EE3-19 Real Time Digital Signal Processing

Coursework Report 1: Lab 2

Jeremy Chan*
Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
jc4913@ic.ac.uk | 00818433

Chak Yeung Dominic Kwok*
Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
cyk113@ic.ac.uk | 00827832

*These authors contributed equally to this work

Abstract— **This is the first report for E3-19 Real Time Digital Signal Processing's (2015-2016). The aim is to understand sine wave generation on hardware (lab 2).**

I. Introduction

A skeleton code for sine wave generation was provided – modifications were made to allow for the generation of a variable frequency sine wave, increase resolution of the lookup table, and to implement a difference equation method instead of the lookup table method. Oscilloscope graphs are shown for multiple frequencies to prove the operation of the code, along with code explanations in both the main text and the appendix.

II. Questions

A. *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?*

The trace table, shown in Figure I, was generated by copying the first 20 results to an array. Hence, it takes 8 samples to generate a whole cycle. This can also be found by $\frac{f_{sample}}{f} = \frac{8000}{1000} = 8$.

Local (1) | Watch (1)

Name	Value	Address
sample	-0.9903384	0x00009980
table	0x0000842C	0x0000842C
[0]	0.7071	0x0000842C
[1]	0.9999807	0x00008430
[2]	0.7070728	0x00008434
[3]	-3.8445e-05	0x00008438
[4]	-0.7071272	0x0000843C
[5]	-0.9999807	0x00008440
[6]	-0.7070456	0x00008444
[7]	7.688999e-05	0x00008448
[8]	0.7071543	0x0000844C
[9]	0.9999807	0x00008450
[10]	0.7070183	0x00008454
[11]	-0.0001153946	0x00008458
[12]	-0.7071815	0x0000845C
[13]	-0.9999807	0x00008460
[14]	-0.7069911	0x00008464
[15]	0.0001538396	0x00008468
[16]	0.7072087	0x0000846C
[17]	0.9999806	0x00008470
[18]	0.7069638	0x00008474
[19]	-0.0001924038	0x00008478
<new>		

Fig. 1. Trace Table of the first 20 outputs

B. *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 frequency2 by changing sampling_freq).*

By looking at $Z\{\sin(2\pi kn)\} = \frac{z\sin(2\pi k)}{z^2-2z\cos(2\pi k)+1}, where\ k = \frac{f}{f_{sample}}$, and using Appendix 1 of the lab booklet, the sinegen(void) function implements an IIR filter with a transfer function: $\frac{Y(Z)}{X(Z)} = \frac{1}{\sqrt{2}-2z^{-1}+\sqrt{2}z^{-2}}\frac{z^2}{z^2} = \frac{z^2}{\sqrt{2}z^2-2z+\sqrt{2}}$. Hence, if $k = \frac{1}{8}$, $Z\left\{\sin\left(\frac{\pi n}{4}\right)\right\} = \frac{z^2\frac{1}{\sqrt{2}}}{z^2-\frac{2}{\sqrt{2}}z+1} = \frac{z^2}{\sqrt{2}z^2-2z+\sqrt{2}}$. Therefore, since sampling_freq is defined as 8000, the output sine_freq is 1000.

By looking at the code to pass the output samples to the DAC, the program continuously polls the DAC to see if its buffer is empty.

```
while (!DSK6713_AIC23_write(H_Codec,
((Int32)(sample * L_Gain)))
        {};
```

Since the DAC hardware is configured to have a sample rate of `0x008d`, hence 8Khz, the DAC buffer will empty once every 125µs.

C. *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?*

```
void init_hardware(),MCBSP_FSETS(XCR1,
XWDLEN1, 32BIT);
```

shows that the resolution is 32 bits. This also matches the `DSK6713_AIC23_Config` initial setting of `0x004f` -> 32 bit.

III. Operation of Code

A. *Initialising the Hardware*

`void init_hardware()` configures the hardware using defined global variables. The struct of the config for configuring the handle can be found in the included header files. This function is executed as the first line of main before any testbench code is executed.

B. *Main Testbench*

The main function first initializes the hardware, enters a while (1) infinite loop that calculates a value for current sine wave coefficient before entering a nested while loop, polling the DAC buffer for write availability. If the DAC is not yet ready (corresponding to the configured DAC sampling frequency), the program will do nothing and try again. The rate of the program trying to ask for write availability depends on the clock speed of the processor running the program.

