Lab 1

Sampling and Reconstruction

ECE 406: Real-Time Digital Signal Processing

February 3, 2015

Christopher Daffron

cdaffron@utk.edu

There were three objectives with this lab. The first one was to get more familiar with Code Composer Studio. The second was to learn how to configure the hardware components of the DSK board. And the third was to get a better understanding of the concepts of sampling, reconstruction, and aliasing through experimentation.

Task 1

The primary purpose of this task was to become more familiar with the features of Code Composer Studio (CCS) and to learn the process of editing source files, building the project, running the project on the C6713 processor on the DSK board, and using the debug features of CCS to monitor and manipulate the running code.

The first part of the task was to import the sample code for the sine8_LED project from the Chassaing book. This was somewhat complex due to the dependencies of the code, but I was able to get the code to compile through importing additional source files, modifying the include directory search path, and including additional library files. Once the program built, I loaded it onto the DSK board, connected the DSK board to an oscilloscope, and ran the program. The output of the oscilloscope is shown below.

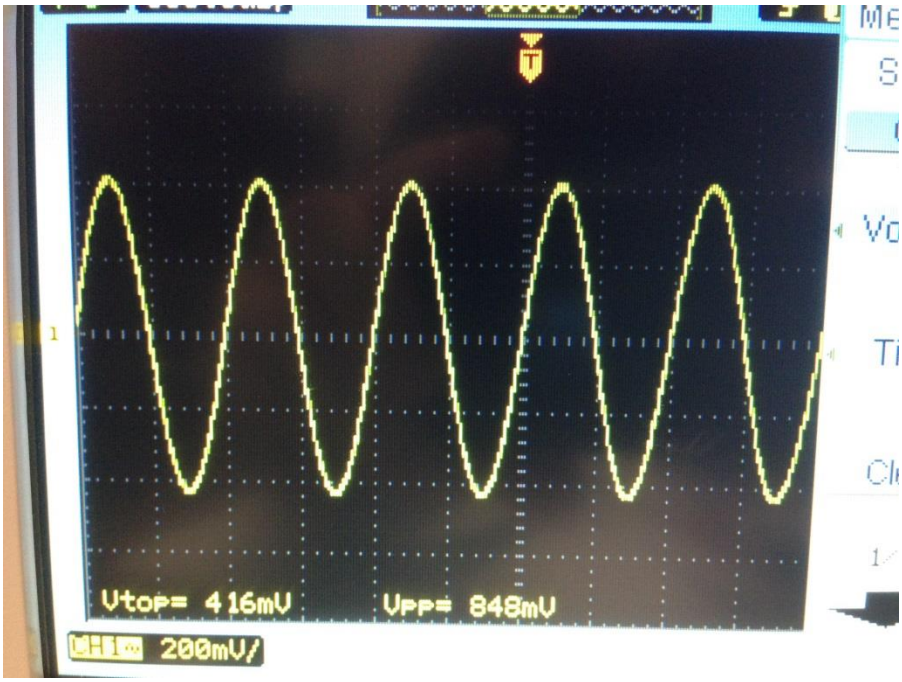

Figure 1: Initial oscilloscope output

After viewing the initial waveform, I used the watch window and then the GEL slider to modify the “gain” variable in the program. This variable caused the amplitude of the generated sine wave to change. This can be seen in the oscilloscope images because the image above shows a peak-to-peak voltage of 848 mV and the image shown below has a peak-to-peak voltage of 2.54 V.

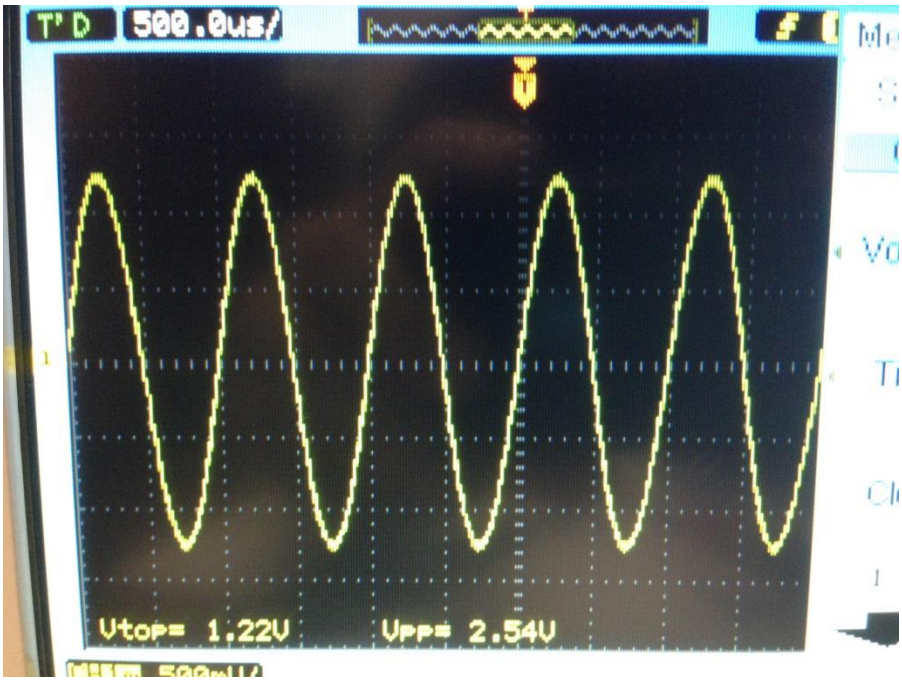

Figure 2: Oscilloscope after gain adjustment

After modifying the “gain” variable using the GEL slider, I created my own GEL slider to adjust the frequency of the sine wave by changing the increment value of the “loopindex” variable. I did this by incrementing by a variable instead of just by 1. By modifying this variable, the results shown below were generated. See incr.gel in the appendix for the slider code.

