Lab 6
Lab report due by 2:00 PM on Monday, May 2, 2016

This lab involves programming the TMS320C6713 DSP Starter Kit (DSK) module. Because we do not have enough boards for all students, you will need to partner with a student to run the experiments. However, each student must turn in her/his own data and write-up including C codes. You must write your own analysis.

---

1 Introduction

In this lab you will learn how to program on the TMS320C6713 DSK using Code Composer Studio. You will implement filters for audio processing.

2 C6713DSK in Code Composer Studio V4

Code Composer Studio V4 is known to work with the TMS320C6713 board. Most computers in the lab should have CCS4, and CCS5. If you prefer, you can work with your laptop, and install CCS5 and all the libraries (see http://processors.wiki.ti.com/index.php/C6713DSK_in_CCSv5). You will need to connect the board to your laptop with the USB cable.

3 C6713DSK in Code Composer Studio V5.3

The following link provides the detailed instruction on how to setup a project in CCS V5.3, with all the necessary files. The installation worked without a glitch on a Macbook Pro running Windows 7.
http://processors.wiki.ti.com/index.php/C6713DSK_in_CCSv5

In the Spectrum Digital foldder, C:\DSK6713 \drivers, right mouse click on c6713dsk.exe, and click on "troubleshoot compatibility". Get it running as an XP program. It should be persistent, so you only have to do it once. Do it for the 6713DSKDiag.exe, and you can run diagnostics on your board, which you should do to rule out any board issues.

You can test your configuration with this very small project:
http://processors.wiki.ti.com/index.php/File:C6713_audio_minimal.zip

The user guide for Code Composer Studio is available here:
http://processors.wiki.ti.com/index.php/Code_Composer_Studio_v5_Users_Guide

There is also a tutorial video here:
http://software-dl.ti.com/dsps/dsps_public_sw/sdo_ccstudio/CCSv5/Demos/CCSv5GettingStarted.html

3.1 Build settings

The following settings should be already set up for you if you use the example available on D2L `startup.zip`.

- General tab
  - Device:
    - Family: C6000
    - Variant: TMS320C6713
  - Connection: Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator
  - Linker command file: C6713DSK.cmd
  - Compiler version: TI v7.4.1
  - Runtime support library: rts6700.lib
- C6000 Compiler tab
  - Basic Options tab:
    - Target Processer version: 6700
    - Include Options tab
      "$CG_TOOL_ROOT/include"
      "C:\DSK6713\c6000\dsk6713\include" (or wherever the file is installed)
      "C:\DSK6713\c6000\bios\include" (or wherever the file is installed)
  - Advanced Options tab:
    - Predefined Symbols tab: CHIP_6713
    - Runtime Model Options tab:
      Data access model (-mem_model:data) far
- Linker tab
  - File Search Path
    "C:\C6xCSL\lib_3x \csl6713.lib"
    "C:\DSK6713\c6000\dsk6713\lib\dsk6713bsl.lib"
    "C:\ti\ccsv5\tools\compiler\c6000_7.4.1\lib \rts6700.lib"

4 Input and output using polling

4.1 The Audio codec

The TMS320C6713 board uses a stereo audio codec to sample input analog signals (Analog to Digital, ADC) and reconstruct analog signals from discrete signals (Digital to Analog, DAC). The sampling rate can be selected from a range of alternative settings from 8 to 96kHz.

The codec is a Texas Instruments AIC23 codec. The documentation for the codec is availabl on D2L: tlv320aic23b.pdf. Please download the manual and familiarize yourself with the register map on page 3-2. You will need this information.

The codec is connected to two analog audio inputs: microphone input, line input and two analog outputs: line output and headphone output, through four 3.5 mm audio jacks. The codec can select the microphone or the line input as the active input. The analog output is driven to both the line out (fixed gain) and headphone (adjustable gain) connectors

The codec is connected to two multichannel buffered serial ports (McBSPs), McBSP0 and McBSP1. McBSP0 is used to send commands to the codec control interface while McBSP1 is used for digital audio data. Once the codec is configured, the control channel (McBSP0) is normally idle while audio data is being transmitted on McBSP1. McBSP1 is used as the bi-directional data channel for ADC input and DAC output. We will configure the to use 16-bit samples in two's complement signed format.

4.2 The Chip Support Library API

The Chip Support Library (CSL) provides an application programming interface (API) used for configuring and controlling the DSP on-chip peripherals. To use a McBSP port, you must first open it and obtain a device handle using MCBSP_open(). Once opened, use the device handle to call the other API functions. The port may be configured by passing a MCBSP_Config structure to MCBSP_config() or by passing register values to the MCBSP_configArg function.

