Lab 2 Real time digital signal processing

Lizhang Lin (00840705)	Yihan Qi (00813873)

1. Objective

In this laboratory session, we aimed to learn how to build a project and to familiarize with how to use functions in C. This report mainly discussed the outcomes of two activities which generated a sinewave on DSP board using table lookup and digital filter respectively.

2. Part 1: Generating a Sinewave Using Digital Filter

At the beginning, a C code called *sine.c* was provided, which is able to generate a sine wave with fixed frequency of 1 KHz under a sampling frequency of 8 kHz with IIR filter algorithm. Theoretically, IIR filter is capable of producing an infinite response because there is a feedback loop in the filter. Namely, a single impulse input will result in an output of infinite number of non-zero values. In addition, IIR filters can achieve a given filtering characteristic using less memory and calculations.

In *sine.c*, we receive sinewave sample in an endless *while* loop in the *main* function. Then we wrote the sample fetched from **sinegen()** function into the hardware. The related code is shown below:

```
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);

 }
```

sinegen() function returns a floating output called *wave*. The way we generated this variable is related to the theory we mention above, IIR filter theory. Firstly, we set two arrays at the very beginning of *sine.c* and assigned their initial values, as shown in the below code.

```c
/* 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 x[1] = {1}; // impulse to start filter

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

Inside *sinegen*, we produced sinewave samples as indicated below:

```c
/********************************** 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;

    // represets the filter coeficients (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];

    return(wave);

}
```

2.1. Question 1

The table shown below illustrates the value fed into the array on each complete cycle of sinewave which contains 8 samples.

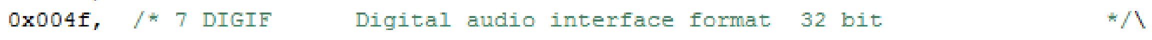
Column1	Initial setting	loop1	loop2	loop3	loop4	loop5	loop6	loop7	loop8
y[0]	0	0.7071	1	0.7071	0	-0.7071	-1	-0.7071	0
y[1]	0	0.7071	1	0.7071	0	-0.7071	-1	-0.7071	0
y[2]	0	0	0.7071	1	0.7071	0	-0.7071	-1	-0.7071
x[0]	1	0	0	0	0	0	0	0	0
Output	N/A	0.7071	1	0.7071	0	-0.7071	-1	-0.7071	0

2.2. Question 2

Therefore, sampling frequency divided by 8 samples gives 1 kHz, which is also known as the output sinewave frequency.
To be noticed, due to the hardware (mainly because of codec) limitation, we could only set one of the default sampling frequencies (*sampling_freq*), which are 8000, 16000, 24000, 32000, 44100, 48000 and 96000 Hz. Therefore, the minimum fixed frequency throttled by the hardware is $\frac{8kHz}{8}$ since there are fixed 8 samples in each cycle of sinewave, which equals to 1 kHz.

2.3. Question 3

Moreover, the number of bits used in encoding each sample which is sent to the audio port is limited by the code below.

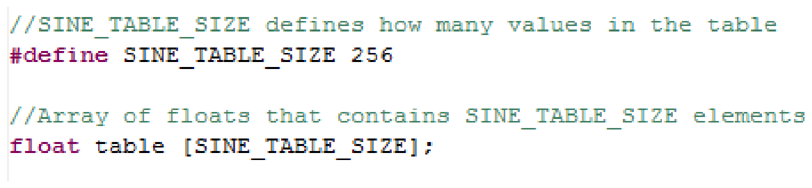


