   //manually start first interrupt
8 ...

```

The routines for the interrupt system are provided by the interrupt Chip-System-Library (CSL). The declarations and prototypes can be found in the header file `csl_irq.h`:

- `IRQ_map()`: Mapping of the interrupt sources to the interrupt vectors. In this example, the McBSP1-Interrupt “CODECEventId” is mapped to the interrupt vector 5 at vector address 0xA0 (see `DSK_vectors_AIC23.ASM`).
- `IRQ_reset()`: Clears possibly pending interrupts.
- `IRQ_globalEnable()`: Sets the “global interrupt enable” flag (GIE), required for the execution of all interrupts.
- `IRQ_nmiEnable()`: Enables interrupt NMI (which cannot be masked), required for the execution of interrupts INT4..INT15 (which can be masked).
- `IRQ_enable()`: Enables the individually used interrupt with the “interrupt enable register” IER. In our example this is bit 5.
- `IRQ_set()`: Finally the interrupt sequence has to be started. This is realized forcing a software interrupt.

### A.3. Multichannel Buffered Serial Ports McBSP

A “DXR interrupt” is generated after each 32-bit word is clocked into the McBSP “Data Transmit Register” (DXR) over its DX line. In response to this “serial data transmitted” interrupt, the DSP chip executes an interrupt service routine. The routine calculates the new output values from the received input values, as well as from the previous input and output values saved in memory. The new digital output values are sent to the CODEC through the serial data output line of the McBSP (which is connected to the serial data input SDI line of the CODEC).

#### A.3.1. McBSP Pins and Registers

Both buffered multichannel serial ports (McBSP0 and McBSP1) consist of a data path and a control path which connect to external boards. Separate incoming and outgoing pins are provided for data transmission. Control information (clocking and frame synchronisation) is communicated optionally via five other pins. The connected device communicates with the McBSP through three 32-bit wide control register, accessible via an equally wide peripheral bus. This is shown in figure [A.2](#). Connection to the data and control paths can be established through the seven pins listed in Table [A.3](#). The McBSP contains 10 memory supported registers (Table [A.4](#)) which control its function.

Serial data is read via the DR-pin and saved in the 32-bit DRR register; 32-bit outgoing data is written in the DXR register and sent through the serial DX-pin. After the content of the Data-Transmit-Registers (DXR) is written in the internal Data-Transmit-Shift-Register, the XRDY-bit (bit 17) is set in the SPCR-register. This bit indicates that new data can be written, and can be used as interrupt signal or for polling purposes. Correspondingly, the RRDY-bit (bit 1) indicates that the DRR register has been written from an external source.

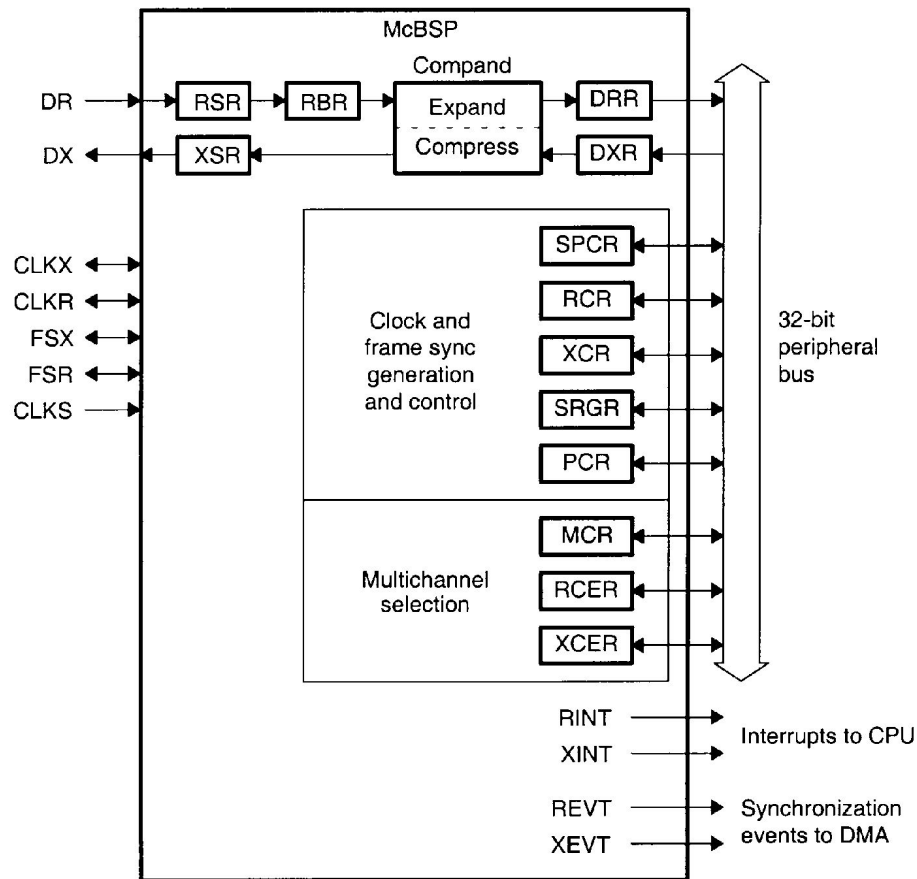


Figure A.2.: Block diagram of a McBSP of the C6x-DSP (SPRU190C).