C. *Look-up Table Function*

A table that consists of the coefficients of a sine wave can be generated by a for loop. Then, to generate a variable sine wave, the program can iterate through this look-up table. To generate a sine wave of a higher frequency, the program can skip values in the look-up table. Care has to be taken regarding the wrapping around of the array index when it is larger than the array size.

The step size can be defined as

```
freq*SINE_TABLE_SIZE/sampling_freq;
```

a) *Modification 1: linear interpolation*

The result of the sine wave to be output can be improved slightly by taking the gradient of the current sample [1].

```
float result =
(iterator<=SINE_TABLE_SIZE-
2)?((table[(int)iterator]) +
((table[(int)(iterator+1)] -
table[(int)iterator])/2)):(table[(int)it
erator]);
```

b) *Modification 2: Adding resolution to look-up table*

The creation of the look-up table involves calculating coefficients for a whole sine wave. Since a sine wave is symmetrical, the look-up table can be improved by only generating a quarter of a sine wave. Extra logic (TABLE I) will then have to be implemented to take into account the current phase, direction of iteration, and whether the current value is positive or negative [2].

TABLE I.

Quadrant (degrees)	Array Index	Negation
0 – 90	Index	0
90 – 180	Table size – index	0
180 – 270	Index	1
270 - 360	Table size - index	1

c) *Modification 3: Implementing the difference equation instead of the lookup table method*

$$Z\{\sin(2\pi kn)\} = \frac{z\sin(2\pi k)}{z^2 - 2z\cos(2\pi k) + 1}$$

```
float a0 = 2*cos(2*PI*sine_freq/sampling_freq);
float b0 = sin(2*PI*sine_freq/sampling_freq);
y[0] = a0 * y[1] - y[2] + b0 * x[0];
```

By changing the hardcoded a0 and b0 values, a variable sinewave can be generated without the need for a lookup table.

IV. Discussion of Results

A. *Limitation of generated frequencies*

The limitation on the upper bound of frequencies that can be generated can be explained using the Nyquist rate – the absolute upper bound on the generated frequency should be $\frac{f_{sample}}{2} = 4KHz$.

Since our sine wave is digitally generated, the limit on the digital frequency, k, is from $-\frac{1}{2} to \frac{1}{2}$. Around the Nyquist rate ($\frac{f_{sample}}{2} = 4KHz$), the generated waveform becomes slightly aliased (Figure 2).

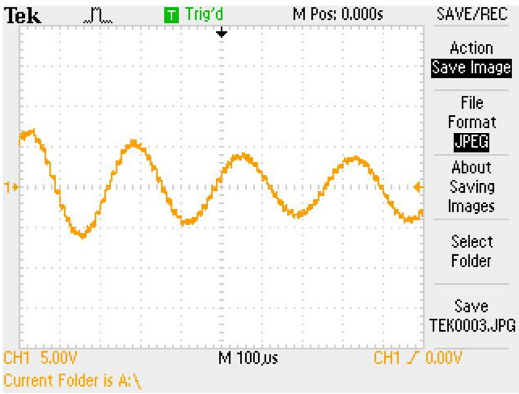


Fig. 2. 8000Hz sampling frequency /3900Hz sine frequency

However, when the generated frequency is over the Nyquist rate of 4KHz, a visible sine wave appears . This sine wave has a frequency of $f_s\left|1 - \frac{f}{f_s}\right|$. For Figure 3, this sine wave was set to be 4500Hz, but measures to be 3500Hz. The filter's frequency response is in Appendix.B.

