RTDSP Lab 2

Yong Wen Chua (ywc110)

1 Questions

1.1 Question 1

Consider the following table for the values generated by `sinegen()`:

Loop	Sample
1	$\frac{\sqrt{2}}{2}$
2	1
3	$\frac{\sqrt{2}}{2}$
4	0
5	$-\frac{\sqrt{2}}{2}$
6	-1
7	$-\frac{\sqrt{2}}{2}$
8	0

By inspection, it will take the loop eight times to generate one complete cycle.

1.2 Question 2

The audio ports sample at 8 KHz, taking a sample every 125 μs. Since it takes the loop eight times to complete a cycle, the audio ports will get a complete cycle every $125 \times 8 = 1000\mu s$. This translates to a 1 KHz sine wave being generated.

Thus, it is the sample rate of the audio port (via the blocking `while` loops in the infinite `while` loop in `main()`) that throttles the sine wave being generated at 1 KHz.

1.3 Question 3

The samples are encoded as 32 bits integers to the audio port (via the `Int32` cast in the code).

2 Code

2.1 Code Operation

2.1.1 Sine Table Generation

```c
void sine_init(void){
  int i;
  for (i = 0; i < SINE_TABLE_SIZE; ++i)
    table[i] = sin(i * (2*PI/SINE_TABLE_SIZE));
}
```

The function to generate the sine table uses a `for` loop to loop around `SINE_TABLE_SIZE` number of times to generate the value using the `sin()` function. The phase is calculated at each iteration as shown.

2.1.2 Sine Wave Generation

The `sinegen()` function, by keeping track of the "phase" of the current wave, will return the value to allow for the generation of the sine wave in the appropriate frequency. It keeps track of the "phase" of the wave using a `static` variable `index`. The static variables `prev_freq` and `prev_sample` are used top check for changes in those settings. If either are changed, the `index` is reset to zero, and the values values described below will be recalculated.

The first value determines the number of samples necessary to generate the entire wave using the following line:

```c
double cycleSampleCount = (double) sampling_freq / (double) sine_freq;
```

The number of entries in the sine table to skip each time a sample is required is then calculated using the line, along with an increment of the `index`:

```c
double step = (double) SINE_TABLE_SIZE/(double) cycleSampleCount;
//... other code ...
index += step;
```

To ensure that we do not exceed the number of entries in the table and cause a segmentation fault, the following line will reset the index with the necessary offset for the next cycle to ensure a smoother wave and allows the generation of "odd" frequencies:

```c
if ((int) index >= SINE_TABLE_SIZE)
    index -= SINE_TABLE_SIZE;
```

Finally, the value will be retrieved from the table and returned. The `round()` is necessary as the table only has discrete index.

```c
return table[(int) round(index)];
```

2.1.3 Increasing the Resolution

There are two ways to increase the resolution.

Firstly: The `round()` function used on the index in the code above will lead to "steps" in the wave generated, as will be described in section 2.3. The resolution can be increased, without using a larger look-up table, by interpolating the values between the "quantised" indices of the look-up table.

Let i be the index we are trying to read from the table. i may, or may not be an integer. Let the function $table(x)$ return the value of the table at index x where $x \in \mathbb{Z}$. Then the value v to be returned from the table can be determined as follows:

$$v = table(\lfloor i \rfloor) + (i - \lfloor i \rfloor) \times [table(\lceil i \rceil) - table(\lfloor i \rfloor)]$$

This is a linear interpolation between the two "quantised" indices of the lookup table.

Secondly: Since the values in the sine wave are basically the same in each quarter of a cycle, appropriate sign change can be used to get the necessary values. Thus, the code could be modified such that only the first quarter of the wave is stored in the sine look-up table. In this way, the resolution of the wave can be increased by four-fold. This technique is employed in the code listing in section §3 and controlled using the macro `INCREASE_RES`.

The sine table is first changed to generate only one quadrant of the wave:

```
void sine_init(void){
  int i;
  for (i = 0; i < SINE_TABLE_SIZE; ++i)
        table[i] = sin(i * ( (2*PI) /(RES_MULTIPLIER*SINE_TABLE_SIZE)));
}
```

The macro `RES_MULTIPLIER` is set to a value of '4'. The code operation in this version is the similar to what is described in 2.1.2.