The most important functions are:

- MCBSP_open: Opens a McBSP port for use
- MCBSP_start: Starts the McBSP device
- MCBSP_close: Closes a McBSP port previously opened via MCBSP_open()
- MCBSP_config: Sets up the McBSP port using the configuration structure
- MCBSP_rrdy : Reads the RRDY status bit of the SPCR register
- MCBSP_read: Performs a direct 32-bit read of the data receive register DRR

- MCBSP_write: Writes a 32-bit value directly to the serial port data transmit register, DXR
- MCBSP_xrdy: Reads the XRDY status bit of the SPCR register

4.3 The Board Support Library API

Instead of using the CSL library, we can use the Board Support Library (BSL) dsk6713bsl32.lib (see Appendix for the available function interfaces).

The codec initialization function DSK6713_AIC23_openCodec opens the codec (using MCBSP_open and MCBSP_config), performs an initialization according to the structure config (by writing into the registers using MCBSP_write), and returns a codec handle to access the codec.

The following configuration corresponds to some default values for the register map (see tlv320aic23b.pdf)

```
// Codec configuration settings
DSK6713_AIC23_Config config = { \
    0x0017,  /* 0 DSK6713_AIC23_LEFTINVOL  Left line input channel volume */ \
    0x0017,  /* 1 DSK6713_AIC23_RIGHTINVOL Right line input channel volume */\
    0x01F9,  /* 2 DSK6713_AIC23_LEFTHPVOL  Left channel headphone volume */  \
    0x01F9,  /* 3 DSK6713_AIC23_RIGHTHPVOL Right channel headphone volume */ \
    0x0011,  /* 4 DSK6713_AIC23_ANAPATH    Analog audio path control */      \
    0x0000,  /* 5 DSK6713_AIC23_DIGPATH    Digital audio path control */     \
    0x0000,  /* 6 DSK6713_AIC23_POWERDOWN  Power down control */             \
    0x0043,  /* 7 DSK6713_AIC23_DIGIF      Digital audio interface format */ \
    0x0001,  /* 8 DSK6713_AIC23_SAMPLERATE Sample rate control */            \
    0x0001   /* 9 DSK6713_AIC23_DIGACT     Digital interface activation */   \
};
```

Figure 1: Codec configuration settings

The following function, DSK6713_AIC23_read from the BSL library, provides a higher level abstraction for reading the left and right channel directly using polling.

```
#include <dsk6713.h>
#include <dsk6713_aic23.h>

Int16 DSK6713_AIC23_read (DSK6713_AIC23_CodecHandle hCodec, Uint32 *val)
{/* If no new data available, return false */
    if (!MCBSP_rrdy(DSK6713_AIC23_DATAHANDLE)) {
        return (FALSE);
    }
    /* Read the data */
    *val = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
    return (TRUE);
}
```

Figure 2: Function DSK6713_AIC23_read from the (BSL) dsk6713bsl32.lib

Assignment

1. Use the codec documentation tlv320aic23b.pdf and describe the values that are in the structure config (of type DSK6713_AIC23_Config) above. You answer needs to be justified using the codec documentation and the register maps.
2. Explain how to change the volume settings in the config structure. What is the minimum value? what is the maximum value?

4.4 A simple audio loop

The program shown in Fig. 3 implements a simple audio loop:

1. An audio sample is read from the codec using the function DSK6713_AIC23_read. Note that the code waits inside the while loop until the codec is ready to read. The left and right channels are both inside the unsigned 32 bit variable STEREO.
2. The left and right channel values are written back to the codec, reconstructed as analog signal, and pushed on the headphone line. Note that the code waits until the function DSK6713_AIC23_write has finished writing the sample.
3. The infinite for loop never ends.

```
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "stdlib.h"
#include "math.h"

// Codec configuration settings
DSK6713_AIC23_Config config = { \
    0x0017,  /* 0 DSK6713_AIC23_LEFTINVOL  Left line input channel volume */ \
    0x0017,  /* 1 DSK6713_AIC23_RIGHTINVOL Right line input channel volume */\
    0x01F9,  /* 2 DSK6713_AIC23_LEFTHPVOL  Left channel headphone volume */  \
    0x01F9,  /* 3 DSK6713_AIC23_RIGHTHPVOL Right channel headphone volume */ \
    0x0011,  /* 4 DSK6713_AIC23_ANAPATH    Analog audio path control */      \
    0x0000,  /* 5 DSK6713_AIC23_DIGPATH    Digital audio path control */     \
    0x0000,  /* 6 DSK6713_AIC23_POWERDOWN  Power down control */             \
    0x0043,  /* 7 DSK6713_AIC23_DIGIF      Digital audio interface format */ \
    0x0001,  /* 8 DSK6713_AIC23_SAMPLERATE Sample rate control */            \
    0x0001   /* 9 DSK6713_AIC23_DIGACT     Digital interface activation */   \
};

void main()
{
        DSK6713_AIC23_CodecHandle hCodec;
        Uint32 stereo;

        // Initialize BSL
        DSK6713_init();

        //Start codec
        hCodec = DSK6713_AIC23_openCodec(0, &config);

        // Set  frequency to 48KHz
        DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_48KHZ);

        for(;;)
        {
          // Read left and right channel samples
          while (!DSK6713_AIC23_read(hCodec, &stereo));

          // write left and right channel samples
          while (!DSK6713_AIC23_write(hCodec, stereo));
       }

       // the statement below is never reached
       // DSK6713_AIC23_closeCodec(hCodec);
}
```


Figure 3: Function main.c that implements an audio loop using polling.

Assignment

3. Download the following archive, which contains a CCS project, startup.zip. Compile and test the code.

4. You will modify the function main.c. Change the input and output volume of the left channel in the config structure. Recompile the code, and check that your changes affect the corresponding output.

5. Modify the sampling rate, f_s, and describe the discernible effect on the reconstructed signal. You will use 8kHz, 16kHz, 32kHz, 44kHz (CD quality), and 96 kHz (archiving, digital studio quality).

6. Implement the following filter:

$$H(z) = 1 + \alpha z^{-N} \tag{1}$$

where $N = 500\text{ms} \times f_s$. You will try several values for α: $\alpha = 0.25, 0.5, 1$. Explain in plain English what the filter does.

5 Input and output using interrupts

The communication with the codec using polling requires that the DSP chip idles until the codec is ready to read or write. A more efficient method of communication involves using interrupts.

5.1 Interrupts

An interrupt is an event that stops the current process in the CPU so that the CPU can attend to the task needing completion because of another event. These events are external to the core CPU but may originate on-chip or off-chip.

When a maskable interrupt occurs, the CPU stops the execution of the current process, and attends to the interrupt. The current values of the resister are saved, and the control is passed to a servicing routine, or interrupt handler. Once the interrupt servicing routine completes, the context that existed before the interrupt occurred is restored.

The C6713 CPU has 14 interrupts. These are reset, the non maskable interrupt (NMI), and interrupts 4-15. These interrupts correspond to the RESET, NMI, and INT_4-INT_15 signals on the CPU boundary. The highest priority interrupt is INT_00 (dedicated to RESET), while the lowest priority is INT_15. NMI is the second-highest priority interrupt and is generally used to alert the CPU of a serious hardware problem such as imminent power failure. The first four interrupts are non-maskable and fixed.

The remaining interrupts (4-15) are maskable and default to a list of interrupt sources. The interrupt source may be reprogrammed to be associated with timers, serial ports, DMAs, analog-to-digital converters, host controllers and other peripheral devices. For our purpose

we will map INT_11 to the interrupt associated with a transmit event on port McBSP1 (the data port for the audio codec).

When the CPU executes an interrupt service routine, other interrupts are disabled (except for the NMI interrupt). In this lab we will not allow interrupts of higher priority to interrupt a service routine. As a result, when an interrupt handler routine is accessing some global variable, no other routine can modify the same variable concurrently.

The CSL provides an API to the manipulation of interrupts and the definition of interrupt handling functions.

5.2 Communication with the audio codec using interrupts

The Serial Port Control Register controls the 32-peripheral bus to access the Multichannel Buffered Serial Port (McBSP). The RRDY and XRDY bits in SPCR indicate the ready state of the McBSP receiver and transmitter, respectively. Reading DRR and writing to DXR affects RRDY and XRDY, respectively. Writes and reads from the serial port can be synchronized by any of the following methods:

- Polling RRDY and XRDY bits. This is the method we used in the previous section.
- Using the events sent to the DMA or EDMA controller (REVT and XEVT). We will not be using this approach.
- Using the interrupts to the CPU (RINT and XINT) that the events generate.

We will be using the last approach in this section.

