Lab 1 – TMS320C6713 DSK and Code Composer Studio

1.1. Introduction

The hardware experiments in the DSP lab are carried out on the Texas Instruments TMS320C6713 DSP Starter Kit (DSK), based on the TMS320C6713 floating point DSP running at 225 MHz. The basic clock cycle instruction time is $1/(225\text{ MHz}) = 4.44$ nanoseconds. During each clock cycle, up to eight instructions can be carried out in parallel, achieving up to $8\times 225 = 1800$ million instructions per second (MIPS).

The C6713 processor has 256KB of internal memory, and can potentially address 4GB of external memory. The DSK board includes a 16MB SDRAM memory and a 512KB Flash ROM. It has an on-board 16-bit audio stereo codec (the Texas Instruments AIC23B) that serves both as an A/D and a D/A converter. There are four 3.5 mm audio jacks for microphone and stereo line input, and speaker and head-phone outputs. The AIC23 codec can be programmed to sample audio inputs at the following sampling rates:

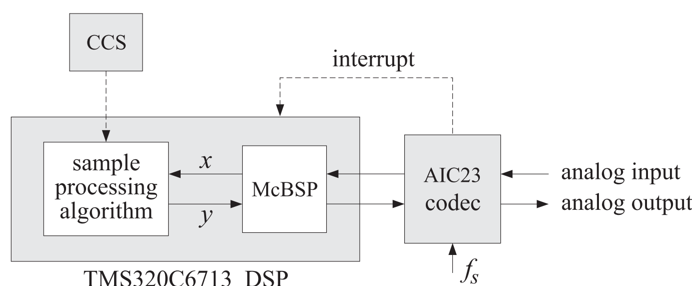
$$f_s = 8,\ 16,\ 24,\ 32,\ 44.1,\ 48,\ 96\ \text{kHz}$$

The ADC part of the codec is implemented as a multi-bit third-order noise-shaping delta-sigma converter (see Ch. 2 & 12 of [1] for the theory of such converters) that allows a variety of oversampling ratios that can realize the above choices of f_s. The corresponding oversampling decimation filters act as anti-aliasing prefilters that limit the spectrum of the input analog signals effectively to the Nyquist interval $[-f_s/2, f_s/2]$. The DAC part is similarly implemented as a multi-bit second-order noise-shaping delta-sigma converter whose oversampling interpolation filters act as almost ideal reconstruction filters with the Nyquist interval as their passband.

The DSK also has four user-programmable DIP switches and four LEDs that can be used to control and monitor programs running on the DSP.

All features of the DSK are managed by the CCS, which is a complete integrated development environment (IDE) that includes an optimizing C/C++ compiler, assembler, linker, debugger, and program loader. The CCS communicates with the DSK via a USB connection to a PC. In addition to facilitating all programming aspects of the C6713 DSP, the CCS can also read signals stored on the DSP's memory, or the SDRAM, and plot them in the time or frequency domains.

The following block diagram depicts the overall operations involved in all of the hardware experiments in the DSP lab. Processing is interrupt-driven at the sampling rate f_s, as explained below.



The AIC23 codec is configured (through CCS) to operate at one of the above sampling rates f_s. Each collected sample is converted to a 16-bit two's complement integer (a **short** data type in C). The codec actually samples the audio input in stereo, that is, it collects two samples for the left and right channels.

At each sampling instant, the codec combines the two 16-bit left/right samples into a single 32-bit unsigned integer word (an **unsigned int**, or **Uint32** data type in C), and ships it over to a 32-bit receive-register of the multichannel buffered serial port (McBSP) of the C6713 processor, and then issues an interrupt to the processor.

Upon receiving the interrupt, the processor executes an interrupt service routine (ISR) that implements a desired sample processing algorithm programmed with the CCS (e.g., filtering, audio effects, etc.). During the ISR, the following actions take place: the 32-bit input sample (denoted by x in the diagram) is read from the McBSP, and sent into the sample processing algorithm that computes the corresponding

32-bit output word (denoted by y), which is then written back into a 32-bit transmit-register of the McBSP, from where it is transferred to the codec and reconstructed into analog format, and finally the ISR returns from interrupt, and the processor begins waiting for the next interrupt, which will come at the next sampling instant.

Clearly, all processing operations during the execution of the ISR must be completed in the time interval between samples, that is, $T = 1/f_s$. For example, if $f_s = 44.1$ kHz, then, $T = 1/f_s = 22.68$ μsec. With an instruction cycle time of $T_c = 4.44$ nsec, this allows $T/T_c = 5108$ cycles to be executed during each sampling instant, or, up to $8\times5108 = 40864$ instructions, or half of that per channel.

