RTDSP Lab2 Write-up – Learning C and Sine Wave Generation

Michael Li ml1811, Weng Lio wnl111(EIE3)

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?*

   Here is the trace table for 12 cycles, where x(n) represents the input and y(n) represents the output.

n	x(n)	y(n)	y(n-1)	y(n-2)
0	1	0.7071	0.0000	0.0000
1	0	1.0000	0.7071	0.0000
2	0	0.7071	1.0000	0.7071
3	0	0.0000	0.7071	1.0000
4	0	-0.7071	0.0000	0.7071
5	0	-1.0000	-0.7071	0.0000
6	0	-0.7071	-1.0000	-0.7071
7	0	0.0000	-0.7071	-1.0000
8	0	0.7071	0.0000	-0.7071
9	0	1.0000	0.7071	0.0000
10	0	0.7071	1.0000	0.7071
11	0	0.0000	0.7071	1.0000

   From the table it is clear that it takes 8 samples to generate a complete sinewave.

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?*

   `Sampling_freq` = 8kHz and it takes 8 samples to generate a complete sinewave, therefore the output sinewave is fixed at 8kHz/8 = 1kHz.

```
//  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))))
{};
```

   These lines in the while loop of main function wait until the left and right output channel is ready. The function `DSK6713_AIC23_write` returns `true` after a wait period that's the reciprocal of the sampling frequency. If we can only write to the output 8000 times a second, and it takes 8 samples to complete a sine wave, this would throttle the output sinewave to 1kHz.

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?*
   As shown in the code snippet above, `sample` returned from the sinegen function is multiplied by the gain and type casted to a 32-bit integer. Therefore 32 bits are used to encode each sample that is sent to the audio port.

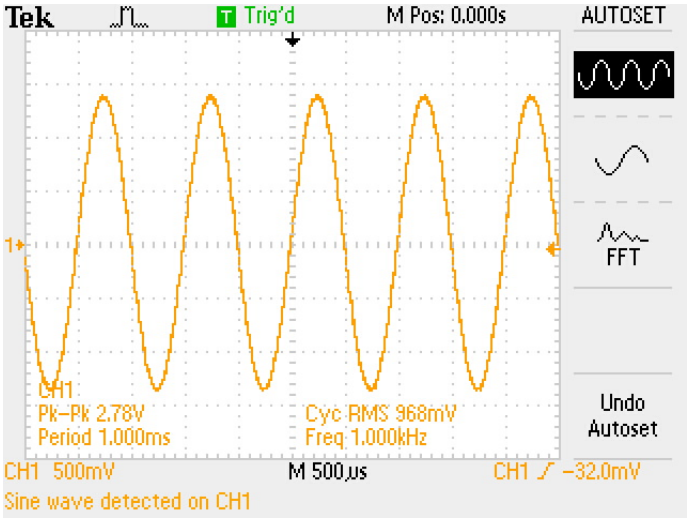
Code Operation

At the start of the main function we initialize the board, the audio ports and the values in the sine lookup table. A value of from the lookup table is then returned as `sample` from the `sinegen` function, which is then sent to the left and right channel of the DSK board at the sampling frequency.

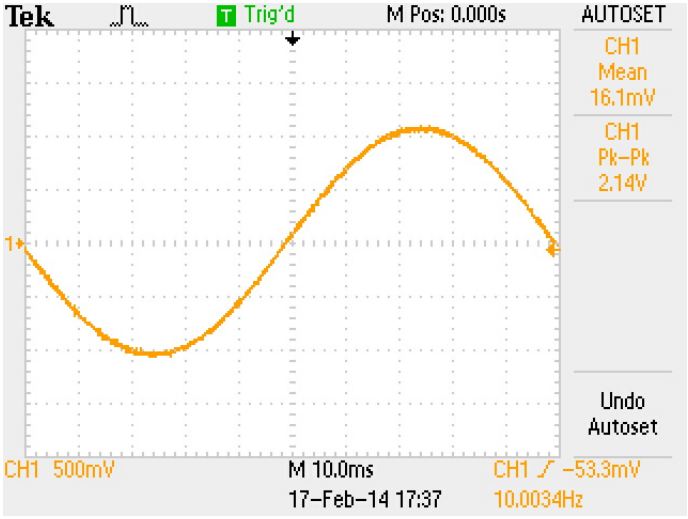
```
float sinegen(void)
{
    // x is global float variable
    float jump;             //gap to next sample in lookup table

    jump = (SINE_TABLE_SIZE*sine_freq/sampling_freq);
    x += jump;              //increment x by jump

    while(x>255){            //wrap round lookup table
        x-=SINE_TABLE_SIZE;
    }

    return(table[(int)round(x)]);
}
```

Inside the `sinegen` function, a float variable `jump` is used to calculate the gap to the next entry of the lookup table required according to `sine_freq` relative to `sampling_freq`. A global variable `x` is used as the index and its value is incremented by `jump` every time `sinegen` is executed.
The `while` loop wraps `x` around the size of the sine lookup table (in this case 256) to avoid array index overflow. Finally we round off `x` instead of just taking the floor to provide better approximation of the corresponding index, the function returns the entry in the sine lookup table with index `x` cast as an `int`.