```
0x004f,  /* 7 DIGIF      Digital audio interface format  32 bit                */\
```

Thus, each sample sent to the audio port is encoded in 32 bits.

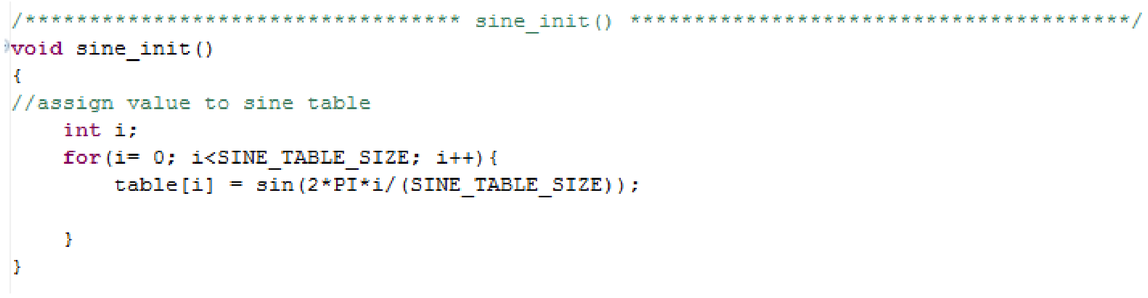
3. Part 2: Generating a Sinewave Using Lookup Table

In the second part of the experiment, we generated a sinewave by referring to lookup table. Firstly, we defined a global constant *SINE_TABLE_SIZE* which defines how many values will be stored in the lookup table. In this experiment, it is set to 256. Then we defined a global variable called *table* which is an array of floating variables containing *SINE_TABLE_SIZE* many elements.



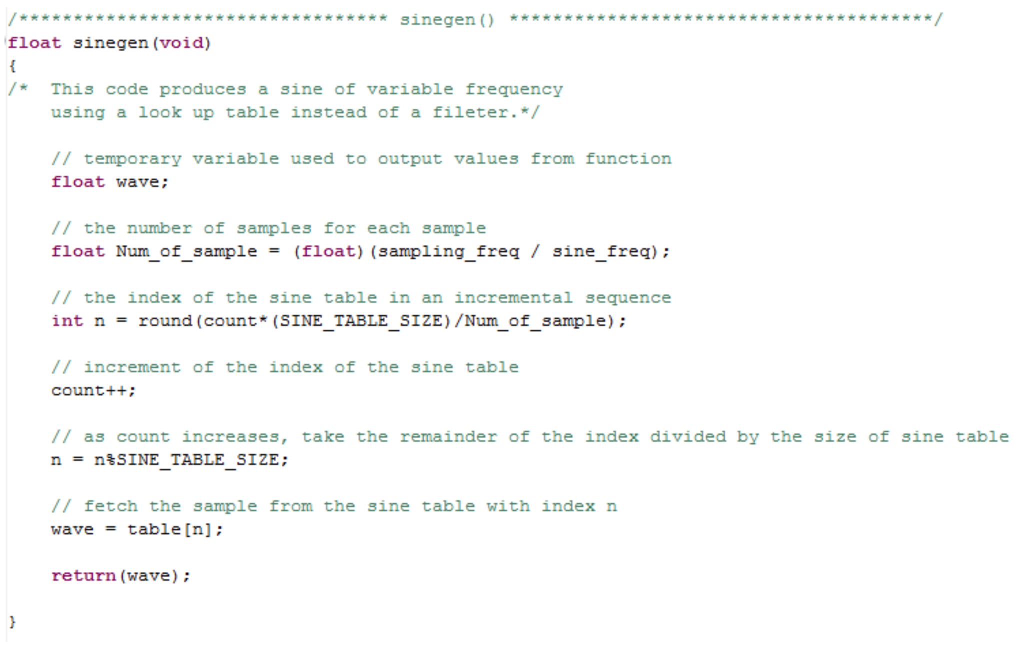
```
//SINE_TABLE_SIZE defines how many values in the table
#define SINE_TABLE_SIZE 256

//Array of floats that contains SINE_TABLE_SIZE elements
float table [SINE_TABLE_SIZE];
```

In order to fill our table with appropriate sine values, we created a function called *sine_init()* to do this. It will automatically initialize sinewave data in look-up table. The algorithm of this function is very straightforward. It’s realized using a *for loop,* with *SINE_TABLE_SIZE* many loops.



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

    }
}
```

We were then ready to write a function called *sinegen* which is able to fetch the data from the look-up table and to change the sampling frequency. The code of sinegen function is shown below.



```
/********************************** sinegen() ***************************************/
float sinegen(void)
{
/*  This code produces a sine of variable frequency
    using a look up table instead of a fileter.*/

    // temporary variable used to output values from function
    float wave;

    // the number of samples for each sample
    float Num_of_sample = (float)(sampling_freq / sine_freq);

    // the index of the sine table in an incremental sequence
    int n = round(count*(SINE_TABLE_SIZE)/Num_of_sample);

    // increment of the index of the sine table
    count++;

    // as count increases, take the remainder of the index divided by the size of sine table
    n = n%SINE_TABLE_SIZE;

    // fetch the sample from the sine table with index n
    wave = table[n];

    return(wave);

}
```

The result of $\frac{sampling\_freq}{sine\_freq}$ is the number of samples (*Num_of_sample*) we wanted to read from the lookup table. Here, we introduced a **global** variable called *count*. It is used for generating the correct samples by gapping each sample by $\frac{SINE\_TABLE\_SIZE}{Num\_of\_sample}$ many index. This increment multiplied with *count* produces *n*, which is the index for fetching the data from the lookup table.