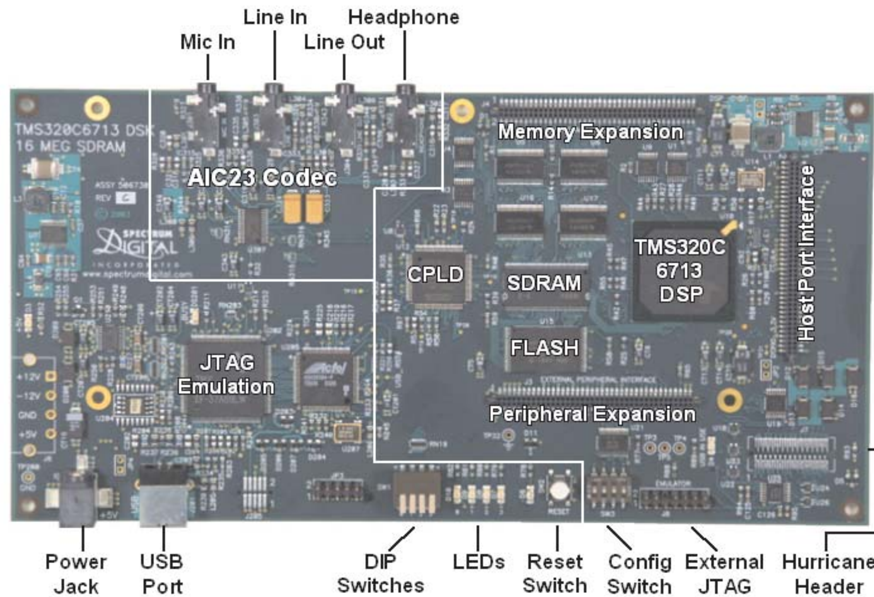
Resources

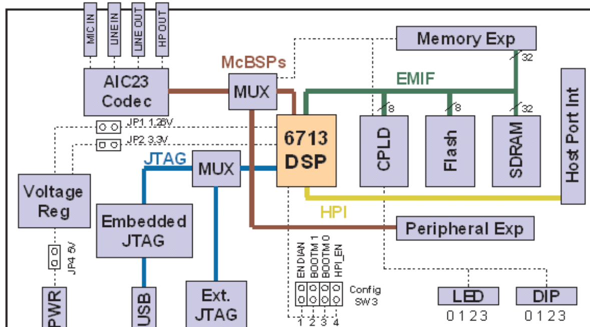
Most of the hardware experiments in the DSP lab are based on C code from the text [1] adapted to the CCS development environment. Additional experiments are based on the Chassaing-Reay text [2].

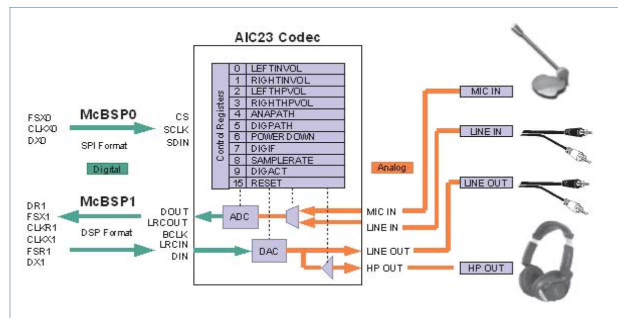
The web page of the lab, `http://www.ece.rutgers.edu/~orfanidi/ece348/`, contains additional resources such as tutorials and user guides. Some books on C and links to the GNU GCC C compiler are given in Ref. [5].

As a prelab, before you attend Lab-1, please go through the powerpoint presentations of Brown's workshop tutorial in Ref. [3], Part-1, and Dahnoun's chapters 1 & 3 listed in Ref. [4]. These will give you a pretty good idea of the TMS320C6000 architecture and features.

The help file, `C:\CCStudio_v3.1\docs\hlp\c6713dsk.hlp`, found in the CCS installation directory of each PC, contains very useful information on the C6713 processor and DSK kit. The following pictures are from that help file:







1.2. Lab Tasks

In this lab, you will learn how to use some basic features of the Code Composer Studio (CCS), such as creating projects, compiling and linking them to the run-time libraries, loading them for execution on the DSP chip, using GEL files for changing program parameters during run-time.