The only difference is that the index is allowed to go up to four times the size of `SINE_TABLE_SIZE`. Then the value is not read directly from the array, but by using a call to the function as given below:

```
return sine_value((int) round(index));
```

The `sine_value` function works by first determing the quadrant of a sine wave in which the index we want to retrieve is at, and also calculates the "progress" in that specific quadrant:

```
int quadrant = index/SINE_TABLE_SIZE;  // the quadrant in which the cycle is in
int modulo = index % SINE_TABLE_SIZE;   // the modulo
```

Then, according to the quadrant the index is in, the appropriate value is read from the table array and adjusted accordingly.

```
if (quadrant == 0)
  value = table[index];
else if (quadrant == 1)
  value =  table[SINE_TABLE_SIZE-modulo-1];
else if (quadrant == 2)
  value =  table[modulo]*-1;
else if (quadrant == 3)
  value =  table[SINE_TABLE_SIZE-modulo-1]*-1;
else
  value =  0;
```

2.2 Traces

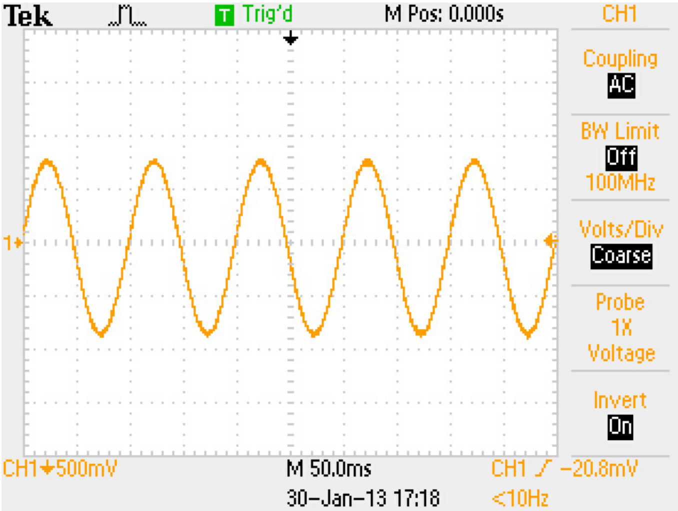

Figure 1: Sampling frequency at 8 KHz; sine frequency at 10 Hz

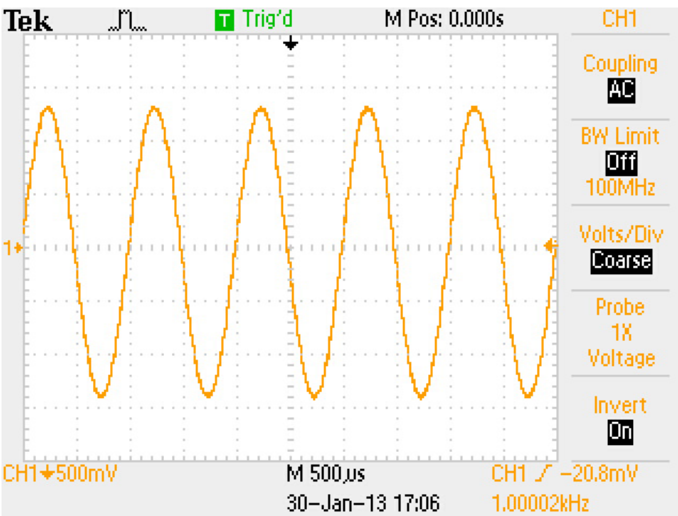

Figure 2: Sampling frequency at 8 KHz; sine frequency at 1 KHz

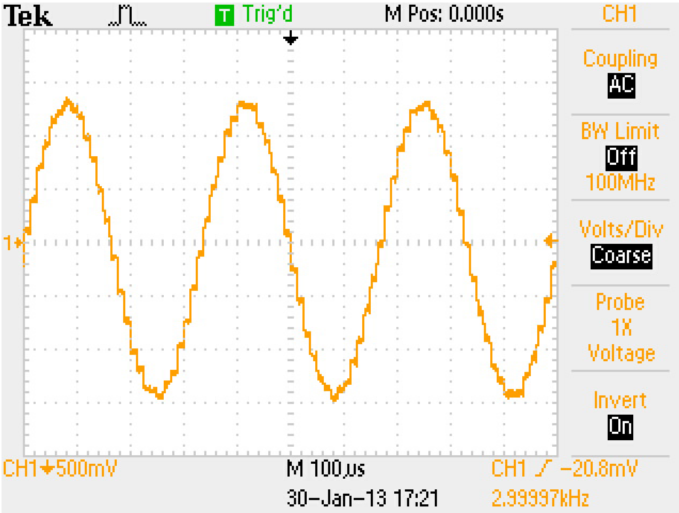

Figure 3: Sampling frequency at 8 KHz; sine frequency at 3 KHz

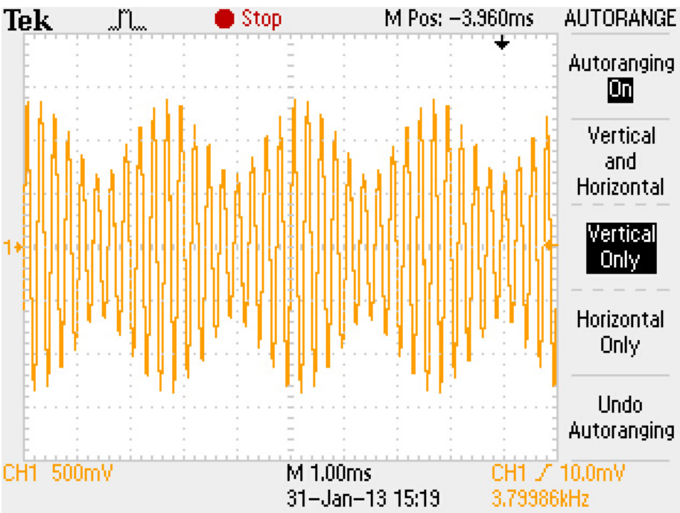

Figure 4: Sampling frequency at 8 KHz; sine frequency at 3.8 KHz

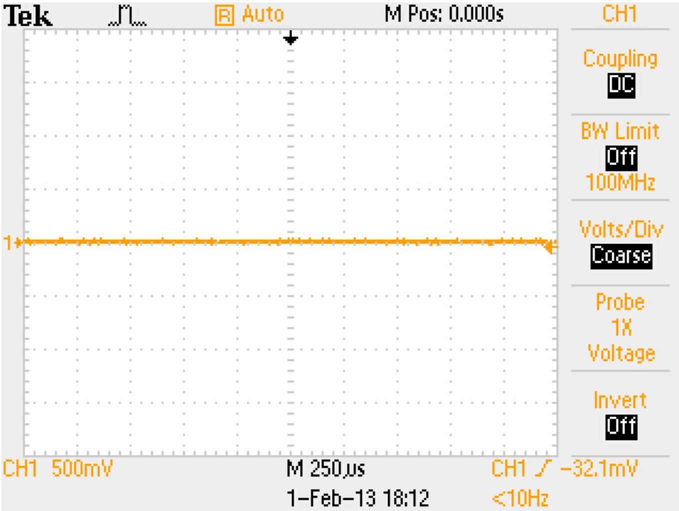

Figure 5: Sampling frequency at 8 KHz; sine frequency at 4 KHz; flat line, as expected

2.3 Limitations

2.3.1 Smoothness

Consider the number of samples required to generate the sine wave as explained in section 2.1.2 and let this be n. The smoothness of the wave generated will be at its best if n is a factor of `SINE_TABLE_SIZE`. This is because the code described in section 2.1.2 will round the index i to be generated to the nearest integer. Thus, it is possible that two "steps" of the wave generation will result in the same value being retrieved from the table, resulting in the "steps" seen in the output wave. If the methods described in section 2.1.3 is employed, then the wave would be smoother.

2.3.2 Upper Frequency Bound

According to Nyquist Sampling Theory, the maximum frequency that can be generated would be half of the sampling frequency of the audio port. This is confirmed as it is observed that the DSP would generate "incorrect" frequencies when it is asked to generate frequencies above the Nyquist frequency as a result of aliasing. Sampling in time domain is essentially equivalent to duplicating the frequency spectrum contents. If the frequency of the wave generated is too high, this results in an overlap in the frequency domain, leading to signal corruption. If the frequency is set at exactly the Nyquist frequency, a flat wave (see figure 5) would be generated as the table would be read at the start and the middle, which both have values of zero.