In some cases, there is a problem that the value of *n* could be greater than the size of the table. We solved this by only taking the reminder of $\frac{n}{SINE\_TABLE\_SIZE}$. Thus, once *n* becomes larger than the size of the table, its value is reassigned as $(n - SINE\_TABLE\_SIZE)$ which indicates the start of next sinewave cycle.

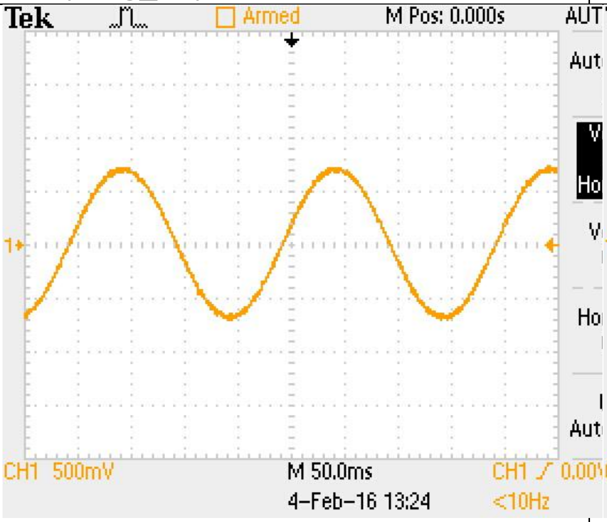
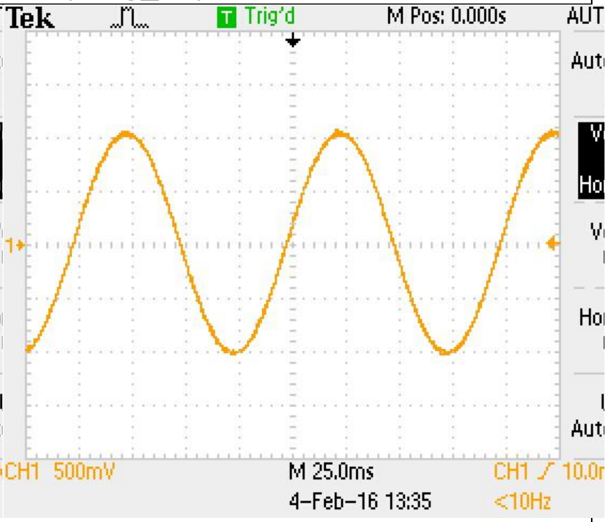
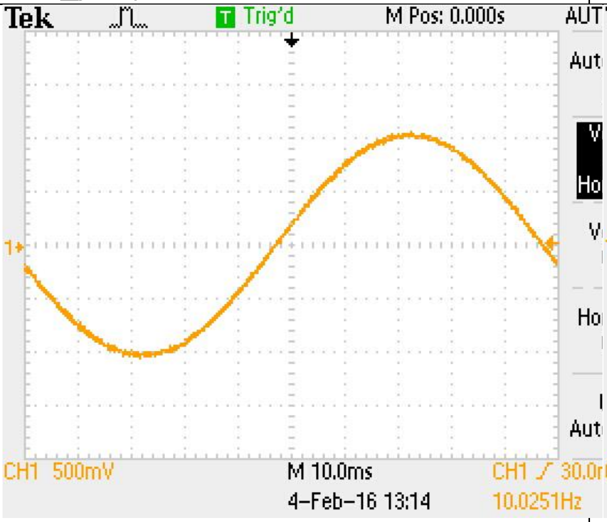
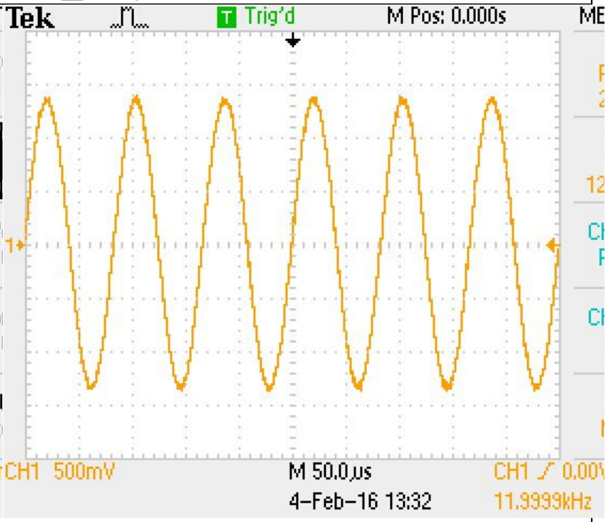
4. Improving the Output Resolution