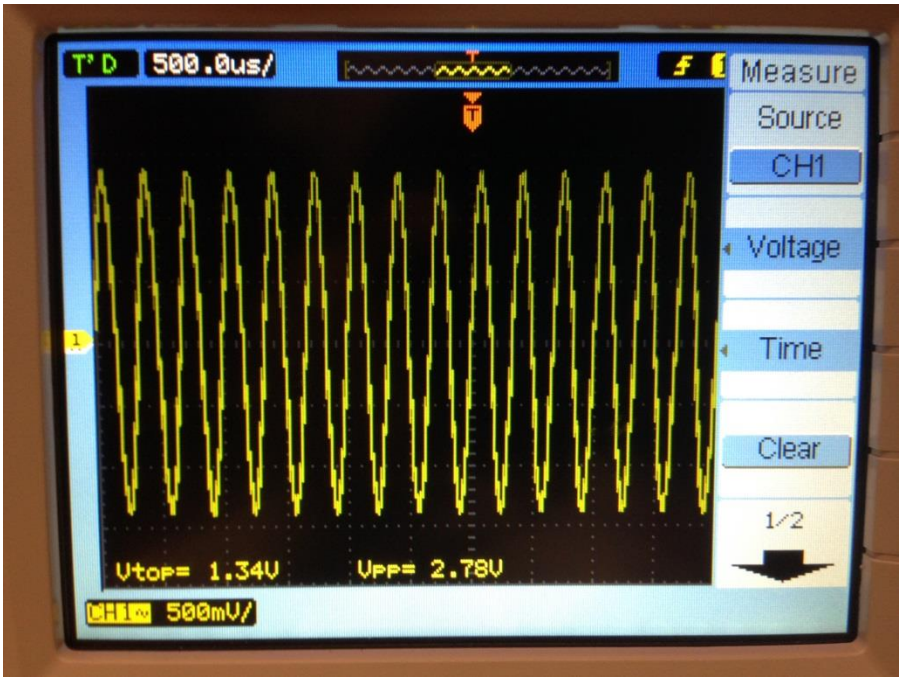

Figure 3: Oscilloscope with frequency adjustment

Finally, this task required the addition of code that would output a 500 Hz cosine wave for 5 seconds when DIP switch 3 is pressed down, instead of the 1 kHz sine wave that is normally generated. I achieved this by entering a while loop when the DIP switch is pressed that outputs the 500 Hz cosine wave for 40,000 samples before allowing normal operation to resume. The loop counts to 40,000 because the samples are produced at 8,000 samples per second and the instructions require the wave to last for 5 seconds. The third LED also turns on while the alternate wave is being generated. See the appendix for the full code that was used. The output on the oscilloscope can be seen below. Note that, because of the way an oscilloscope samples the data, the cosine wave looks the same as a sine wave at the same frequency.

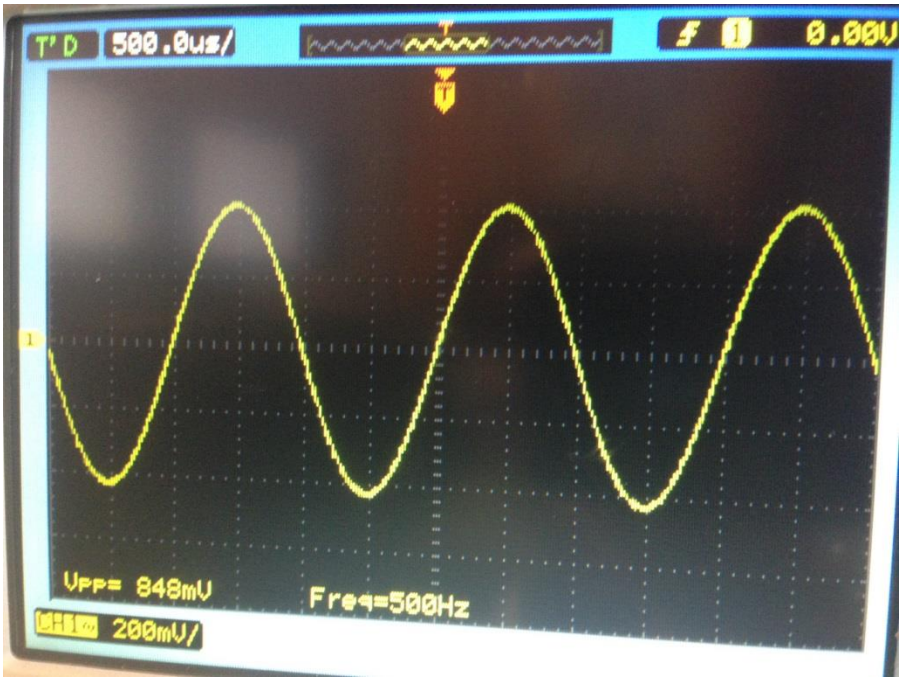

Figure 4: Oscilloscope showing 500 Hz cosine wave

Task 2

The primary purpose of this task was to learn about basic input and output using a polling-based approach. For this project, the code from loop_poll.c in Example 2.1 of Chassaing was to be used. In the same way as I did for the previous task, I imported the necessary source files, header include paths, and library files into Code Composer Studio and was successful in getting the code to compile. This code uses a polling-based approach and periodically gets a sample from the input and drives the output with it. This is done 8,000 times per second, which is an 8 kHz sampling rate. Once I loaded the code onto the DSK, I connected the line out port to the oscilloscope and the microphone input to the function generator. This is where I encountered the first problem. For some reason the incoming signal did not display correctly, with the very