### A.3.2. Configuration of the McBSP

The McBSP routines for the interrupt system are provided by the Chip-System-Library (CSL). The declarations and prototypes can be found in the header file `cs1_mcbasp.h`: Listing A.3 shows the example for CODEC activation via McBSP1 (from `get_started.c`) used in the lab:

- `DSK6713_init()` initialises the DSK board.
- `DSK6713_AIC23_openCodec()` from the DSK6713-Board System Library (BSL) creates a pointer to a configuration object (handle). This function opens both McBSP ports and configures the Codec via McBSP1. `DSK6713_AIC23_DATAHANDLE` is the handle for the McBSP1 port data, in the application it has to be declared as *external*.
- `DSK6713setFreq()` sets the Codec samplerate.
- `MCBSP_config` configures McBSP1 (the configuring structure `AIC23CfgData` is declared in `C6713dskinit.h`).
- McBSP1 has to be started.

Pin	Direction	Description
CLKR	I/O/Z	Receive clock
CLKX	I/O/Z	Receive clock
CLKS	I	Receive clock
DR	I	Receive clock
DX	O/Z	Transmitted serial data
FSR	I/O/Z	Receive frame synchronisation
FSX	I/O/Z	Transmit frame synchronisation

Table A.3.: McBSP hardware pins to external devices.

Register	Address	Description
DRR	0x18C0000	Data Receive Register
DXR	0x18C0004	Data Transmit Register
SPCR	0x18C0008	Serial Port Control Register
RCR	0x18C000C	Receive Control Register
XCR	0x18C0010	Transmit Control Register
SRGR	0x18C0014	Receive frame synchronisation
MCR	0x18C0018	Multichannel Register
RCER	0x18C001C	Receive Channel Enable Reg.
XCER	0x18C0020	Transmit Channel Enable Reg.
PCR	0x18C0024	Pin Control Register

Table A.4.: Memory addresses and description of the McBSP0 registers.

- A McBSP1 transmit event determines the variable CODECEventId. The configuration structure of McBSP1 defines, that the interrupt occurs after the transmission of a frame (FSX pin). This event variable is used in the interrupt routines later.

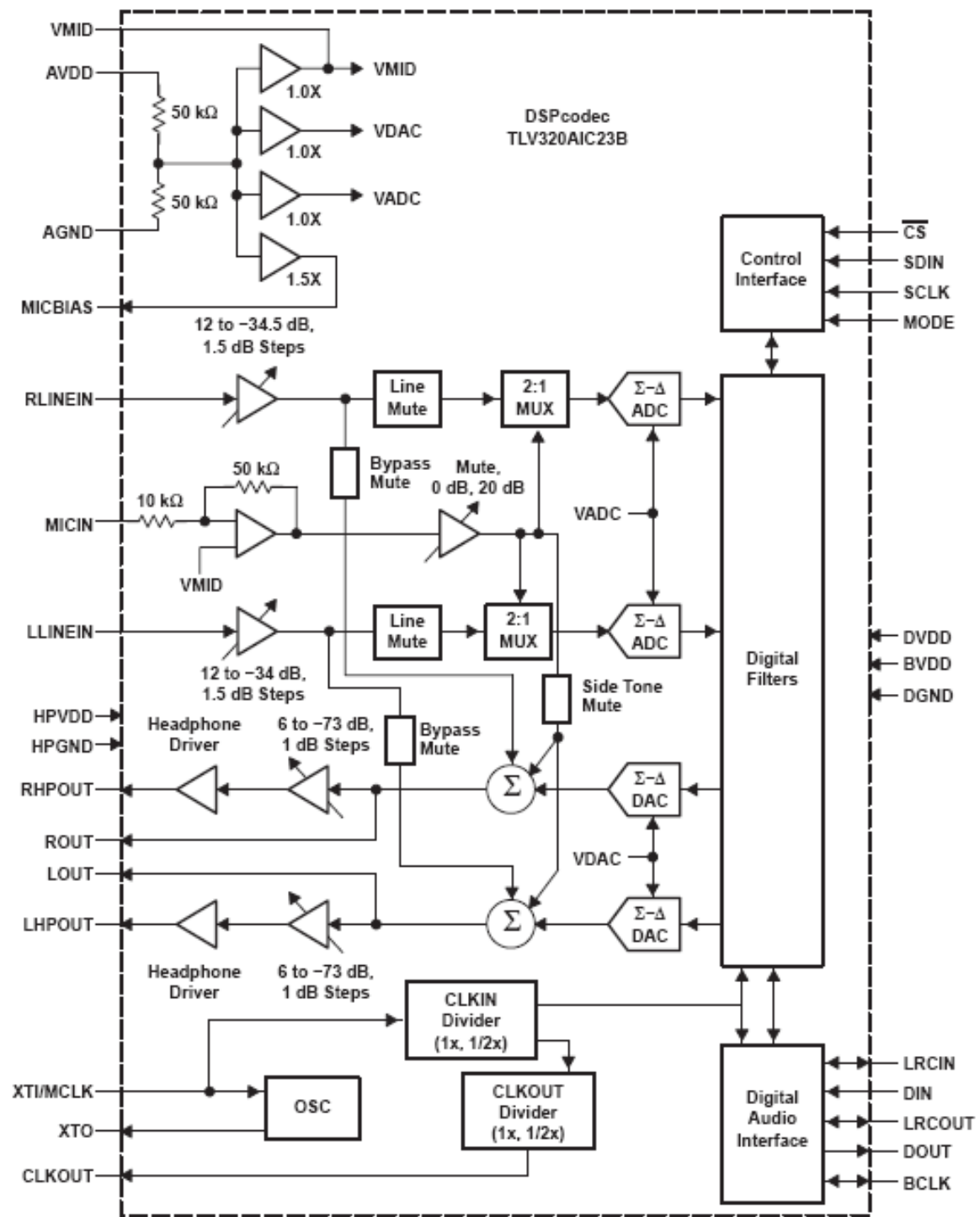
Listing A.3: Config McBSP1 - Init. CODEC - in-out functions

```

1 IRQ_globalDisable();           //disable interrupts