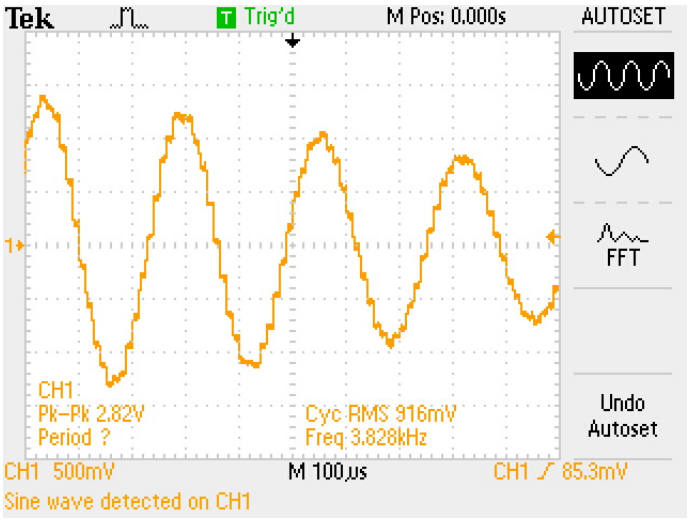
Scope Traces



Scope trace showing 1kHz sine output



Scope trace showing lower limit of operation at 10Hz



Scope trace showing upper limit as we approach the Nyquist freq. of 4kHz

Limitations of code operation

Using this table lookup method means the resolution of the outputted sine wave is limited by the size of the table. To increase resolution without using a larger lookup table, we can store only a quarter of the sine wave and make use its symmetric nature, i.e. traverse the table in both directions and multiply the returned values by 1 or -1 according to the quadrant desired.

The theoretical maximum sine output frequency should be slightly under half the sampling frequency we set. This constraint arises due to the symmetric nature of sine waves; if we output at half the sampling frequency, we would be outputting 2 entries from the lookup table per period, one at the start, one halfway, both of which are zero. Reconstructing anything sensible out of this would be difficult. As we increase the desired output frequency closer to this limit, the sine wave became noisier due to any rounding error being much more pronounced with so few samples, as well as the actual timing of the output being not precisely as set by the sampling frequency.

Since we are simply adding the jump to find the next index in the array to read from, we want to avoid truncating `jump` to zero due to integer casting of the index as this would not minimise the lower bound of frequencies we can output. Hence we retain x as a float throughout the function and it is only read as an `int` from the table when the function returns. When using a sampling frequency of 8kHz we were able to produce a sine wave of 10Hz up to just below the Nyquist frequency.

When we output at a frequency slower than 31.25Hz with sampling frequency at 8kHz, `jump` is less than 1. Therefore, we get a side effect where the sampling time is effectively reduced because of limited entries in the lookup table. To illustrate, for some samples, the calculated index will not have reached the next integer and we will end up outputting the same sample value. A much better solution would be to use the approach where only half or a quarter of the sine wave is stored to increase the resolution without increasing data usage as mentioned before.

Full Code Listing

```
/*************************************************************************************
                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"

// 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

//sine generation look up table size
#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;
```

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


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 = 1000.0;

float table[SINE_TABLE_SIZE];
float x = 0;
int count = 0;
/******************************* Function prototypes *********************************/
void init_hardware(void);
float sinegen(void);
void sine_init(void);
/********************************** 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)
{
    // x is global float variable
    float jump;                                 //gap to next sample in lookup table

    jump = (SINE_TABLE_SIZE*sine_freq/sampling_freq);
    x += jump;                                  //increment x by jump

    while(x>255){                               //wrap round lookup table
        x-=SINE_TABLE_SIZE;
    }

    return(table[(int)round(x)]);
}

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

Declaration: We confirm that this submission is our own work.
In it, I give references and citations whenever I refer to or use the published, or unpublished, work of others. I am aware that this course is bound by penalties as set out in the College examination offenses policy.

Signed: Michael Li, Weng Lio