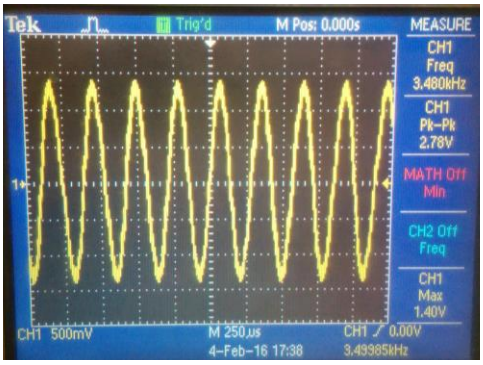


Fig. 3. Scope showing 4500Hz -> 3500Hz

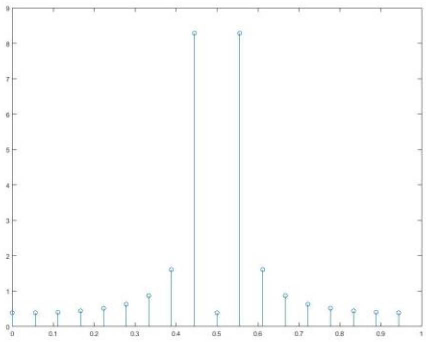


Fig. 4. 8000Hz sampling frequency /4500Hz sine frequency, pre filter

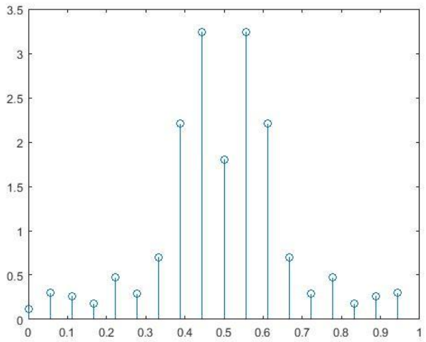


Fig. 5. 8000Hz sampling frequency /4500Hz sine frequency, post filter

Additionally, there are A.M.-like characteristics when zooming out (Figure 6). By looking at Figure 3-19 of the Texas Instruments AIC23 datasheet (Appendix.A), the low pass filter implementation inside the DAC can be seen. When the output frequency is near the Nyquist frequency, e.g. at 4050Hz, the gain is around 0.5. Therefore, the output of the DAC (Figure 7) is

$$\sin\big((2)(3950)\pi t\big) + 0.5\sin\big((2)(4050)\pi t\big)$$

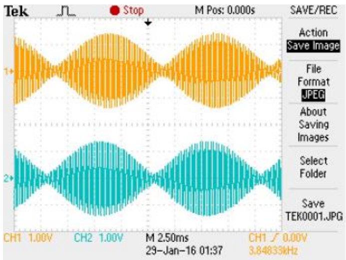


Fig. 6. AM Characteristics near Nyquist

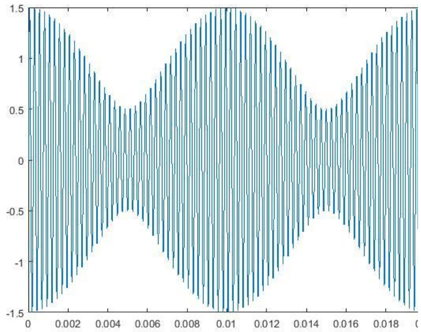


Fig. 7. Output of DAC (MATLAB)

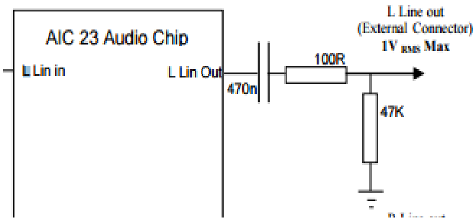


Fig. 8. Output High Pass Filter of DAC

There is no lower bound on the generated frequency with respect to the code as the step size can be as small as possible – however, by looking at the output stage of the DAC (Figure 8), attenuation can be seen at frequencies lower than 100Hz.

By using MATLAB to plot a Bode plot of the filter (Figure 9), the high pass characteristics can be seen. Hence, there is no lower bound, but only a reduced amplitude when at low frequencies.

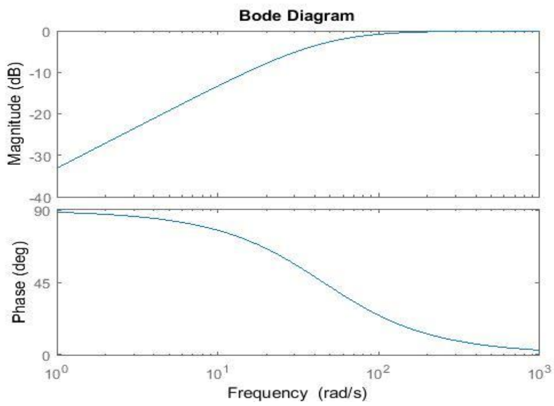


Fig. 9. Bode Plot of the DAC Output Stage

V. CONCLUSION

This task was completed successfully and improvements were made to improve the output of the generated sine wave.