top and bottom of the incoming sine wave being inverted and clipped. I think this had something to do with the resistance of the microphone port. Upon switching the input used to the line in port, the problem was fixed. Now that the signal was correctly being passed through the DSK board, I needed to use the "Graph Property Dialog" to view the signal in the "out_buffer." This presented a problem, as there is not an out_buffer in the provided code and only a single sample is kept in memory using the approach taken in the provided code. To fix this problem, I added a 512 entry buffer to the code that stores the samples as they are being sampled and outputted. To accomplish this, I referenced loop_buf.c in Chassaing's book. By doing this, the most recent 512 outputs are always kept in memory. Upon doing this, I was able to graph the recent values in CCS. See the appendix for the code used to accomplish this. Having completed the initial setup for the task, I set the signal generator to provide a 0.5 kHz sine wave and viewed it on the oscilloscope and graphed it. The results are shown below.

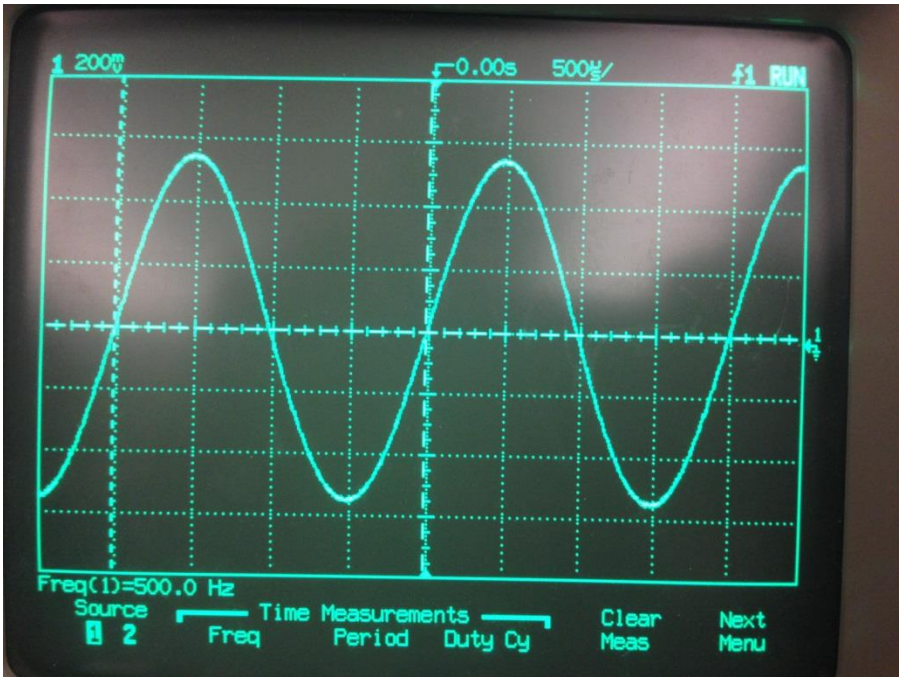

Figure 5: 0.5 kHz sine wave after going through the DSK

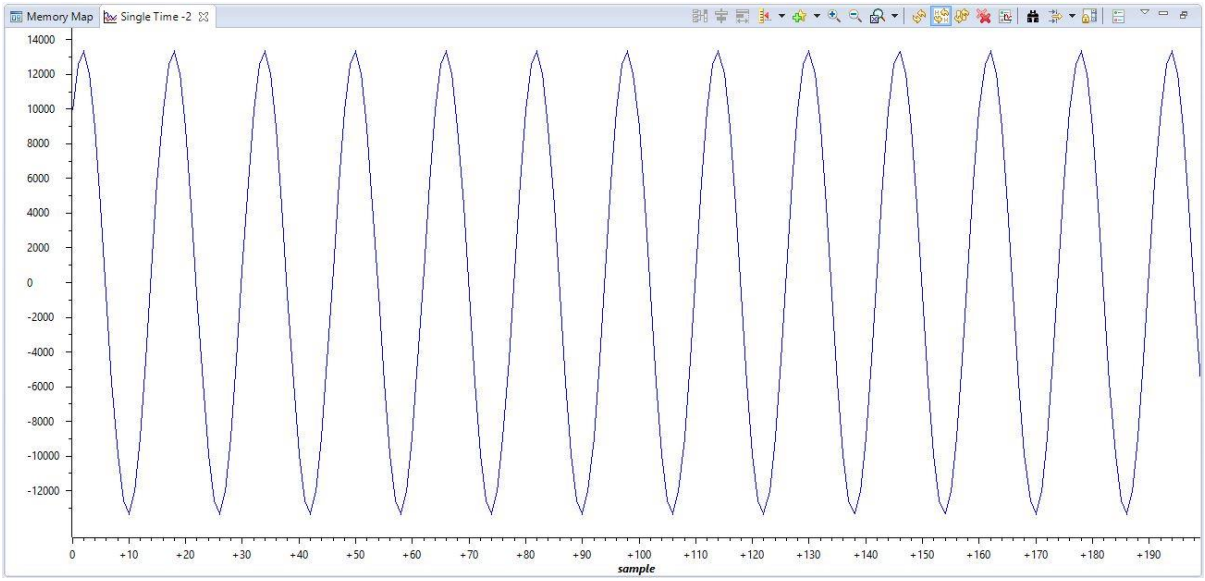

Figure 6: 0.5 kHz "out_buffer" graph in CCS

I also used the "Memory Info File" to view the contents of the "out_buffer" in table form. The table is shown below.

Variables | Expressions | Registers

Expression	Type	Value	Address
buffer	int[512]	0x000033A8	0x000033A8
[0 ... 99]			
[0]	int	-2100	0x000033A8
[1]	int	-2028	0x000033AC
[2]	int	-2091	0x000033B0
[3]	int	-1942	0x000033B4
[4]	int	-2080	0x000033B8
[5]	int	1499	0x000033BC
[6]	int	2466	0x000033C0
[7]	int	2134	0x000033C4
[8]	int	2284	0x000033C8
[9]	int	2155	0x000033CC
[10]	int	2206	0x000033D0
[11]	int	2135	0x000033D4
[12]	int	2151	0x000033D8
[13]	int	2102	0x000033DC
[14]	int	2106	0x000033E0
[15]	int	2062	0x000033E4
[16]	int	2074	0x000033E8
[17]	int	2006	0x000033EC
[18]	int	2081	0x000033F0
[19]	int	1655	0x000033F4
[20]	int	-2064	0x000033F8
[21]	int	-2300	0x000033FC
[22]	int	-2221	0x00003400
[23]	int	-2224	0x00003404
[24]	int	-2186	0x00003408
[25]	int	-2182	0x0000340C
[26]	int	-2142	0x00003410
[27]	int	-2147	0x00003414
[28]	int	-2091	0x00003418
[29]	int	-2125	0x0000341C

Table 1: Memory Info File for the 0.5 kHz sine wave

After viewing the 0.5 kHz sine wave, I changed the signal generator to output a 1 kHz sine wave. The results are shown below.