You will hear what aliasing effects sound like (i.e., distortions arising from using the wrong sampling rate). You will hear what quantization effects sound like (i.e., when you use too few bits for your audio samples). You will find out how the stereo A/D converter packs the two 16-bit samples from the left and right audio channels into a 32-bit word and sends it over to the processor, and how it gets unpacked into the two individual 16-bit left/right words by the processor. You will also study panning between speakers, and several nonlinear input/output functions such as fuzz (hard clipping) and tube amplifier (soft clipping) for guitar distortion.

1.3. Template Program

You will begin with a basic talkthrough program, listed below, that simply reads input samples from the codec and immediately writes them back out. This will serve as a template on which to build more complicated sample processing algorithms by modifying the interrupt service routine `isr()`.

```
// template.c - to be used as starting point for interrupt-based programs
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // DSK initialization declarations and function prototypes

short xL, xR, yL, yR;        // left and right input and output samples from/to codec
float g=1;                   // gain to demonstrate watch windows and GEL files

// here, add more global variable declarations, #define's, #include's, etc.
// ------------------------------------------------------------------------------------

void main()                  // main program executed first
{
  initialize();              // initialize DSK board and codec, define interrupts

  sampling_rate(8);          // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
  audio_source(LINE);        // LINE or MIC for line or microphone input

  while(1);                  // keep waiting for interrupt, then jump to isr()
}

// ------------------------------------------------------------------------------------

interrupt void isr()         // sample processing algorithm - interrupt service routine
{
   read_inputs(&xL, &xR);    // read left and right input samples from codec

   yL = g * xL;              // replace these with your sample processing algorithm
   yR = g * xR;

   write_outputs(yL,yR);     // write left and right output samples to codec

   return;
}

// ------------------------------------------------------------------------------------
// here, add more functions to be called within isr() or main()
```

The template has three sections. In the top section, global variables are declared and defined, such as the left/right input/output audio samples x_L, x_R, y_L, y_R, whose scope is the entire file and are known to all functions in the file. Additional `#define` and `#include` statements, such as `#include <math.h>`, and additional global variable declarations may be added in this section.

The second section consists of the function `main()`, which is executed first, and performs the initialization of the DSK board, sets the sampling rate, selects the audio input, and then goes into an infinite loop waiting for an interrupt. Upon receiving the interrupt, it jumps to the function `isr()`. Additional local variables and other preliminary operations, such as the zeroing of delay-line buffers, may be added in this section before the `wait(1)` statement.

The third section consists of the interrupt service routine `isr()`, which implements the desired sample processing algorithm. Note that the keyword **interrupt** has been added to the C language implementation of the CCS. In the template file, the ISR function reads the left/right input samples, process them by multiplying them by a gain, sends them to the output, and returns back to `main()`.

The reading and writing of the input and output samples are done with the help of the functions `read_inputs()` and `write_outputs()`, which are declared in the header file `dsplab.h` and defined in `dsplab.c`. These two files must always be included in your programs and reside in the common directory `C:\dsplab\common\`.

Besides the above three basic sections, other sections may be added that define additional functions to be called within `isr()` or `main()`.

Working with CCS

For each application to be run on the C6713 processor, one must create a "project" in the Code Composer Studio, which puts together all the information about the required C source files, header files, and C libraries, including all the compiler and linker build options. During the lab session, you will be working in the temporary folders:

```
C:\labuser
C:\labuser\dsplab
C:\labuser\dsplab\template
C:\labuser\dsplab\examples
```

Before you start work, please double-click on the desktop icon called `cleanup`. This refreshes the above `labuser` directory and removes all files created by previous students. Consequently, before you leave the lab, you must save your work files in a thumb-drive or email them to yourselves.

To save you time, the project file, `template.pjt`, for the above template has already been created, and may be simply edited for all other projects. To proceed, copy the following three files from the template directory `C:\labuser\dsplab\template`:

```
template.c
template.pjt
template.gel
```

into your temporary working directory, e.g., `C:\labuser`, and double-click the project file, `template.pjt`, which will open in an ordinary text editor. The first few lines of that file are shown below:

```
[Project Settings]
ProjectDir="C:\dsplab\template\"
ProjectType=Executable
CPUFamily=TMS320C67XX
Tool="Compiler"
Tool="CustomBuilder"
Tool="DspBiosBuilder"
Tool="Linker"
Config="Debug"
Config="Release"

[Source Files]
Source="C:\CCStudio_v3.1\C6000\cgtools\lib\rts6700.lib"
Source="C:\CCStudio_v3.1\C6000\csl\lib\csl6713.lib"
Source="C:\CCStudio_v3.1\C6000\dsk6713\lib\dsk6713bsl.lib"
Source="C:\dsplab\common\dsplab.c"
Source="C:\dsplab\common\vectors.asm"
Source="template.c"
```