It should be noted that if the frequency change check used when resetting the index as described in section 2.1.2 is not implemented, a wave at exactly the Nyquist frequency can be generated. This is possible by exploiting the "stray" offsets left behind through the previous generation of waves in other frequencies. However, this causes an element of uncertainty in the wave generation (as it will depend on whatever has been generated before) and the frequency change check is implemented to remove this element of uncertainty.

As the frequency of the wave generated approaches the Nyquist frequency, the amplitude of the wave generated will oscillate (see figure 4). This is because of the non-ideal nature of the sine wave generated (as a result of the "steps" as described in.section 2.3.1). Thus, the wave would have frequency components other than the one we are trying to generate. This is equivalent to having an amplitude modulation in the time domain, resulting in what we can observe in figure 4.

2.3.3 Lower Frequency Bound

Theoretically, according to the implementation described in section 2.1.2 due to the use of `round()` in the index of the table to be read, there should be no lower frequency bound of the wave that can be generated. A very low frequency will simply result in many samples required per wave, leading to the "steps" described in section 2.3.1, which can be mitigated by using the techniques described in section 2.1.3. It has been observed that a decent wave can be generated at 10 Hz and a wave is also generated at 1 Hz. However, the line out port attenuates the amplitude of lower frequencies (having a cut-off frequency of approximately 7.2 Hz by calculation from the 470μF capacitor and 47$k\Omega$ resistor used), thus limiting the generation of sine waves at lower frequencies.

3 Code Listing

The code listing has some slight differences with the code described in previous sections due to the implementation of the resolution technique described in section 2.1.3.

```c
/*************************************************************************************
            DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                        IMPERIAL COLLEGE LONDON

             EE 3.19: Real Time Digital Signal Processing
                Dr Paul Mitcheson and Daniel Harvey

               LAB 2: Learning C and Sinewave Generation

                        ********* S I N E . C **********

             Demonstrates outputting data from the DSK's audio port.
         Used for extending knowledge of C and using look up tables.

 *************************************************************************************
         Updated for use on 6713 DSK by Danny Harvey: May-Aug 06/Dec 07/Oct 09
       CCS V4 updates Sept 10
 *************************************************************************************/
/*
 *  Initialy this example uses the AIC23 codec module of the 6713 DSK Board Support
 *  Library to generate a 1KHz sine wave using a simple digital filter.
 *  You should modify the code to generate a sine of variable frequency.
 */
/************************** Pre-processor statements *****************************/

//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

// Some functions to help with configuring hardware
#include "helper_functions_polling.h"
```

```c


// PI defined here for use in your code
#define PI 3.141592653589793

// The number of entries in the sine lookup table
#define SINE_TABLE_SIZE 256

// Define this to allow for increase of resoltuion
#define INCREASE_RES

// Multiplier depending on whether INCREASE_RES is set
#ifdef INCREASE_RES
    #define RES_MULTIPLIER 4
#else
    #define RES_MULTIPLIER 1
#endif


/****************************** Global declarations ******************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
       /*******************************************************************/
       /*   REGISTER              FUNCTION              SETTINGS          */
       /*******************************************************************/\
    0x0017,  /* 0 LEFTINVOL   Left line input channel volume  0dB                   */\
    0x0017,  /* 1 RIGHTINVOL  Right line input channel volume 0dB                   */\
    0x01f9,  /* 2 LEFTHPVOL   Left channel headphone volume   0dB                   */\
    0x01f9,  /* 3 RIGHTHPVOL  Right channel headphone volume  0dB                   */\
    0x0011,  /* 4 ANAPATH     Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH     Digital audio path control      All Filters off       */\
    0x0000,  /* 6 DPOWERDOWN  Power down control              All Hardware on       */\
    0x004f,  /* 7 DIGIF       Digital audio interface format  32 bit                */\
    0x008d,  /* 8 SAMPLERATE  Sample rate control             8 KHZ                 */\
    0x0001   /* 9 DIGACT      Digital interface activation    On                    */\
       /*******************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

/* Sampling frequency in HZ. Must only be set to 8000, 16000, 24000
32000, 44100 (CD standard), 48000 or 96000  */
int sampling_freq = 8000;

// Holds the value of the current sample
float sample;

/* Left and right audio channel gain values, calculated to be less than signed 32 bit
 maximum value. */
Int32 L_Gain = 2100000000;
Int32 R_Gain = 2100000000;