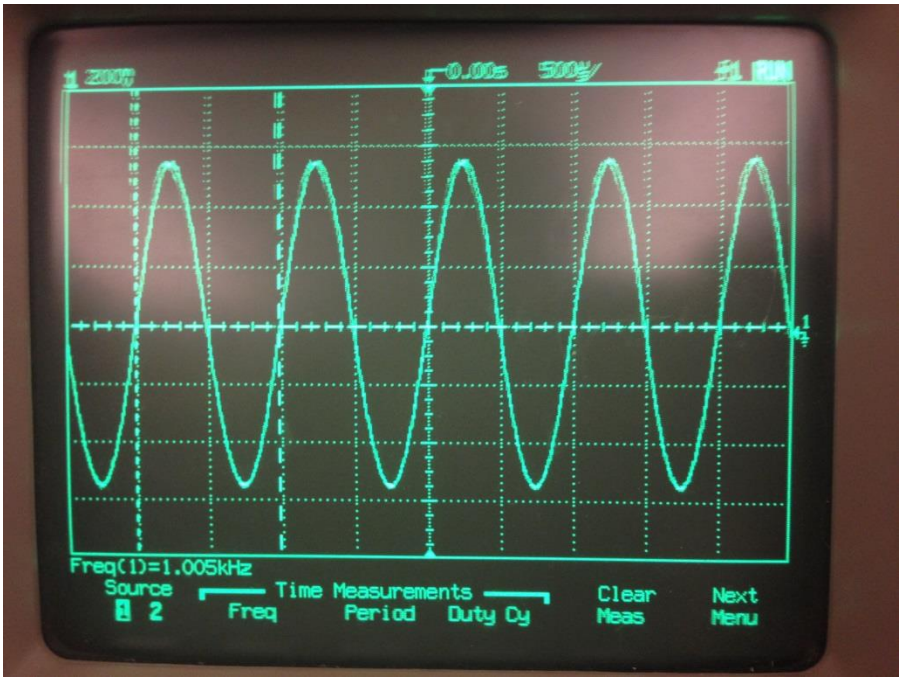

Figure 7: 1 kHz sine wave after going through the DSK

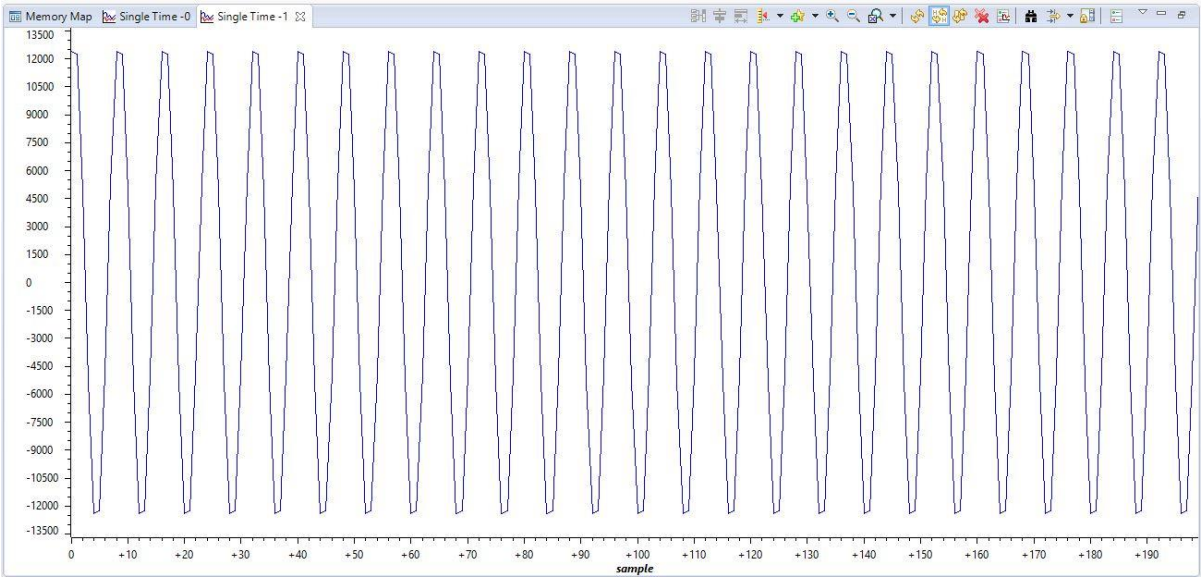

Figure 8: 1 kHz “out_buffer” graph in CCS

After viewing the 1 kHz sine wave, I changed the signal generator to output a 3.5 kHz sine wave. The results are shown below.

