1.Questions

1. **Provide a trace table of Sinegen for several loops of the code. How many samples does it have to generate to complete a whole cycle?**

   Answer: 8 samples per period, tested using variable watching and calculation by hand.

2. **Can you see why the output of the sinewave is currently fixed at 1 kHz? Why does the program not output samples as fast as it can? What hardware throttles it to 1 kHz? (If you are having problems working this out try changing the sampling frequency[2] by changing sampling_freq).**

   Answer: The sampling frequency is set to 8khz and the number of samples per period is 8. 8k/8 = 1 k. As described in the lecture, the hardware DAC (periphal) constaints the clock rate at 1 kHz, regardless of how fast the processing units calculate.

3. **By reading through the code can you work out the number of bits used to encode each sample that is sent to the audio port?**

   Answer: 32 bits as defined in the source code.

2.Operation of Code

To successfully generate a sine wave, values of sine functions at different points are needed. Therefore, an array called *table* is defined, size of which depends on variable *SINE_TABLE_SIZE.*

The first step of our code is to initialize hardware (as detailed by the original code) and array variable table. The library function sin() is able to return a double value of sine wave with increment of 2*PI/SINE_TABLE_SIZE in radian. Nexct, the iteration of for loop will fill the array table with values of sine wave.

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

The second step is to generate an actual sine wave by placing the sinegen() function in the while loop of main function. While called each time, the sinegen () function returns the next value following the preceding one generated by the previous siniegen (). This memory capability is achieved with a global variable index and this variable is incremented by the value of (SINE_TABLE_SIZE/(sampling_freq/sine_freq)), which equals to the exact index increment needed to achieve a wave with specific sampling frequency and sine wave frequency. Therefore, this increasing variable allows sinegen () to access to correct element of a sine wave and return the respective value to the handle in main ().

```
float sinegen(void)
{
///*  This code produces a fixed sine of 1KHZ (if the sampling frequency is 8KHZ)
//    using a digital filter.
//  You will need to re-write this function to produce a sine of variable frequency
//  using a look up table instead of a filter.*/
//
//  // temporary variable used to output values from function
    float wave;
//
//  // represents the filter coefficients (square root of 2 and 1/square root of 2)
//  float a0 = 1.4142;
//  float b0 = 0.7071;
//
//  y[0] = a0 * y[1] - y[2] + b0 * x[0]; // Difference equation
//
//  y[2] = y[1]; // move values through buffer
//  y[1] = y[0];
//
//  x[0] = 0; // reset input to zero (to create the impulse)
//
//  wave = y[0];
    wave = table[(int)index];
    // reads value from LUT
    index+=(float)(SINE_TABLE_SIZE/(sampling_freq/sine_freq));
    // increment index with respect to the target sine frequency
    if (index>SINE_TABLE_SIZE){
        index -= SINE_TABLE_SIZE;
    }
    // handle wrap around case
    // Alternatively, use code below to increase speed
    //      index &= SINE_TABLE_SIZE-1
    // not need to use if statement
    return(wave);
```

Moreover, if we wish to increase the resolution (the smoothness of sine wave), the most obvious method would be to increase sampling rate, which is equivalent to reducing the step size on x-axis. Besides, a cleverer idea is to exploit the symmetricity of sine wave. It is well-known that the sine wave is symmetrical with respect to x-axis. Therefore, knowing the upper side of sine wave is equivalent to knowing the lower side of sine wave. Similarly, knowing the left side of upper sine wave is equivalent to knowing the right side of upper sine wave. With this idea, some manipulations on accessing the look-up table can achieve storing 256 values but effectively storing 256*4 values. However, it should be noted that this mathematical trick increases the computational complexity significantly. It is worthwhile when the memory space is insufficient but higher resolution is demanded.

3.Bounds of frequency

Lower Bound: Due to the restrictions on sampling frequency, the lowest sampling frequency that can be set is 8000Hz. In this case, according to formula $wave\ frequency = \frac{\text{sampling frequency}}{\text{no.of samples}}$, the lower bound of wave frequency is 31.25Hz (8000Hz/256). Once the wave frequency is lower than 31.25Hz, the number of samples needed to maintain sampling frequency is higher. However, the program specifies that only 256 values are available, meaning that some values will be wrongly accessed more than 1 time. As a result, the sine wave generated is not as expected.

Upper Bound: According to the Nyquist theorem, the sampling frequency should be greater than twice the signal frequency. Therefore, given that the highest sampling frequency supported on this system is 96000Hz, the upper bound of our signal frequency is 48000Hz. If the wave frequency goes beyond this threshold, aliasing will occur and signal will therefore be corrupted.

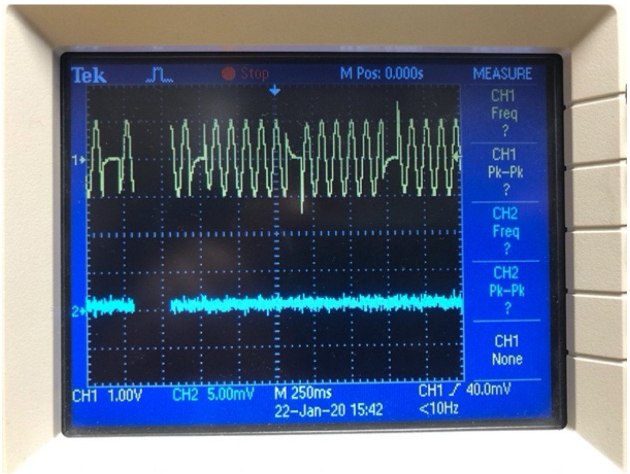
4.Scope Trace

The following pictures show the scope trace between 10HZ to 4000HZ (the Nyquist frequency for 8000Hz sampling frequency).