Only the second and bottom lines in the above listing need to be edited. First, edit the project directory entry to your working directory, e.g.,

```
ProjectDir="C:\labuser"
```

Alternatively, you may delete that line—it will be recreated by CCS when you load the project. Then, edit the source-file line `Source="template.c"` to your new project's name, e.g.,

```
Source="new_project.c"
```

Finally, rename the three files with your new names, e.g.,

```
new_project.c
new_project.pjt
new_project.gel
```

Next, turn on the DSK kit and after the initialization beep, open the CCS application by double-clicking on the CCS desktop icon. Immediately after it opens, use the keyboard combination "ALT+C" (or the menu item *Debug -> Connect*) to connect it to the processor. Then, with the menu item *Project -> Open* or the key combination "ALT+P O", open the newly created project file by navigating to the project's directory, e.g., `C:\labuser`. Once the project loads, you may edit the C source file to implement your algorithm. Additional C source files can be added to your project by the keyboard combination "ALT+P A" or the menu choices *Project -> Add Files to Project*.

Set up CCS to automatically load the program after building it, with the menu commands: *Option -> Customize -> Program/Project Load -> Load Program After Build*. The following key combinations or menu items allow you to compile and load your program, run or halt the program:

```
compile & load:   F7,         Project -> Build
run program:      F5,         Debug -> Run
halt program:     Shift+F5,   Debug -> Halt
```

It is possible that the first time you try to build your program you will get a warning message:

```
warning: creating .stack section with default size of 400 (hex) words
```

In such case, simply rebuild the project, or, in the menu item *Project -> Build Options -> Linker*, enter a value such as `0x500` in the stack entry.

When you are done, please remember to save and close your project with the keyboard combinations "ALT+P S" and "ALT+P C", and save your programs in your account on ECE.

Lab Procedure

a. Copy the template files into your temporary working directory, edit the project's directory as described above, and build the project in CCS. Connect your MP3 player to the line input of the DSK board and play your favorite song, or, you may play one of the wave files in the directory: `c:\dsplab\wav`.

b. Review the template project's build options using the menu commands: *Project -> Build Options*. In particular, review the Basic, Advanced, and Preprocessor options for the Compiler, and note that the optimization level was set to `none`. In future experiments, this may be changed to `-o2` or `-o3`.

   For the Linker options, review the Basic and Advanced settings. In particular, note that the default output name `a.out` can be changed to anything else. Note also the library include paths and that the standard included libraries are:

```
rts6700.lib     (run-time library),      C:\CCStudio_v3.1\C6000\cgtools\lib\rts6700.lib
csl6713.lib     (chip support library),  C:\CCStudio_v3.1\C6000\csl\lib\csl6713.lib
dsk6713bsl.lib  (board support library), C:\CCStudio_v3.1\C6000\dsk6713\lib\dsk6713bsl.lib
```

The run-time library must always be included. The board support library (BSL) contains functions for managing the DSK board peripherals, such as the codec. The chip support library (CSL) has functions for managing the DSP chip's features, such as reading and writing data to the chip's McBSP. The user manuals for these may be found on the TI web site listed on the lab's web page.

c. The gain parameter g can be controlled in real-time in two ways: using a watch window, or using a GEL file. Open a watch window using the menu item: *View -> Watch Window*, then choose *View -> Quick Watch* and enter the variable g and add it to the opened watch window using the item *Add to Watch*. Run the program and click on the g variable in the watch window and enter a new value, such as $g = 0.5$ or $g = 2$, and you will hear the difference in the volume of the output.

d. Close the watch window and open the GEL file, `template.gel`, with the menu *File -> Load GEL*. In the *GEL* menu of CCS a new item called "gain" has appeared. Choose it to open the gain slider. Run the program and move the slider to different positions. Actually, the slider does not represent the gain g itself, but rather the integer increment steps. The gain g changes by $1/10$ at each step. Open the GEL file to see how it is structured. You may use that as a template for other cases.

e. Modify the template program so that the output pans between the left and right speakers every 2 seconds, i.e., the left speaker plays for 2 sec, and then switches to the right speaker for another 2 sec, and so on. There are many ways of doing this, for example, you may replace your ISR function by

```
#define D 16000      // represents 2 sec at fs = 8 kHz
short d=0;           // move these before main()

interrupt void isr()
{
   read_inputs(&xL, &xR);

   yL = (d < D)  * xL;
   yR = (d >= D) * xR;

   if (++d >= 2*D) d=0;

   write_outputs(yL,yR);

   return;
}
```

Rebuild your program with these changes and play a song. In your lab write-up explain why and how this code works.