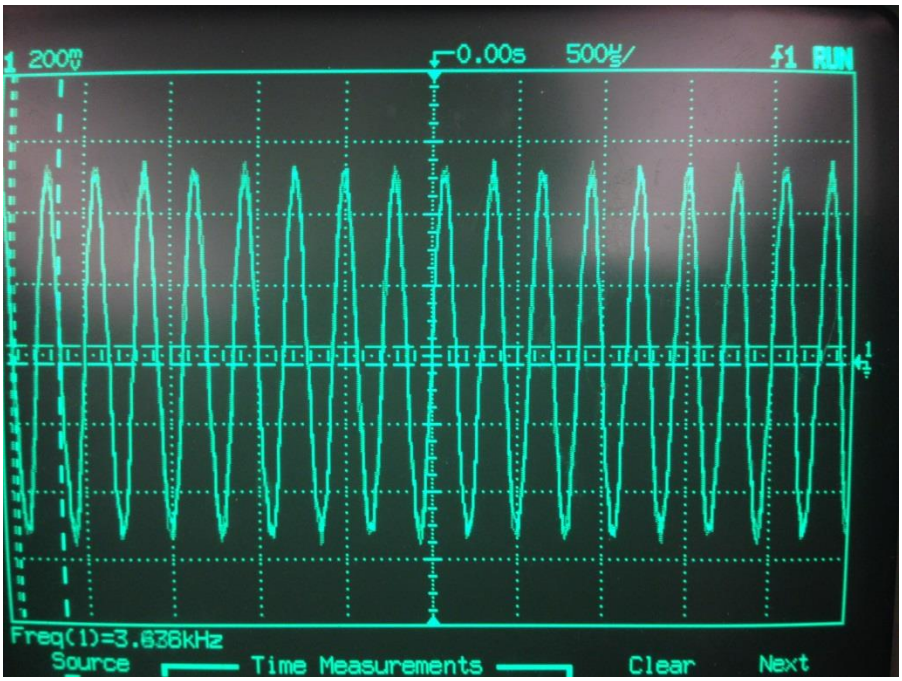

Figure 9: 3.5 kHz sine wave after going through the DSK

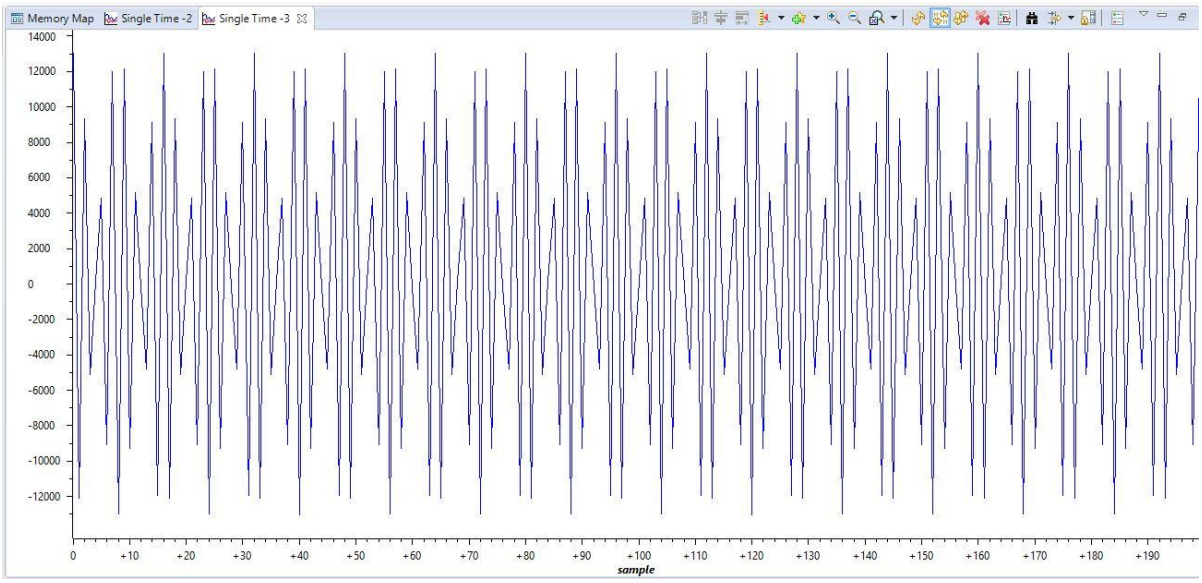

Figure 10: 3.5 kHz “out_buffer” graph in CCS

This signal is a little different than the first two ones and the difference between them can be seen especially clearly in Figure 10. The reason for this slight oscillation is that the frequency of the incoming sine wave (3.5 kHz) is almost half of the sampling frequency (8 kHz), also known as the Nyquist rate. After viewing the 3.5 kHz signal, I set the signal generator to provide a 7 kHz signal. When setting the signal generator to provide this signal, I noticed an interesting phenomenon. As the frequency gradually increased, the signal going to the oscilloscope seemed

to get gradually attenuated until nothing was visible. This is likely because of the presence of a low-pass filter in the codec. Though nothing was visible on the oscilloscope, I was still able to get a waveform in the CCS graph, though the amplitude was very small. The amplitude of the previous sine waves generally oscillated between -12,000 and +12,000, but this signal oscillates between -4 and +4. The waveform is shown below.

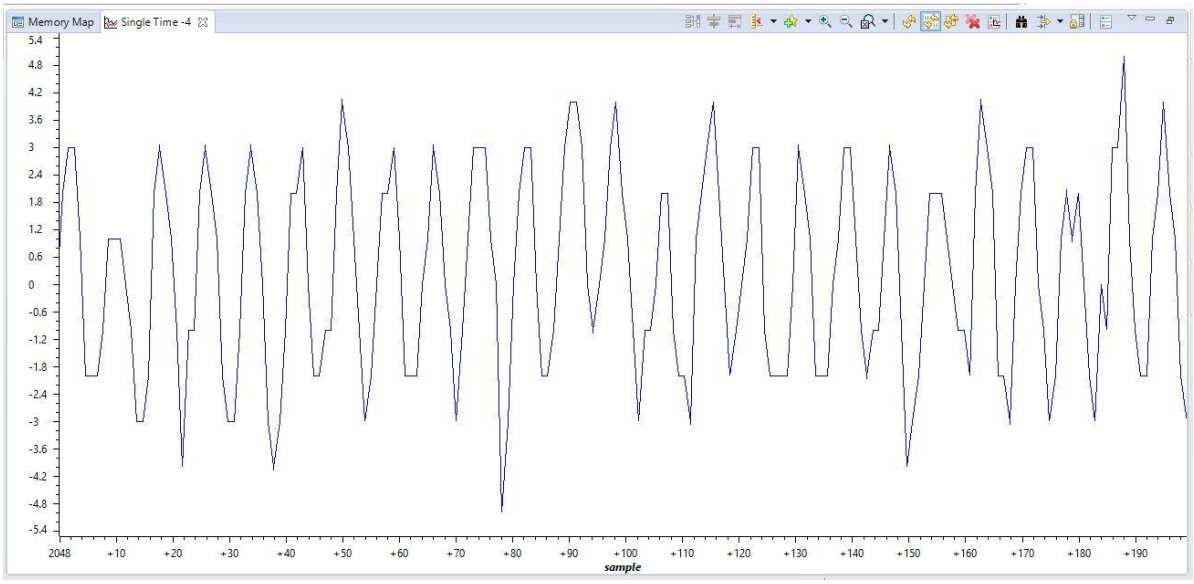

Figure 11: 7 kHz “out_buffer” graph in CCS

By counting the number of repetitions of the sine wave in the graph and cross referencing that with the number of samples and sampling rate, the frequency of the sine wave can be determined. Upon counting this, I found that there are 12.5 repetitions of the sine wave in 100 samples. The samples are taken at 8,000 samples per second. So, though some basic math, I calculated that the sine wave has a frequency of 1 kHz. This is expected because this is an example of folding and the viewed sine wave is generated from the negative part of the spectrum. Once I finished with the 7 kHz signal, I increased the frequency of the signal generator to 9 kHz. This signal had the same problems as the 7 kHz signal with no results visible on the oscilloscope but with a waveform visible in the CCS graph window. The waveform is shown below.