We now explain how to build a servicing routine that will read and write data as soon as the XRDY interrupt occurs.

An important remark is in order here. In principle, we should be using two interrupt handlers triggered by the XRDY and RRDY bits to transmit and read, respectively. However, we can rely on a single interrupt (the XRDY, for instance) and safely read and write on the bus. Indeed, in the function DSK6713_AIC23_openCodec, the connection with the codec is open in such a way that the read and write operations on the 32-peripheral bus are synchro (frame sync), and controlled by the audio codec. This configuration means that both the transmit and receive data are synchronized with phase alignment. In other words, each receive (the codec sends a frame to the chip) or transmit (the chip sends a frame to the coded) is aligned on the same clock. The important consequence is that each time the bus is ready to transmit (transmit status bit XRDY = 1), a new input sample can be read because the transmit and receive frame syncs are identical.

The code in the function initIRQ in Fig. 4 demonstrates the steps that are necessary to set up an interrupt handling function. In this case, we map the interrupt XRDY, which indicates that the bus Mc_BSP1 is ready for transmitting data, and we map this interrupt to INT_11. The particular event ID associated with XRDY is retrieved using the CSL function MCBSP_getXmtEventId with the codec handle DSK6713_AIC23_codecdatahandle. DSK6713_AIC23_codec

is actually a #define for DSK6713_AIC23_DATAHANDLE, which is a global variable that is returned by the function MCBSP_open, when the communication with McBSP1 is open. The functio MCBSP_open is called in DSK6713_AIC23_openCodec for both McBSP0 and McBSP1.

```
void initIRQ(int IRQ_id)
{
  // Globally disables interrupts by clearing the GIE bit of the CSR register.

  IRQ_globalDisable();

  // initialize the chip and the codec

  c6713_dsk_init();

  // Retrieves transmit event ID for McBSP1.  Note that the handle to
  // the McBSP1 port  DSK6713_AIC23_codecdatahandle is generated by MCBSP_open

  CODECEventId = MCBSP_getXmtEventId(DSK6713_AIC23_codecdatahandle);

  // sets the base address of the interrupt vector table

  IRQ_setVecs(vectors);

  // map the event ID associated with the handling of McBSP1
  // with the physical interrupt IRQ_id

  IRQ_map(CODECEventId, IRQ_id);

  // resets the event ID by disabling then clearing it.

  IRQ_reset(CODECEventId);

  // Globally enables interrupt. This function globally enables
  // interrupts by setting the GIE bit of the CSR register to 1.

  IRQ_globalEnable();

  // Enables the NMI interrupt event

  IRQ_nmiEnable();

  // enable the specified event

  IRQ_enable(CODECEventId);

  pushAIC23 (0);                    // start McBSP interrupt outputting a sample
}
```

Figure 4: Initialization of the interrupt servicing function

The assembly code in Fig. 6 and 7 initializes the interrupt service table (IST). The IST is a table of fetch packets that contain code for servicing the interrupts. When the CPU begins processing an interrupt, it references the IST. A branch to the location of the interrupt service routine code is found in the IST. Finally, Fig. 5 demonstrates how the interrupt servicing function handles the read and write events coming from the codec.

```
#include "DSK6713_AIC23.h"              // codec support
#define DSK6713_AIC23_INPUT_MIC  0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011

Uint16 inputsource=DSK6713_AIC23_INPUT_LINE; //  select input
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;      // set sampling rate

extern Uint32 fetchAIC23 (void);
extern void   output_sample (int);
extern void   initIRQ(int IRQ_id);

interrupt void c_int11()              // interrupt service routine
{
  Uint32 stereo;

  stereo = fetchAIC23 ();                    // input data
  pushAIC23 (stereo);                        // output data
  return;
}

void main()
{
  initIRQ(11);                                 // init DSK, codec, McBSP
  while(1);                                 // infinite loop
}
```

Figure 5: Example of a stereo audio loop using interrupts.

```
*-------------------------------------------------------------------------------
* Global symbols defined here and exported out of this file
*-------------------------------------------------------------------------------
   .global _vectors ;global symbols
   .global _c_int00
   .global _vector1
   .global _vector2
   .global _vector3
   .global _vector4
   .global _vector5
   .global _vector6
   .global _vector7
   .global _vector8
   .global _vector9
   .global _vector10
   .global _c_int11 ; Hookup the c_int11 ISR in main()
   .global _vector12
   .global _vector13
   .global _vector14
   .global _vector15

*-------------------------------------------------------------------------------
* Global symbols referenced in this file but defined somewhere else.
* Remember that your interrupt service routines need to be referenced here.
*-------------------------------------------------------------------------------
   .ref _c_int00 ;entry address

*-------------------------------------------------------------------------------
* This is a macro that instantiates one entry in the interrupt service table.
*-------------------------------------------------------------------------------
RST_VEC_ENTRY .macro addr
    NOP
    MVKL  addr,B0
    MVKH  addr,B0
    B     B0
    NOP
    NOP   2
    NOP
    NOP
   .endm
```