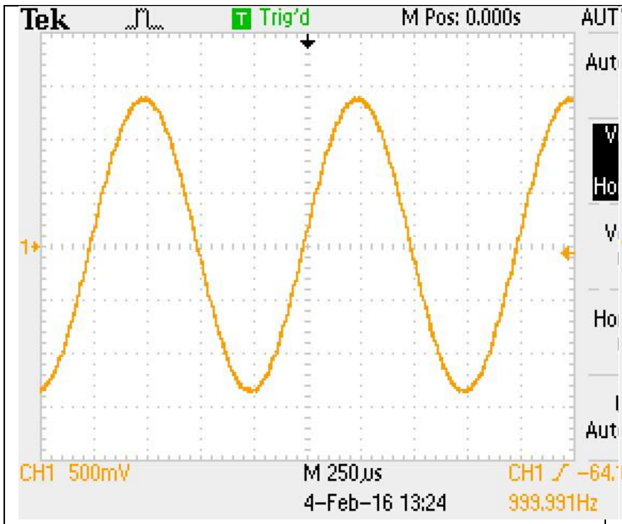
In order to improve the output resolution, we deliberately assigned any increment or the number of samples as **floating** types in calculation and rounded the final result as **integer** *n*. Therefore, the error produced in the numerical manipulation is reduced.
Moreover, we had another idea of improving the output resolution though it's not realized in our code. In ordinary sinewave generation, if we want to read 8 data from the lookup table, we will equally fetch them throughout the whole table. In fact, we could utilise the symmetrical properties of sinewave and only

produce part of the wave and then copy it for producing the rest. In another words, we could fetch 8 samples within a quarter cycle rather than a whole cycle so that the quarter wave generated will have more resolution. For the rest of the cycle (3/4 cycle), we could get them via transforming the first quarter graphically, such as folding & overturning.

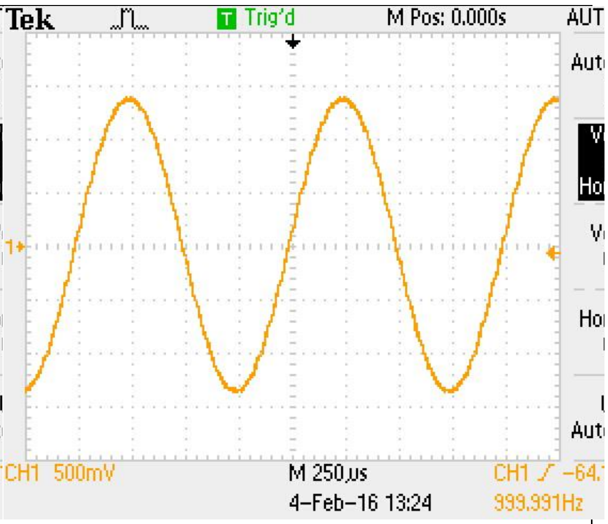
5. Scope Trace

Here, we presented a range of scope traces under different sampling frequencies.

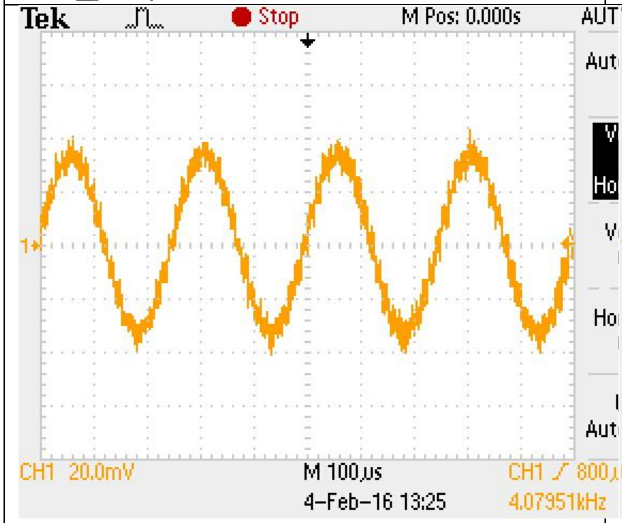
sampling_frq = 8kHz	sampling_frq = 96kHz
	
sine_freq=5 Hz	*sine_freq= 5 Hz*
	
sine_freq=10 Hz	*sine_freq=12 kHz*



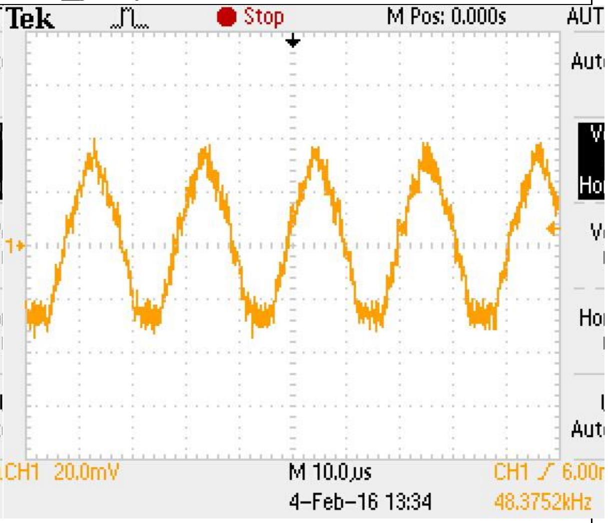
sine_freq=1 kHz



sine_freq=1 kHz



sine_freq=4 kHz (Nyquist frequency)