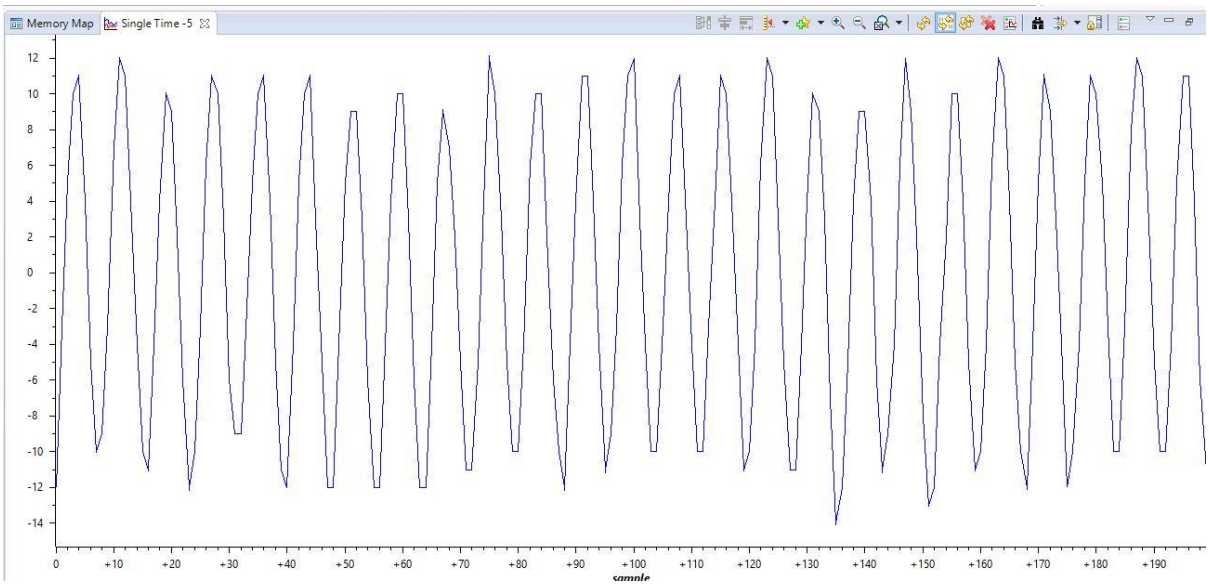

Figure 12: 9 kHz “out_buffer” graph in CCS

Using the same method as was described for the 7 kHz signal, I counted the repetitions of the sine wave to determine its frequency. This sine wave also has 12.5 repetitions in 100 samples and was also collected with a sampling rate of 8,000 samples per second, so the frequency is 1 kHz. This is expected because the input signal is 1 kHz different than the sampling rate and is an example of aliasing and the viewed sine wave is generated from the positive part of the spectrum. Once I was finished analyzing the 9 kHz input signal, I set the signal generator to provide a square wave at 270 Hz with an amplitude of 0.2 V. The results of this are shown below.