Further improvements can include pre-computing the lookup table instead of generating it at runtime, and using two lookup tables to further improve the accuracy of the sine wave [2].

REFERENCES

[1] R. Bencina, "Ross Bencina," [Online]. Available: http://www.rossbencina.com/code/sinusoids. [Accessed 1 2 2016].

[2] P. Cheung, "EE3_DSD," 24 1 2008. [Online]. Available: http://www.ee.ic.ac.uk/pcheung/teaching/ee3_DSD/Topic%205%20-%20Function%20Evaluation.pdf. [Accessed 1 2 2016].

APPENDIX

A. *Texas Instruments AIC23 Datasheet*

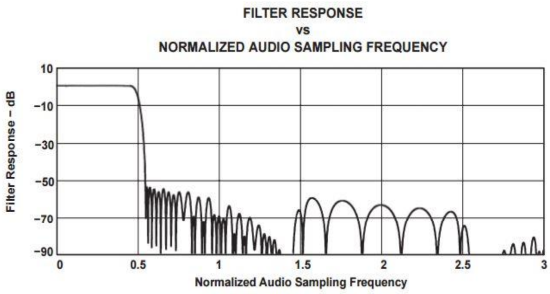
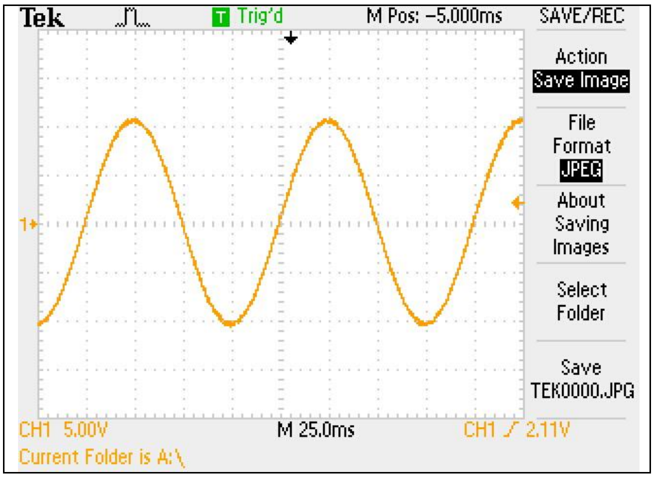
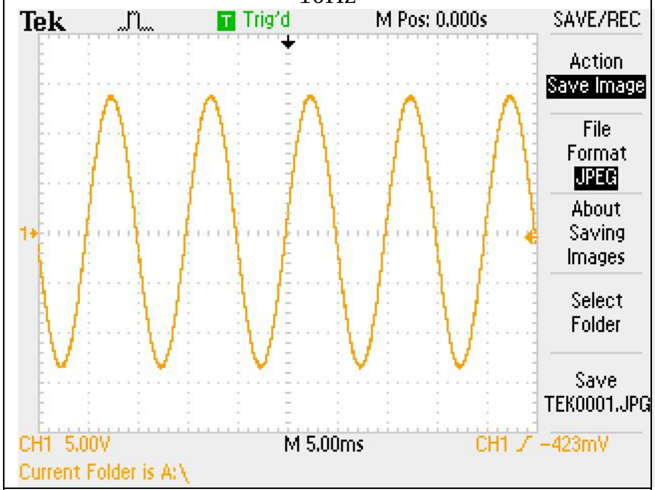


