**Department of Electrical Engineering**

|  |  |
| --- | --- |
| **Faculty Member:\_\_\_\_\_Dr Ahmad Salman\_** | **Dated: \_\_\_3/13/2019\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |
|  |  |
| **Course/Section:\_\_\_\_BEE-8/D\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Semester: \_\_\_6th\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |
|  |  |

**EE330-304 Digital Signal Processing**

**Lab6: Sampling, Quantization, Aliasing using DSK**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **PLO4** | | **PLO5** | **PLO8** | **PLO9** |
| **Name** | **Reg. No** | **Viva / Quiz / Lab Performance** | **Analysis of data in Lab Report** | **Modern Tool Usage** | **Ethics and Safety** | **Individual and Team Work** |
|  |  | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** |
| **Muhammad Anas Siddiqui** | **174308** |  |  |  |  |  |
| **Haris Rehman** | **182797** |  |  |  |  |  |
| **Husnain Afridi** | **192414** |  |  |  |  |  |
|  |  |  |  |  |  |  |

**Lab6: Sampling, Quantization, Aliasing using DSK**

**Objectives**

The objective in this lab is familiarization with audio processing using DSP starter Kit

* Familiarization with Real time Audio processing using DSK
* Sampling of Audio with DSK
* Quantization with DSK

**Lab Instructions**

* The students should perform and demonstrate each lab task separately for step-wise evaluation (please ensure that course instructor/lab engineer has signed each step after ascertaining its functional verification)

* Each group shall submit one lab report on LMS within 6 days after lab is conducted. Lab report submitted via email will not be graded.

. Students are however encouraged to practice on their own in spare time for enhancing their

**Lab Report Instructions**

All questions should be answered precisely to get maximum credit. Lab report must ensure following items:

* Lab objectivesMATLAB/C codes
* Results (graphs/tables) duly commented and discussed
* Conclusion

# Sampling, Quantization, Aliasing using DSK

## 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 MHz)= 4.44 nanoseconds. During each clock cycle, up to eight instructions can be carried out in parallel, achieving up to 8×225 = 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:

fs = 8, 16, 24, 32, 44.1, 48, 96 kHz

The ADC part of the codec is implemented as a multi-bit third-order noise-shaping delta-sigma converter that allows a variety of oversampling ratios that can realize the above choices of fs. The corresponding oversampling decimation filters act as anti-aliasing prefilters that limit the spectrum of the input analog signals effectively to the Nyquist interval [−fs/2,fs/2]. The DAC part is similarly implemented as a 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 fs, as explained below.

The AIC23 codec is configured (through CCS) to operate at one of the above sampling rates fs. 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/fs. For example, if fs = 44.1 kHz, then, T= 1/fs = 22.68 μsec. With an instruction cycle time of Tc = 4.44 nsec, this allows T/Tc = 5108 cycles to be executed during each sampling instant, or, up to 8×5108 = 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]. 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:



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.

### Template Program

You will begin with a basic talk through 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().



The template has three sections. In the top section, global variables are declared and defined, such as the left/right input/output audio samples xL,xR,yL,yR, 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:\ti\common\.Besides the above three basic sections, other sections may be added that define additional functions to be called within isr() or main().

## Lab Tasks

### Lab Task 1

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=8kHz

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.

Reason:-

As it is clear from the statement that output that is produced is for 2 seconds for this sampling frequency of 8000hz have been set so if we want to hear it for 2 seconds it is 16000 samples are needed for left and right for this purpose interrupt routine so when the program runs condition is checked if samples are less than 16000 we send xl to that side it will happen until d becomes 16000 and when it is greater then all input samples are heard in right ear and when it is greater than 32000 it becomes zero .

### Lab Task 2

#### 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 fa =f−fs =5−8 =−3kHz. 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 =3kHz, the decimated sinusoid will be aliased with fa =3−4 =−1kHz.

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,... ] 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 f1 = 1kHz, f2 = 3kHz, and f3 = 1kHz, 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.

1. Next, uncomment the above line so that down sampling 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.
2. Set the codec sampling rate to 44 kHz and repeat the previous two steps. What do you expect to hear in this case?

*At we changed the fs = 44kHz, three tones can be heard in matlab.*

**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 down sampling operation, rebuild and run your program, and send this signal through the DSK, and describe what you hear.

First 1000Hz tone is heard then no sound heard and then 1000Hz tone is heard . Two 1000 hertz tones were transferred to the output but 5000Hz tone was rejected by codec so no sound was heard.

### Lab Task 3:

#### 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, [b1b2 ···b16], the corresponding 16-bit integer is given by

x=−b12−1 +b22−2 +b32−3 +···+b162−16 (1.1)

The MSB bit b1 is the sign bit. The range of representable integers is: −32768 ≤ x ≤ 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:

[b1,b2, ..., b13,b14,b15,b16] ⇒ [0,0,0,b1,b2, ..., b13] ⇒ [b1,b2, ..., b13,0,0,0]

**a**. Modify the basic template program so that the output samples are requantized to B bits, where 1 ≤ B≤ 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.

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

Explanation

In-order to understand the operation above consider that the codec has already sampled the signal and our 16 bits are as follows:-

16 bits are as follows:-

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

Now we are going to perform a re-quantization by reducing a couple of bits in the sample and trying to construct the original signal. We take the 16 bits of a signal present in xL for the sake of simplicity we are assuming that both xL and xR contain the same number of bits as mentioned above. Now we would figure out yL which involves a right shift followed by a left shift. The bits are shifted by L which is obtained from L=16-B where B is any integer.

yL after a right shifting is lets say by L=15 we get

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1] now further left shifting it with 15 we get

[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Now yR is also the same

Lets consider L=8

yL after a rightshift by 8 bits is [0 0 0 0 0 0 0 0 0 1 1 1 1 1111 ] now left shifting it we get

[1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0]

So it can be concluded that as we increase the number of bits by which we want to shift we are actually introducing more quantization noise into our signal and this is exactly that we heard when we moved from B=8 to B=1 in the above mentioned tests. Upon going from B=8 to B=1 quantization noise increased.

### Lab Task 4(just study)

#### 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 xL,xR 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-bitsshort c[2]; // left-channel = codec.c[1], right-channel = codec.c[0]*

*} codec;*

The two members of the data structure share common 32-bit memory storage. The member codec.ucontains 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*

*}*

*// --------------------------------------------------------------------------------*

Conclusion

In this Lab we got practical overview of anti aliasing ,sampling and reconstruction process