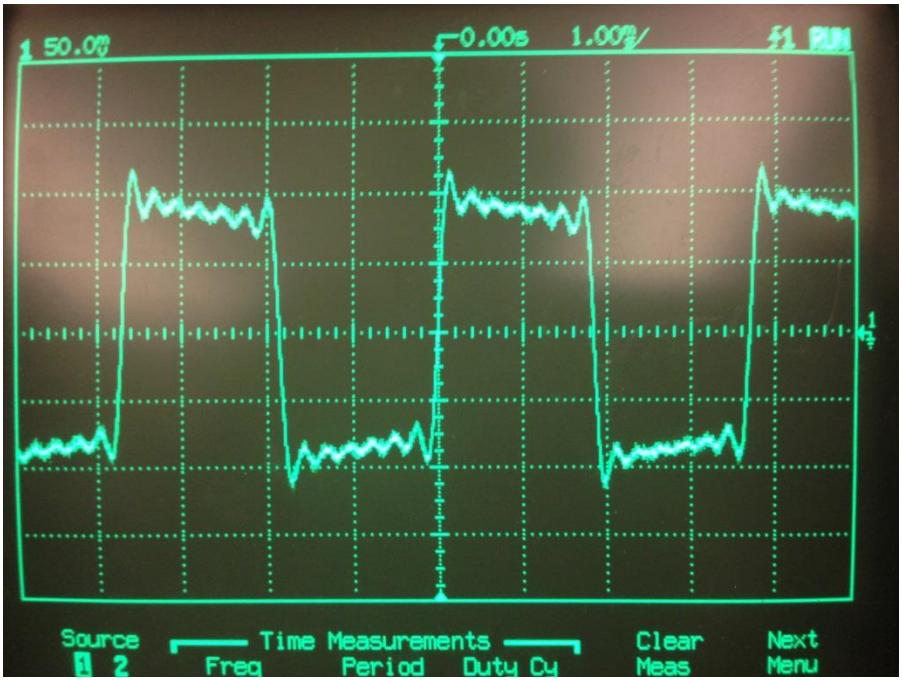

Figure 13: Square wave after going through the DSK

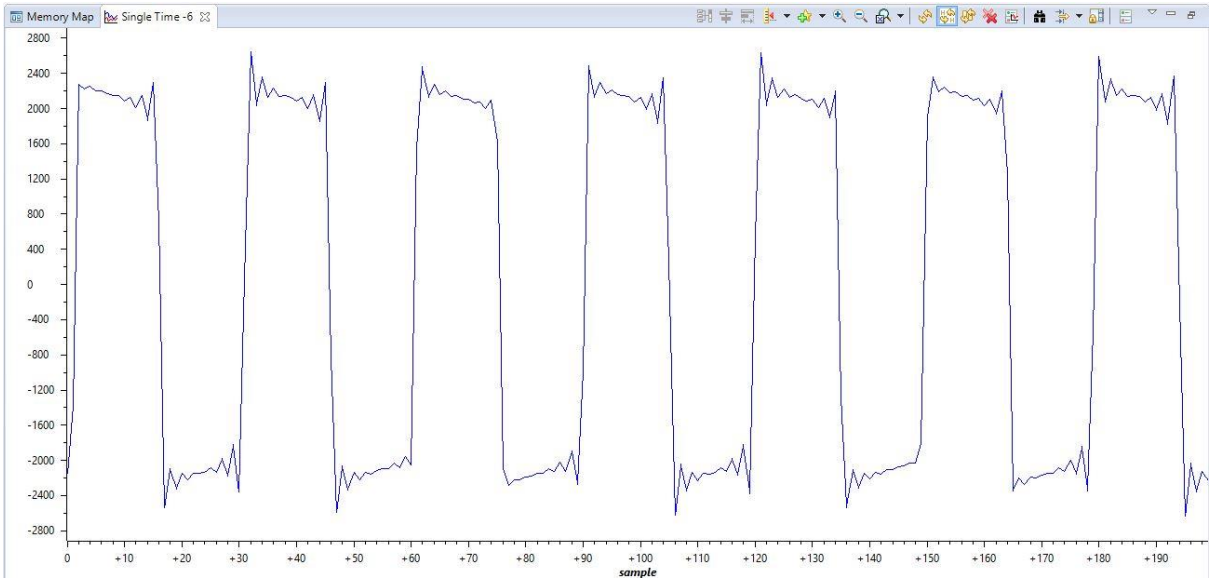

Figure 14: Square wave “out_buffer” graph in CCS

This signal has an interesting effect in the oscillation in what should be the constant value at the top and bottom of the square wave. This effect is caused by the frequency characteristics of the sampling. The vertical parts of the square wave immediately after a constant value are like an instantaneous infinite frequency that then immediately becomes a zero frequency when the value is constant. Because of this, artifacts are introduced into the resulting signal, resulting in the oscillation seen.

Conclusion

In this lab, I learned about how to use the various features of Code Composer Studio, how to configure and work with the DSK hardware, and I improved my understanding of sampling, reconstruction, and aliasing. After completing this lab, I feel like I greatly improved my understanding of how to program the DSK and how to write code that utilizes the various API calls that are available to interact with the DSP chip, the codec, and the various I/O components on the board. I also improved my understanding of how the DSK samples and reproduces data and how aliasing can affect the resulting signal.

Appendix

Task 1

Sine8_LED.c

```
//sine8_LED.c  sine generation with DIP switch control
```

```
#include "dsk6713_aic23.h"          //codec support
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ;   //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_MIC;//select input
#define LOOPLENGTH 8
short loopindex = 0;               //table index
short loopincr = 1;                                  //loop increment value
short gain = 10;                               //gain factor
int timeCounter = 0;
short alternate = 0;
short sine_table[LOOPLENGTH]={0,707,1000,707,0,-707,-1000,-707}; //sine values
short sine_table2[LOOPLENGTH*2]={1000,924,707,383,0,-383,-707,-924,-1000,-924,-707,-
383,0,383,707,924}; //cosine values

void main()
{
  comm_poll();                     //init DSK,codec,McBSP
  DSK6713_LED_init();                     //init LED from BSL
  DSK6713_DIP_init();                     //init DIP from BSL
  while(1)                                        //infinite loop
  {
       //generate 500 Hz cosine wave when DIP3 is pressed