2 DSK6713_init();                //call BSL to init DSK-EMIF,PLL)
3 hAIC23_handle=DSK6713_AIC23_openCodec(0, &config);
4                               //handle(pointer) to codec
5 DSK6713_AIC23_setFreq(hAIC23_handle, fs);    //set sample rate
6
7 MCBSP_config(DSK6713_AIC23_DATAHANDLE,&AIC23CfgData);
8                               //configure McBSP1
9 MCBSP_start(DSK6713_AIC23_DATAHANDLE, MCBSP_XMIT_START |
10 MCBSP_RCV_START | MCBSP_SRGR_START |
11 MCBSP_SRGR_FRAMESYNC, 220); //start data channel again
12
13 CODECEventId= MCBSP_getXmtEventId(DSK6713_AIC23_DATAHANDLE);
14                               //McBSP1 Xmit

```

## A.4. The AIC23 CODEC



NOTE: MCLK, BCLK, and SCLK are all asynchronous to each other.

Figure A.3.: Block diagram of the AIC23B (from slws106h.pdf)

# Bibliography

- [1] Lab\_support\_en.pdf: *Hints for the lab.*  
Moodle, in this course under "Instructions and dokumentation"
- [2] 6713\_dsk\_techref.pdf: *TMS320C6713 DSK Technical Reference.*  
Moodle, in this course under "Instructions and dokumentation"
- [3] spra291.pdf: *Implementing FFT on TMS320 SPRA291.*  
Moodle, in this course under "Instructions and dokumentation"
- [4] spru401j.pdf: *TMS320C6000 Chip Support Library API Reference Guide.*  
Moodle, in this course under "Instructions and dokumentation"
- [5] spra488c.pdf: *TMS320C6000 McBSP Initialization.*  
Moodle, in this course under "Instructions and dokumentation"
- [6] CCS5\_Graphical\_Display\_en.pdf: *Instructions for using graphical display of data buffers.*  
Moodle, in this course under "Instructions and dokumentation"
- [7] CCS5\_ProfilingCode\_en.pdf: *How to profile code.*  
Moodle, in this course under "Instructions and dokumentation"
- [8] UPV\_Starter\_en.pdf: *Introduction Audioanalyser R&S UPV.*  
Moodle, in this course under "Instructions and dokumentation"
- [9] S.K.Mitra: *Digital Signal Processing, McGraw-Hill, 2001*
- [10] E.C.Ifeachor, B.W.Jervis: *Digital Signal Processing - A Practical Approach, 2nd ed., Prentice Hall, 2002*
- [11] von Grünigen: *Digitale Signalverarbeitung, Fachbuchverlag Leipzig, 2004*