Figure 6: Assembly code to define the interrupt service table. Continues in Fig 7.

```
VEC_ENTRY .macro addr
    STW   B0,*--B15
    MVKL  addr,B0
    MVKH  addr,B0
    B     B0
    LDW   *B15++,B0
    NOP   2
    NOP
    NOP
   .endm
*------------------------------------------------------------------------------
* This is a dummy interrupt service routine used to initialize the IST.
*------------------------------------------------------------------------------
_vec_dummy:
  B    B3
  NOP  5
*------------------------------------------------------------------------------
* This is the actual interrupt service table (IST). It is properly aligned and
* is located in the subsection .text:vecs. This means if you don't explicitly
* specify this section in your linker command file, it will default and link
* into the .text section. Remember to set the ISTP register to point to this
* table.
*------------------------------------------------------------------------------
.sect ".vectors"    ;aligned IST section
 .align 1024
_vectors:
_vector0:   RST_VEC_ENTRY _c_int00 ;RESET
_vector1:   VEC_ENTRY _vec_dummy   ;NMI
_vector2:   VEC_ENTRY _vec_dummy   ;RSVD
_vector3:   VEC_ENTRY _vec_dummy
_vector4:   VEC_ENTRY _vec_dummy
_vector5:   VEC_ENTRY _vec_dummy
_vector6:   VEC_ENTRY _vec_dummy
_vector7:   VEC_ENTRY _vec_dummy
_vector8:   VEC_ENTRY _vec_dummy
_vector9:   VEC_ENTRY _vec_dummy
_vector10:  VEC_ENTRY _vec_dummy
_vector11:  VEC_ENTRY _c_int11    ; Hookup the c_int11 ISR in main()
_vector12:  VEC_ENTRY _vec_dummy
_vector13:  VEC_ENTRY _vec_dummy
_vector14:  VEC_ENTRY _vec_dummy
_vector15:  VEC_ENTRY _vec_dummy
```

Figure 7: Assembly code (continued from Fig. 6) to define the interrupt service table.

Assignment

7. Write a function with the following specification:

```
void pushAIC23 (AIC_data stereo)
{
       // output the left and right channels into the DAC of the codec
       // using the serial port data transmit register, McBSP1
}
```

   where the structure AIC_data is defined in c6713dskinit.h.

```
#define LEFT 1 //data structure for union of 32-bit data
#define RIGHT 0 //into two 16-bit data
union {
     Uint32 uint;
     short channel[2];
} AIC_data;
```

   You will use the function MCBSP_write (see the documentation on D2L CSL_API.pdf).

8. Write a function with the following specification: Uint32 fetchAIC23 (void)

```
{
       // fetch the left and right channels from the ADC of the codec
       // using the serial port data transmit register, McBSP1
}
```

   You will use the function MCBSP_read (see the documentation on D2L CSL_API.pdf).

9. Use the code in Fig. 5, available on D2L interrupt.c to implement a simple stereo audio loop.

   You will need the code shown in Figs. 6 and 7, available on D2L vectors_irq.asm.

   You will also need the code shown in Figs. 4, available on D2L c6713dskinit.zip.

   You will need to add the files c6713dskint.c, files c6713dskint.h, and vectors_irq.asm to the project.

Assignment

10. Using interrupts, write a function stero_tone (fl,fr) that outputs a sinusoidal waveform of fl Hz in the left channel, and fr Hz in the right channel of the headphone output. Your function will output one sample for call to c_int11. You will need to keep track of time using a global variable.

11. Using interrupts, write a function fir that filters the audio input from the codec and outputs the filtered audio to the headphone. The filter coefficients are

$$h[0] = \frac{3}{4},\ h[\pm 1] = \frac{1}{4},\ h[\pm 2] = -\frac{1}{8} \tag{2}$$

Compute the frequency response of the filter, and describe its role. Compare the expected behavior and the actual output.
What is the important property that the filter has?