       if(DSK6713_DIP_get(3)==0)
       {
              DSK6713_LED_on(3);    // turn on LED3
              DSK6713_LED_off(0); // turn off other LEDs
              timeCounter = 0;
              alternate = 0;
              loopindex = 0;
              while(timeCounter < 40000)
              {
                     output_left_sample(sine_table2[loopindex++]*gain);
                     alternate++;
                     if(loopindex >= LOOPLENGTH*2) loopindex = 0;
                     timeCounter++;
              }
              DSK6713_LED_off(3);
       }
       else if(DSK6713_DIP_get(0)==0)               //=0 if DIP switch #0 pressed
       {
              DSK6713_LED_off(3);
              DSK6713_LED_on(0);                         //turn LED #0 ON
              output_left_sample(sine_table[loopindex+=loopincr]*gain); //output sample
              if (loopindex >= LOOPLENGTH) loopindex = 0; //reset table index
       }
       else
       {
              DSK6713_LED_off(0);            //turn LED off if not pressed
              DSK6713_LED_off(3);
       }
  }                                  //end of while(1) infinite loop
}                                                //end of main
```

incr.gel

```
/*incr.gel GEL slider to vary frequency of sinewave*/
/*generated by program sine8_LED.c*/
```

```
menuitem "Sine Frequency"

slider Increment(0,5,1,1,incr_parameter) /*incr by 1, up to 5*/
{
        loopincr = incr_parameter;       /*vary increment of table*/
}
```

Task 2

loop_poll.c

```
//loop_poll.c loop program using polling
#include "DSK6713_AIC23.h"           //codec support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;   //set sampling rate
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE; // select input
short sample_data;

#define BUFSIZE 512 // size of the buffer
int buffer[BUFSIZE]; // buffer to hold recent samples
int buf_ptr = 0; // pointer into the buffer

void main()
{
  //short sample_data;
  comm_poll();             //init DSK, codec, McBSP
  while(1)                              //infinite loop
  {
    sample_data = input_left_sample(); //input sample
    buffer[buf_ptr] = sample_data; //store sample in buffer
    if(++buf_ptr >= BUFSIZE) buf_ptr = 0; //check pointer
    output_left_sample(sample_data);   //output sample
  }
}
```