1.4. Aliasing

This part demonstrates aliasing effects. The smallest sampling rate that can be defined is 8 kHz with a Nyquist interval of $[-4, 4]$ kHz. Thus, if a sinusoidal signal is generated (e.g. with MATLAB) with frequency outside this interval, e.g., $f = 5$ kHz, and played into the line-input of the DSK, one might expect that it would be aliased with $f_a = f - f_s = 5 - 8 = -3$ kHz. However, this will not work because the antialiasing oversampling decimation filters of the codec filter out any such out-of-band components before they are sent to the processor.

An alternative is to decimate the signal by a factor of 2, i.e., dropping every other sample. If the codec sampling rate is set to 8 kHz and every other sample is dropped, the effective sampling rate will be 4 kHz, with a Nyquist interval of $[-2, 2]$ kHz. A sinusoid whose frequency is outside the decimated Nyquist interval $[-2, 2]$ kHz, but inside the true Nyquist interval $[-4, 4]$ kHz, will not be cut off by the antialiasing filter and will be aliased. For example, if $f = 3$ kHz, the decimated sinusoid will be aliased with $f_a = 3 - 4 = -1$ kHz.

Lab Procedure

Copy the template programs to your working directory. Set the sampling rate to 8 kHz and select line-input. Modify the template program to output every other sample, with zero values in-between. This can be accomplished in different ways, but a simple one is to define a "sampling pulse" periodic signal whose values alternate between 1 and 0, i.e., the sequence $[1, 0, 1, 0, 1, 0, \dots]$ and multiply the input samples by that sequence. The following simple code segment implements this idea:

```
yL = pulse * xL;
yR = pulse * xR;

pulse = (pulse==0);
```

where `pulse` must be globally initialized to 1 before `main()` and `isr()`. Why does this work? Next, rebuild the new program with CCS.

Open MATLAB and generate three sinusoids of frequencies $f_1 = 1$ kHz, $f_2 = 3$ kHz, and $f_3 = 1$ kHz, each of duration of 1 second, and concatenate them to form a 3-second signal. Then play this out of the PCs sound card using the `sound()` function. For example, the following MATLAB code will do this:

```
fs = 8000; f1 = 1000; f2 = 3000; f3 = 1000;
L = 8000; n = (0:L-1);
A = 1/5;                              % adjust playback volume

x1 = A * cos(2*pi*n*f1/fs);
x2 = A * cos(2*pi*n*f2/fs);
x3 = A * cos(2*pi*n*f3/fs);

sound([x1,x2,x3], fs);
```

a. Connect the sound card's audio output to the line-input of the DSK and rebuild/run the CCS down-sampling program after commenting out the line:

```
pulse = (pulse==0);
```

This disables the downsampling operation. Send the above concatenated sinusoids to the DSK input and you should hear three distinct 1-sec segments, with the middle one having a higher frequency.

b. Next, uncomment the above line so that downsampling takes place and rebuild/run the program. Send the concatenated sinusoids to the DSK and you should hear all three segments as though they have the same frequency (because the middle 3 kHz one is aliased with other ones at 1 kHz). You may also play your favorite song to hear the aliasing distortions, e.g., out of tune vocals.

c. Set the codec sampling rate to 44 kHz and repeat the previous two steps. What do you expect to hear in this case?

d. To confirm the antialiasing prefiltering action of the codec, replace the first two lines of the above MATLAB code by the following two:

```
fs = 16000; f1 = 1000; f2 = 5000; f3 = 1000;
L = 16000; n = (0:L-1);
```

Now, the middle sinusoid has frequency of 5 kHz and it should be cutoff by the antialiasing prefilter. Set the sampling rate to 8 kHz, turn off the downsampling operation, rebuild and run your program, and send this signal through the DSK, and describe what you hear.