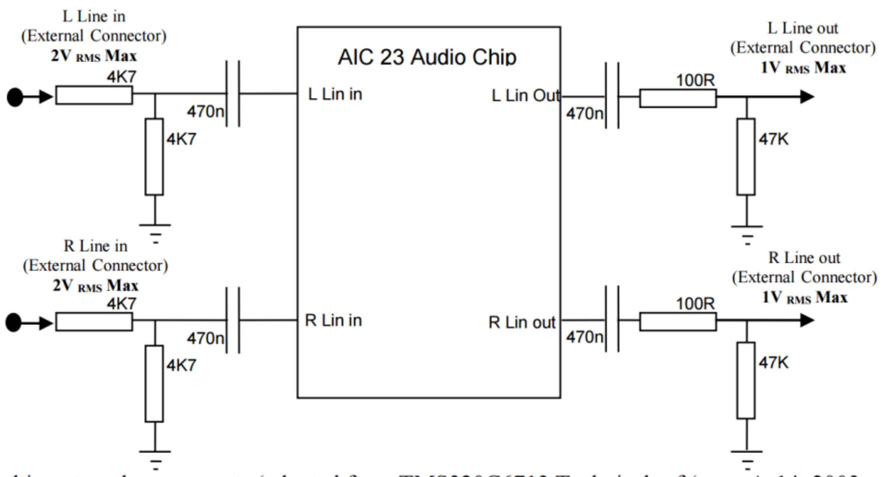


sine_freq=48 kHz (Nyquist frequency)

During the observation of scope traces, we found the lower and upper boundaries of the sinewave frequency. For the upper bound, it is easy to understand that *sine_freq* will never be larger than Nyquist frequency, which is the half of the *sampling_freq*. Thus, if *sampling_frq* is 8 kHz, the maximum *sine_freq* we could get must be 4 kHz.
However, the lower bound is limited by the hardware configuration. The diagram shown below is picture of the Analogue IO configuration of the 6713 DSK, which is snipped from the lab manual. We could see this at the output end, there are two simple high pass filters which are located on R-out and L-out separately. Its cut-off frequency is $f = \frac{1}{2\pi RC} = \frac{1}{2\pi\times 47k\times 470n} = 7.205\ Hz$.
Therefore, we concluded that 7.205Hz affects the lower bound of *sine_freq* so that any *sine_freq* below this value may result in a deteriorated sinewave on

the oscilloscope.