```

```
/* Use this variable in your code to set the frequency of your sine wave
   be carefull that you do not set it above the current nyquist frequency! */
float sine_freq = 1000.0;

/* The array to hold the values of the sine lookup table */
float table[SINE_TABLE_SIZE] = {0};

/****************************** Function prototypes *******************************/
void init_hardware(void);
float sinegen(void);
void sine_init(void);    // Remember the void in the function parameters because this is C!
#ifdef INCREASE_RES
  float sine_value(int);
#endif
/********************************* Main routine **********************************/
void main()
{

  // initialize board and the audio port
  init_hardware();

  // initialise sine table
  sine_init();

    // Loop endlessley generating a sine wave
   while(1)
    {
    // Calculate next sample
    sample = sinegen();

      /* Send a sample to the audio port if it is ready to transmit.
           Note: DSK6713_AIC23_write() returns false if the port if is not ready */

        //  send to LEFT channel (poll until ready)
        while (!DSK6713_AIC23_write(H_Codec, ((Int32)(sample * L_Gain))))
        {};
    // send same sample to RIGHT channel (poll until ready)
        while (!DSK6713_AIC23_write(H_Codec, ((Int32)(sample * R_Gain))))
        {};

    // Set the sampling frequency. This function updates the frequency only if it
    // has changed. Frequency set must be one of the supported sampling freq.
    set_samp_freq(&sampling_freq, Config, &H_Codec);

  }

}

/****************************** init_hardware() **********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

```

```c
  /* Defines number of bits in word used by MSBSP for communications with AIC23
   NOTE: this must match the bit resolution set in in the AIC23 */
  MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);

  /* Set the sampling frequency of the audio port. Must only be set to a supported
     frequency (8000/16000/24000/32000/44100/48000/96000) */

  DSK6713_AIC23_setFreq(H_Codec, get_sampling_handle(&sampling_freq));

}

/******************************** sinegen() ************************************/
float sinegen(void){
    static double index = 0;  // Store the "progress" of the sine wave generation
    static double prev_freq = 0;  // Previous frequency
    static double prev_sample = 0;  // Previous sampling frequency
    // Based on sampling frequency and sine frequency, determine number of samples per cycle
    static double cycleSampleCount;
    // Determine the number of intervals to "skip" each time we proceed to the next stage of the sine wave generation
    static double step;

    if (prev_freq != sine_freq || prev_sample != sampling_freq){    // If frequency has changed, we should take note.
        index = 0;
        prev_freq = sine_freq;
        prev_sample = sampling_freq;
        cycleSampleCount = (double) sampling_freq / (double) sine_freq;
        step = (double) (SINE_TABLE_SIZE*RES_MULTIPLIER)/(double) cycleSampleCount;
    }
    index += step; // "advance" to the next step

    // Check that index is in range
    if ((int) index >= SINE_TABLE_SIZE*RES_MULTIPLIER)
            index -= (SINE_TABLE_SIZE*RES_MULTIPLIER);    // Reset with an offset so that we can try and generate more frequencies

    #ifndef INCREASE_RES
      return table[(int) round(index)]; // "read" from the table;
    #else
      return sine_value((int) round(index));
    #endif
}

/******************************** sine_init() ************************************/
void sine_init(void){
  int i;

  for (i = 0; i < SINE_TABLE_SIZE; ++i)
        table[i] = sin(i * ( (2*PI) /(RES_MULTIPLIER*SINE_TABLE_SIZE)));
}

#ifdef INCREASE_RES
  float sine_value(int index){
      int quadrant = index/SINE_TABLE_SIZE;    // the quadrant in which the cycle is in
      int modulo = index % SINE_TABLE_SIZE;    // the modulo
```

```
    float value;
    if (quadrant == 0)
      value = table[index];
    else if (quadrant == 1)
      value =  table[SINE_TABLE_SIZE-modulo-1];
    else if (quadrant == 2)
      value =  table[modulo]*-1;
    else if (quadrant == 3)
      value =  table[SINE_TABLE_SIZE-modulo-1]*-1;
    else
      value =  0;

      return value;
  }
#endif
```