1.5. Quantization

The DSK's codec is a 16-bit ADC/DAC with each sample represented by a two's complement integer. Given the 16-bit representation of a sample, $[b_1 b_2 \cdots b_{16}]$, the corresponding 16-bit integer is given by

$$x = (-b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \cdots + b_{16} 2^{-16}) 2^{16} \tag{1.1}$$

The MSB bit b_1 is the sign bit. The range of representable integers is: $-32768 \le x \le 32767$. As discussed in Ch. 2 of Ref. [1], for high-fidelity audio at least 16 bits are required to match the dynamic range of human hearing; for speech, 8 bits are sufficient. If the audio or speech samples are quantized to less than 8 bits, quantization noise will become audible.

The 16-bit samples can be requantized to fewer bits by a right/left bit-shifting operation. For example, right shifting by 3 bits will knock out the last 3 bits, then left shifting by 3 bits will result in a 16-bit number whose last three bits are zero, that is, a 13-bit integer. These operations are illustrated below:

$$[b_1, b_2, \dots, b_{13}, b_{14}, b_{15}, b_{16}] \quad \Rightarrow \quad [0, 0, 0, b_1, b_2, \dots, b_{13}] \quad \Rightarrow \quad [b_1, b_2, \dots, b_{13}, 0, 0, 0]$$

Lab Procedure

a. Modify the basic template program so that the output samples are requantized to B bits, where $1 \le B \le 16$. This requires right/left shifting by $L = 16 - B$ bits, and can be implemented very simply in C as follows:

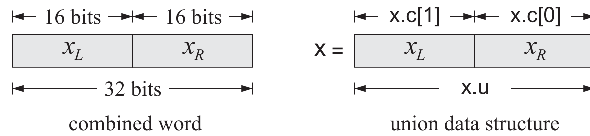
```
yL = (xL >> L) << L;
yR = (xR >> L) << L;
```

Start with $B = 16$, set the sampling rate to 8 kHz, and rebuild/run the program. Send a wave file as input and listen to the output.

b. Repeat with the following values: $B = 8, 6, 4, 2, 1$, and listen to the gradual increase in the quantization noise.

1.6. Data Transfers from/to Codec

We mentioned in the introduction that the codec samples the input in stereo, combines the two 16-bit left/right samples x_L, x_R into a single 32-bit unsigned integer word, and ships it over to a 32-bit receive-register of the McBSP of the C6713 processor. This is illustrated below.



The packing and unpacking of the two 16-bit words into a 32-bit word is accomplished with the help of a union data structure (see Refs. [2,3]) defined as follows:

```
union {              // union structure to facilitate 32-bit data transfers
   Uint32 u;         // both channels packed as codec.u = 32-bits
   short c[2];       // left-channel = codec.c[1], right-channel = codec.c[0]
} codec;
```

The two members of the data structure share a common 32-bit memory storage. The member `codec.u` contains the 32-bit word whose upper 16 bits represent the left sample, and its lower 16 bits, the right sample. The two-dimensional short array member `codec.c` holds the 16-bit right-channel sample in its first component, and the left-channel sample in its second, that is, we have:

```
xL = codec.c[1];
xR = codec.c[0];
```

The functions `read_inputs()` and `write_outputs()`, which are defined in the common file `dsplab.c`, use this structure in making calls to low-level McBSP read/write functions of the chip support library. They are defined as follows:

```
// ------------------------------------------------------------------------------------

void read_inputs(short *xL, short *xR)              // read left/right channels
{
   codec.u = MCBSP_read(DSK6713_AIC23_DATAHANDLE);   // read 32-bit word

   *xL = codec.c[1];                                // unpack the two 16-bit parts
   *xR = codec.c[0];
}

// ------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------

void write_outputs(short yL, short yR)              // write left/right channels
{
   codec.c[1] = yL;                                 // pack the two 16-bit parts
   codec.c[0] = yR;                                 // into 32-bit word

   MCBSP_write(DSK6713_AIC23_DATAHANDLE,codec.u);   // output left/right samples
}

// ------------------------------------------------------------------------------------
```

Lab Procedure

The purpose of this lab is to clarify the nature of the union data structure. Copy the template files into your working directory, rename them `unions.*`, and edit the project file by keeping in the source-files section only the run-time library and the main function below.

```
// unions.c - test union structure

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(void)
{
   unsigned int v;
   short xL,xR;

   union {
      unsigned int u;
      short c[2];
   } x;

   xL = 0x1234;
   xR = 0x5678;
   v  = 0x12345678;

   printf("\n%x  %x  %d  %d\n", xL,xR, xL,xR);

   x.c[1] = xL;
   x.c[0] = xR;
   printf("\n%x  %x  %x  %d  %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);

   x.u = v;
   printf("%x  %x  %x  %d  %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);
```

```
x.u = (((int) xL)<<16 | ((int) xR) & 0x0000ffff);
printf("%x  %x  %x  %d  %d\n", x.u, x.c[1], x.c[0], x.c[1], x.c[0]);
```