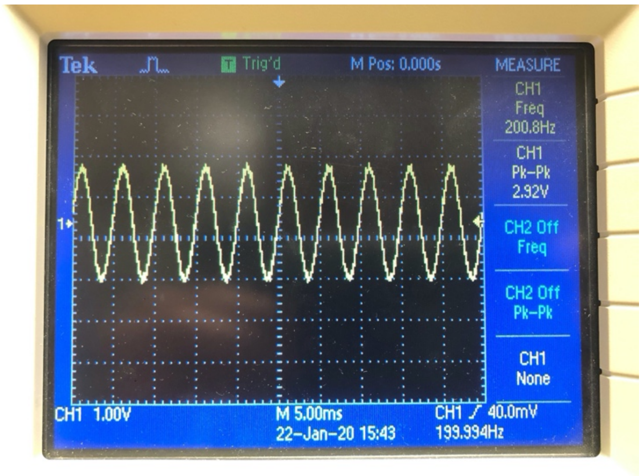
In the case of 10 Hz, the frequency lower than lower bound indicates that the signal generated will be as expected. In fact, the sine wave was in such a bad shape that no pattern could be detected. On the other hand, in the case of 8000Hz, it can be observed that the signal generated was corrupted completely.

However, in the case of 200Hz and 2000Hz, clear sine wave could be observed. It should be noted that the smoothness of sine wave will decrease as the signal frequency increases, as a result of increasing step size.

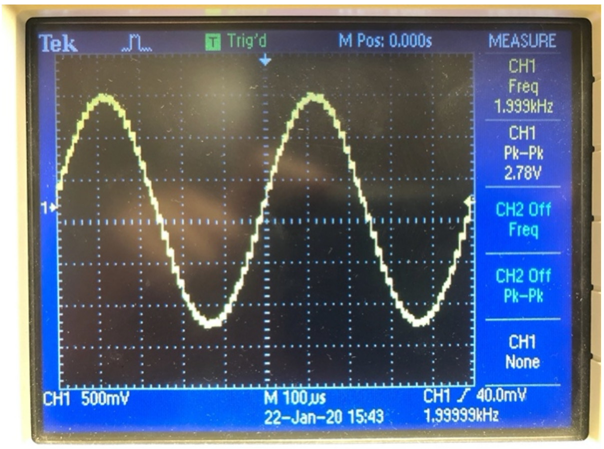
10 Hz



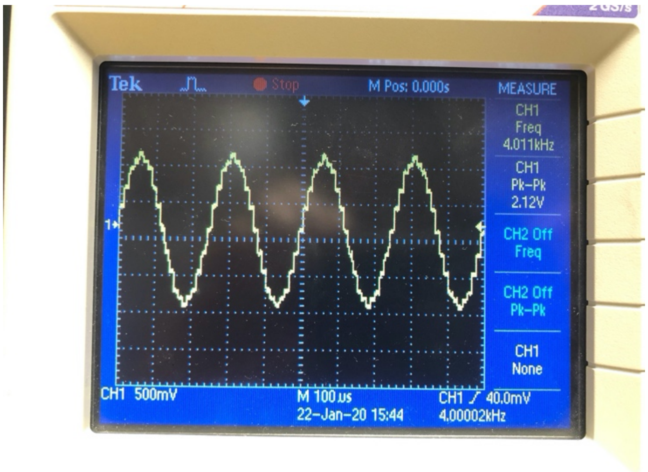
200Hz



2000 Hz



4000 Hz (Nyquist Sampling frequency)



The observations above are indeed what we expected.

5.Appendix

```
/*************************************************************************************
        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 outputing 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"
#include <stdio.h>

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

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


// PI defined here for use in your code
#define PI 3.141592653589793
#define SINE_TABLE_SIZE 256




/******************************* 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;
// Look up table for sin with size [SINE_TABLE_SIZE]
float table[SINE_TABLE_SIZE];



// Array of data used by sinegen to generate sine. These are the initial values.
float y[3] = {0,0,0};
float x[1] = {1}; // impulse to start filter

float a0 = 1.4142; // coefficients for difference equation
float b0 = 0.707;

// 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 = 2000.0;
float index=0;

/******************************* Function prototypes ********************************/
void init_hardware(void);
float sinegen(void);
void sine_init();
/********************************** Main routine ************************************/
void main()
{

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

 /* 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)
{
///*  This code produces a fixed sine of 1KHZ (if the sampling frequency is 8KHZ)
//    using a digital filter.
//  You will need to re-write this function to produce a sine of variable frequency
//  using a look up table instead of a filter.*/
//
// // temporary variable used to output values from function
 float wave;
//
// // represents the filter coefficients (square root of 2 and 1/square root of 2)
// float a0 = 1.4142;
// float b0 = 0.7071;
//
// y[0] = a0 * y[1] - y[2] + b0 * x[0]; // Difference equation
//
// y[2] = y[1]; // move values through buffer
// y[1] = y[0];
//
// x[0] = 0; // reset input to zero (to create the impulse)
//
// wave = y[0];
   wave = table[(int)index];
   // reads value from LUT
   index+=(float)(SINE_TABLE_SIZE/(sampling_freq/sine_freq));
   // increment index with respect to the target sine frequency
   if (index>SINE_TABLE_SIZE){
      index -= SINE_TABLE_SIZE;
   }
   // handle wrap around case
   // Alternatively, use code below to increase speed
   //      index &= SINE_TABLE_SIZE-1
   // not need to use if statement
   return(wave);

}

void sine_init(){
   int i;
   float value;
   for (i=0;i<SINE_TABLE_SIZE;i++){
      value=sin(i*2*PI/(SINE_TABLE_SIZE));


      table[i]=value;
   }
}
```