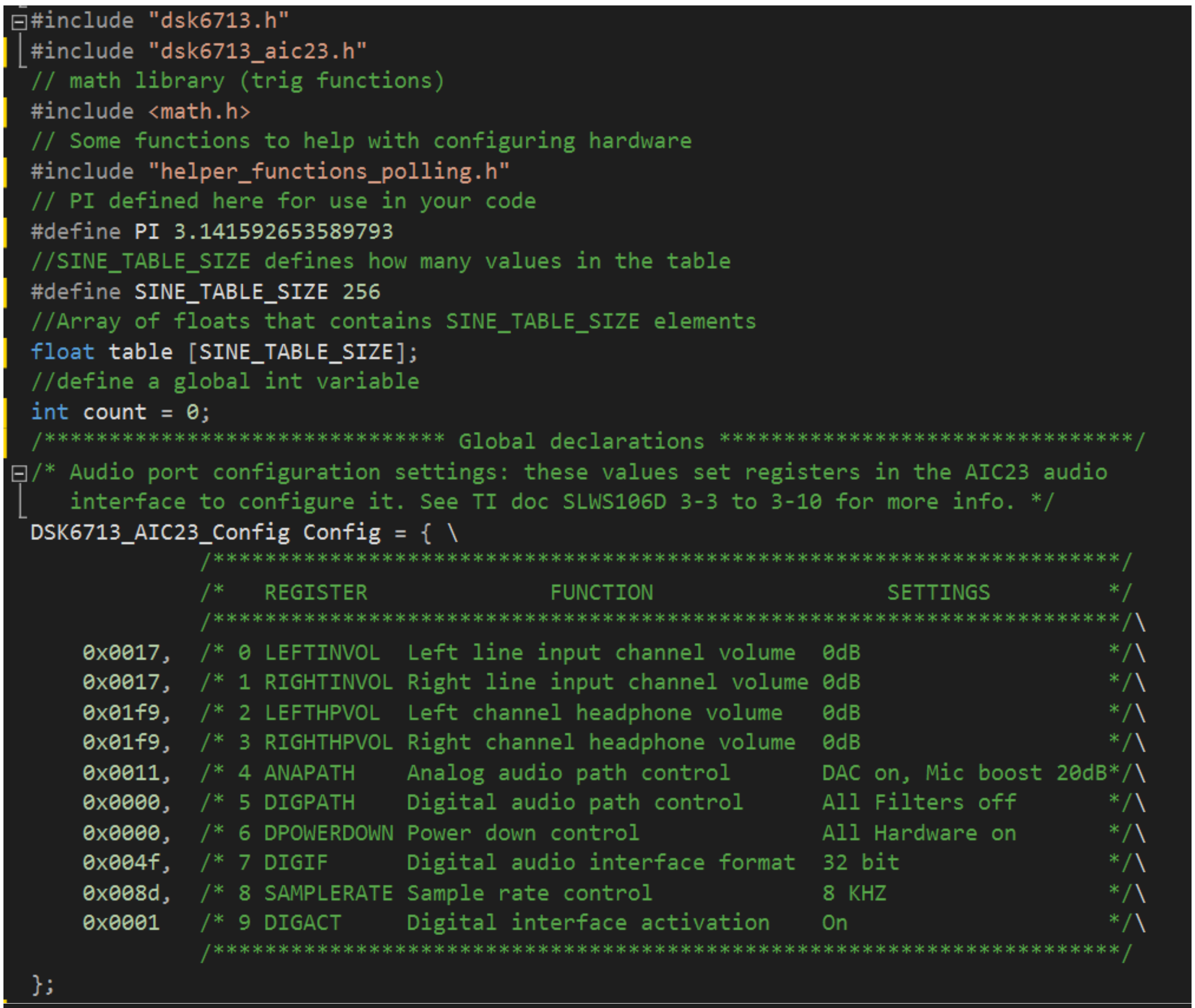
Appendix 2 –Analogue IO configuration on the 6713 DSK



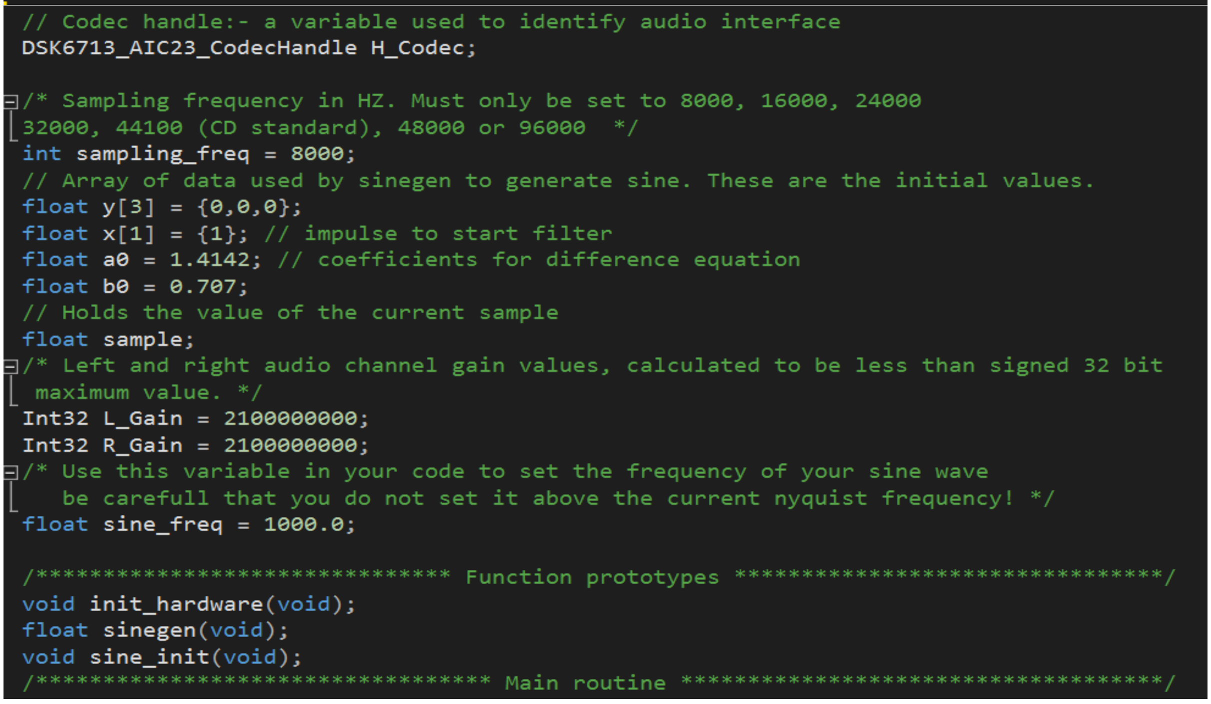
AIC23 Audio chip external components (adapted from TMS320C6713 Technical ref (page A-14, 2003 revision A)