The program defines first a union structure variable `x` of the codec type. Given two 16-bit left/right numbers `xL,xR` (specified as 4-digit hex numbers), it defines a 32-bit unsigned integer `v` which is the concatenation of the two. The first `printf` statement prints the two numbers `xL,xR` in hex and decimal format. Note that the hex printing conversion operator `%x` treats the numbers as unsigned (some caution is required when printing negative numbers), whereas the decimal operator `%d` treats them as signed integers.

Next, the numbers `xL,xR` are assigned to the array members of the union `x`, such that `x.c[1] = xL` and `x.c[0] = xR`, and the second `printf` statement prints the contents of the union `x`, verifying that the 32-bit member `x.u` contains the concatenation of the two numbers with `xL` occupying the upper 16 bits, and `xR`, the lower 16 bits. Explain what the other two `printf` statements do.

Build and run the project (you may have to remove the file `vectors.asm` from the project's list of files). The output will appear in the `stdout` window at the bottom of the CCS. Alternatively, you may run this outside CCS using GCC. To do so, open a DOS window in your working directory and type the DOS command `djgpp`. This establishes the necessary environment variables to run GCC, then, run the following GCC command to generate the executable file `unions.exe`:

```
gcc unions.c -o unions.exe -lm
```

Repeat the run with the following choice of input samples:

```
xL = 0x1234;
xR = 0xabcd;
v = 0x1234abcd;
```

Explain the outputs of the print statements in this case by noting the following properties, which you should prove in your report:

$$(\texttt{0xffff0000})_{\text{unsigned}} = 2^{32} - 2^{16}$$

$$(\texttt{0xffffabcd})_{\text{unsigned}} = 2^{32} + (\texttt{0xabcd})_{\text{signed}}$$

$$(\texttt{0xffffabcd})_{\text{signed}} = (\texttt{0xabcd})_{\text{signed}}$$

1.7. Guitar Distortion Effects

In all of the experiments of Lab-2, the input/output maps are memoryless. We will study implementation of delays in a later lab. A memoryless mapping can be linear but time-varying, as was for example the case of panning between the speakers or the AM/FM wavetable experiments discussed discussed in another lab. The mapping can also be nonlinear.

Many guitar distortion effects combine delay effects with such nonlinear maps. In this part of Lab-1, we will study only some nonlinear maps in which each input sample x is mapped to an output sample y by a nonlinear function $y = f(x)$. Typical examples are hard clipping (called fuzz) and soft clipping that tries to emulated the nonlinearities of tube amplifiers. A typical nonlinear function is $y = \tanh(x)$. It has a sigmoidal shape that you can see by the quick MATLAB plot:

```
fplot('tanh(x)', [-4,4]); grid;
```

As suggested in Ref. [6], by keeping only the first two terms in its Taylor series expansion, that is, $\tanh(x) \approx x - x^3/3$, we may define a more easily realizable nonlinear function with built-in soft clipping:

$$y = f(x) = \begin{cases} +2/3, & x \geq 1 \\ x - x^3/3, & -1 \leq x \leq 1 \\ -2/3, & x \leq -1 \end{cases} \tag{1.2}$$

This can be plotted easily with

```
fplot('(abs(x)<1).*(x-1/3*x.^3) + sign(x).*(abs(x)>=1)*2/3', [-4,4]); grid;
```

The threshold value of $2/3$ is chosen so that the function $f(x)$ is continuous at $x = \pm 1$. To add some flexibility and to allow a variable threshold, we consider the following modification:

$$y = f(x) = \begin{cases} +\alpha c, & x \geq c \\ x - \beta c (x/c)^3, & -c \leq x \leq c \\ -\alpha c, & x \leq -c \end{cases}, \qquad \beta = 1 - \alpha \tag{1.3}$$

where we assume that $c > 0$ and $0 < \alpha < 1$. The choice $\beta = 1 - \alpha$ is again dictated by the continuity requirement at $x = \pm c$. Note that setting $\alpha = 1$ gives the hard-thresholding, fuzz, effect:

$$y = f(x) = \begin{cases} +c, & x \geq c \\ x, & -c \leq x \leq c \\ -c, & x \leq -c \end{cases} \tag{1.4}$$

Lab Procedure

First, run the above two `fplot` commands in MATLAB to see what these functions look like. The following program is a modification of the basic `template.c` program that implements Eq. (1.3):

```
// soft.c - guitar distortion by soft thresholding
// ------------------------------------------------------------------------------------

#include "dsplab.h"          // init parameters and function prototypes
#include <math.h>

// ------------------------------------------------------------------------------------

#define a 0.67               // approximates the value 2/3
#define b (1-a)

short xL, xR, yL, yR;        // codec input and output samples

int x, y, on=1, c=2048;      // on/off variable and initial threshold c

int f(int);                  // function declaration

// ------------------------------------------------------------------------------------

void main()                  // main program executed first
{
  initialize();              // initialize DSK board and codec, define interrupts

  sampling_rate(16);         // possible sampling rates: 8, 16, 24, 32, 44, 48, 96 kHz
  audio_source(LINE);        // LINE or MIC for line or microphone input

  while(1);                  // keep waiting for interrupt, then jump to isr()
}

// ------------------------------------------------------------------------------------

interrupt void isr()         // sample processing algorithm - interrupt service routine
{
   read_inputs(&xL, &xR);    // read left and right input samples from codec

   if (on) {
      yL = (short) f((int) xL);  yL = yL << 1;     // amplify by factor of 2
      yR = (short) f((int) xR);  yR = yR << 1;
      }
```

```
    else
       {yL = xL; yR = xR;}

    write_outputs(yL,yR);     // write left and right output samples to codec

    return;
}

// ---------------------------------------------------------------------------------

int f(int x)
{
    float y, xc = x/c;            // this y is local to f()

    y = x * (1 - b * xc * xc);

    if (x>c)  y =  a*c;           // force the threshold values
    if (x<-c) y = -a*c;

    return ((int) y);
}

// ----------------------------------------------------------------
```

a. Create a project for this program. In addition, create a GEL file that has two sliders, one for the `on` variable that turns the effect on or off in real time, and another slider for the threshold parameter `c`. Let `c` vary over the range $[0, 2^{14}]$ in increments of 512.

   Build and run the program, load the gel file, and display the two sliders. Then, play your favorite guitar piece and vary the slider parameters to hear the changes in the effect. (The wave file `turn-turn3.wav` in the directory `c:\dsplab\wav` is a good choice.)

b. Repeat the previous part by turning off the nonlinearity (i.e., setting $\alpha = 1$), which reduces to a fuzz effect with hard thresholding.

1.8. References

[1] S. J. Orfanidis, *Introduction to Signal Processing*, online book, 2010, available from:
`http://www.ece.rutgers.edu/~orfanidi/intro2sp/`

[2] R. Chassaing and D. Reay, *Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK*, 2nd ed., Wiley, Hoboken, NJ, 2008.

[3] D. R. Brown III, 2009 Workshop on Digital Signal Processing and Applications with the TMS320C6713 DSK, Parts 1 & 2, available online from:
`http://spinlab.wpi.edu/courses/dspworkshop/dspworkshop_part1_2009.pdf`
`http://spinlab.wpi.edu/courses/dspworkshop/dspworkshop_part2_2009.pdf`

[4] N. Dahnoun, "DSP Implementation Using the TMS320C6711 Processors," contained in the Texas Instruments "C6000 Teaching Materials" CD ROM, 2002-04, and available online from TI:
`http://www.ti.com/ww/cn/uprogram/share/ppt/c6000/Chapter1.ppt`
`http://www.ti.com/ww/cn/uprogram/share/ppt/c6000/Chapter2.ppt`
`http://www.ti.com/ww/cn/uprogram/share/ppt/c6000/Chapter3.ppt`

[5] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1988.

S. P. Harbison and G. L. Steele, *C: A Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 1984.

A. Kelly and I. Pohl, *A Book on C*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1990.

GNU gcc, `http://gcc.gnu.org/`
DJGPP - Windows version of GCC, `http://www.delorie.com/djgpp/`
GCC Introduction, `http://www.network-theory.co.uk/docs/gccintro/`

[6] C.R. Sullivan. “Extending the Karplus-Strong Algorithm to Synthesize Electric Guitar Timbres with Distortion and Feedback,” *Computer Music J.*, **14**, 26, (1990).