6. Appendix

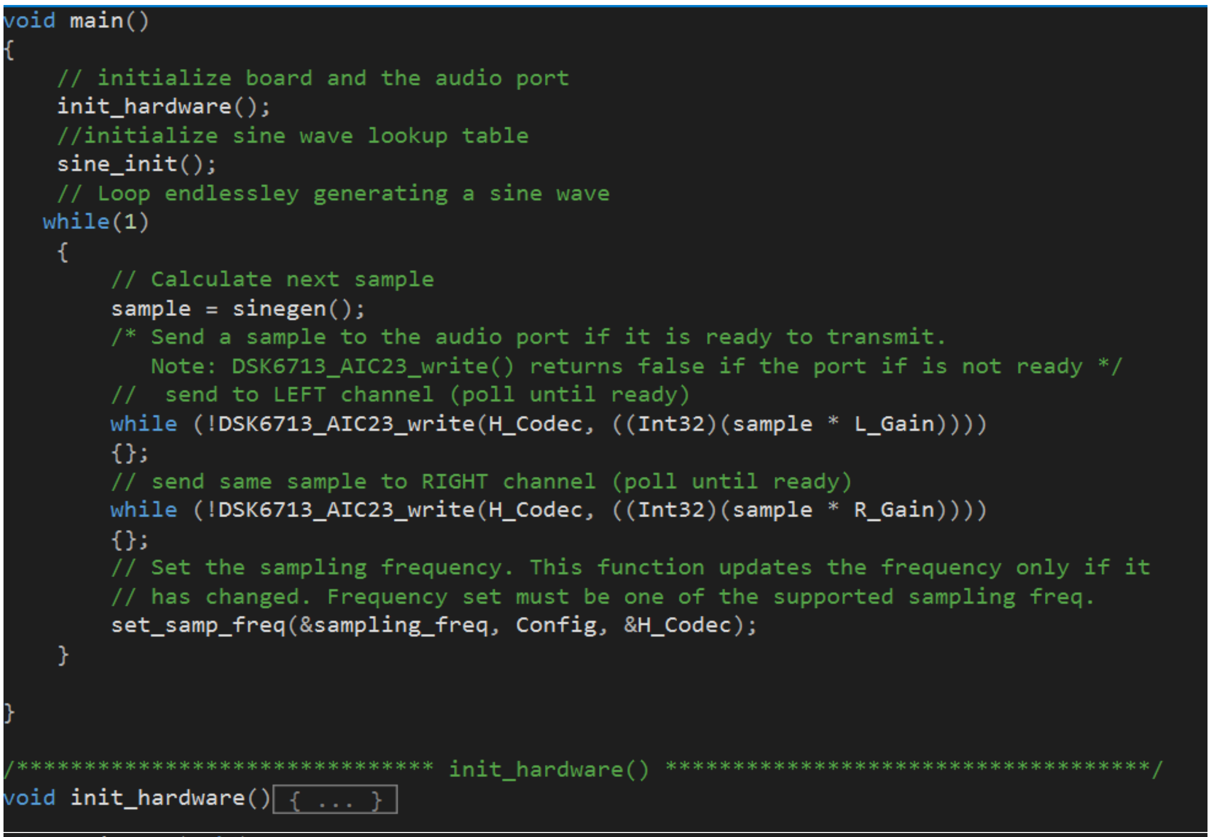
Part 1 & 2 Code:



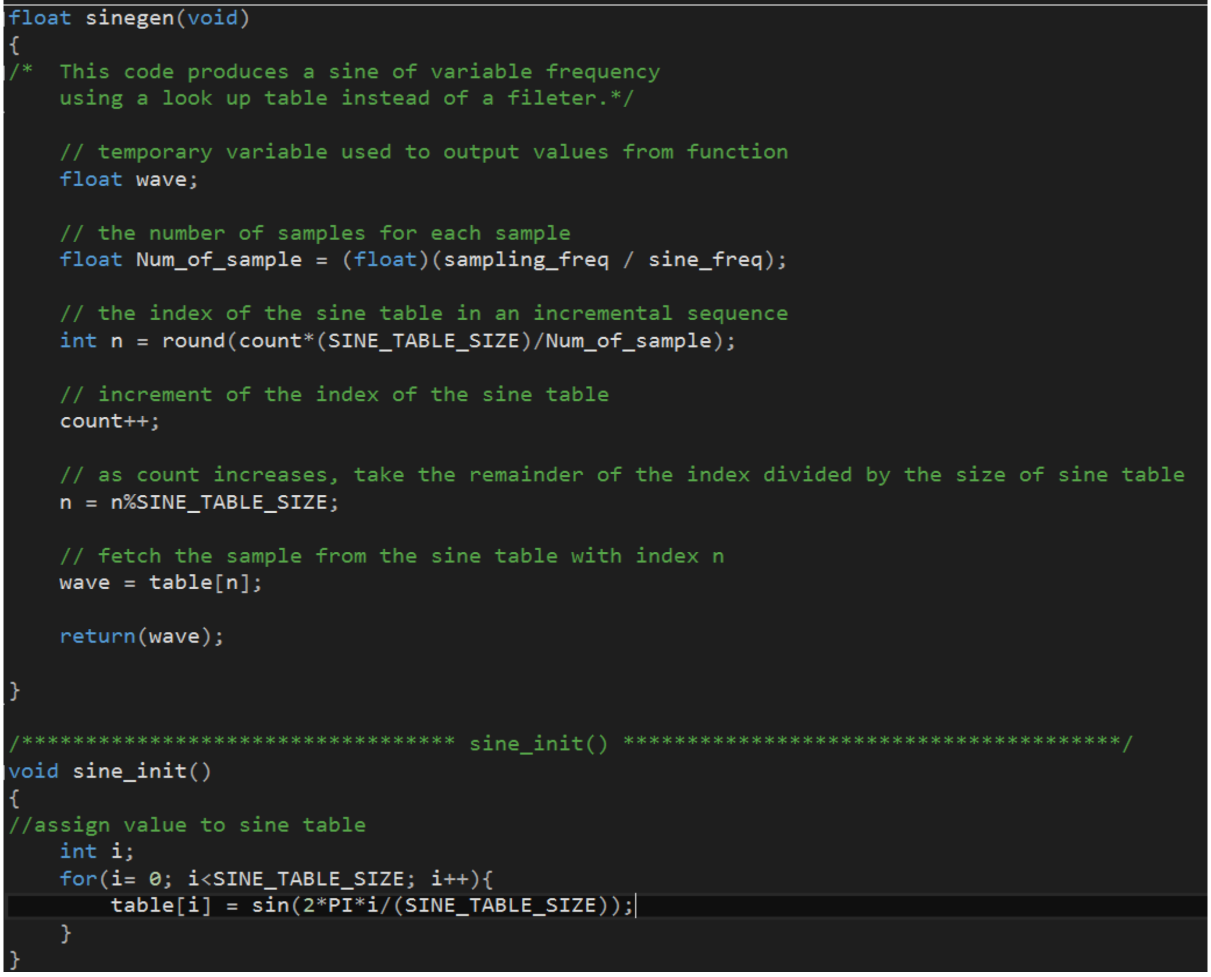
```c
#include "dsk6713.h"
#include "dsk6713_aic23.h"
// math library (trig functions)
#include <math.h>
// Some functions to help with configuring hardware
#include "helper_functions_polling.h"
// PI defined here for use in your code
#define PI 3.141592653589793
//SINE_TABLE_SIZE defines how many values in the table
#define SINE_TABLE_SIZE 256
//Array of floats that contains SINE_TABLE_SIZE elements
float table [SINE_TABLE_SIZE];
//define a global int variable
int count = 0;
/****************************** 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                     */\
             /**********************************************************************/
};
```



```c
// 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 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 = 1000.0;

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



```c
void main()
{
    // initialize board and the audio port
    init_hardware();
    //initialize sine wave lookup 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() { ... }
```



```c
float sinegen(void)
{
/*  This code produces a sine of variable frequency
    using a look up table instead of a fileter.*/

    // temporary variable used to output values from function
    float wave;

    // the number of samples for each sample
    float Num_of_sample = (float)(sampling_freq / sine_freq);

    // the index of the sine table in an incremental sequence
    int n = round(count*(SINE_TABLE_SIZE)/Num_of_sample);

    // increment of the index of the sine table
    count++;

    // as count increases, take the remainder of the index divided by the size of sine table
    n = n%SINE_TABLE_SIZE;

    // fetch the sample from the sine table with index n
    wave = table[n];

    return(wave);

}

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