Figure 3–19. DAC Digital Filter Response 0: USB Mode

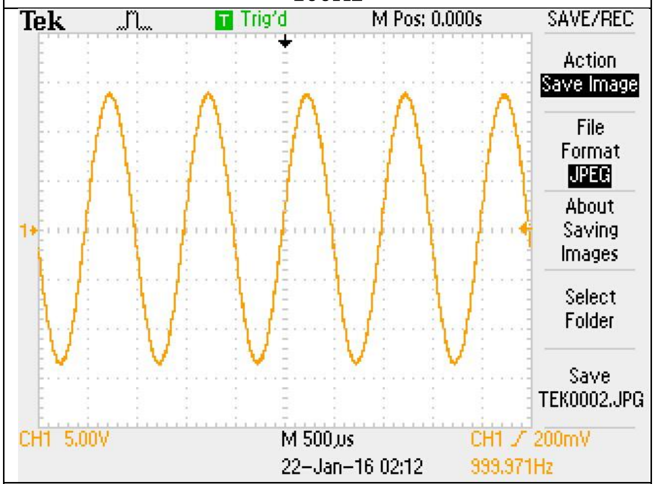
B. *Oscilloscope Traces (all at 8KHz sampling rate)*



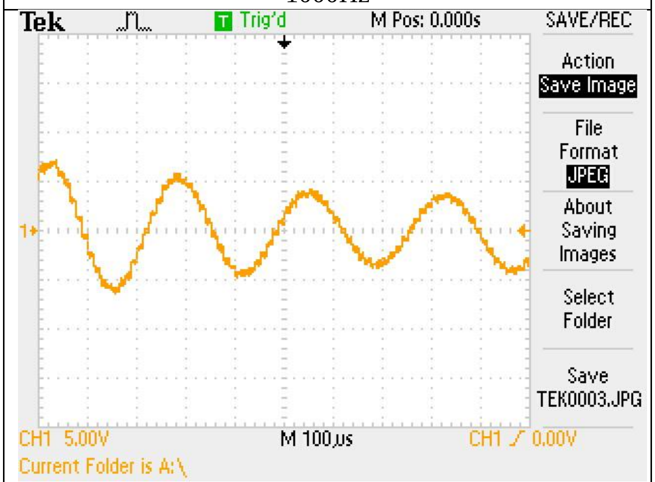
10Hz



100Hz

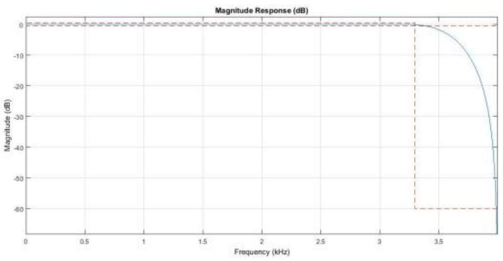


1000Hz



3900Hz

C. *MATLAB Filter Frequency Response*



D. *Code*

```
/*********************************************
*****************************************
                           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>
#include<stdlib.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;


// 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;
float sample2;
float table[SINE_TABLE_SIZE]={0};
float table3[SINE_TABLE_SIZE]={0};
void sine_init(void);
/* 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 = 3950.0;

float sinegen2(float* table, int freq);
float sinegen3(float* table, float freq);

/******************************** Function
prototypes ********************************/
void init_hardware(void);
float sinegen(void);
void sine3_init(void);
/*********************************** Main
routine ***************************************/
void main()
{
       // initialize board and the audio port
       init_hardware();
       sine_init();
       sine3_init();
    // Loop endlessley generating a sine wave
   while(1)
    {


              // Calculate next sample
              sample =
sinegen2(table,(int)sine_freq);
              sample2 =
sinegen3(table3,sine_freq);
              //sample2=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)(sample2 * 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;

       // represets the filter coeficients
(square root of 2 and 1/square root of 2)
       float a0 =
2*cos(2*PI*sine_freq/sampling_freq);
```

```
	float b0 =
sin(2*PI*sine_freq/sampling_freq);

	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);

}

void sine_init(void){
	int i=0;
	for (i=0;i< SINE_TABLE_SIZE;i++){
	table[i]=sin(2*PI*i/SINE_TABLE_SIZE);
	}
}
void sine3_init(void){
	int i=0;
	for (i=0;i< SINE_TABLE_SIZE;i++){
	table3[i]=sin(PI*0.5*i/SINE_TABLE_SIZE
);
	}
}

float sinegen2(float* table, float f){
	static float iterator = 0;
	//use linear iterpolation between next
value to get an average
	float result =
(iterator<=SINE_TABLE_SIZE-
2)?((table[(int)iterator]) +
((table[(int)(iterator+1)] -
table[(int)iterator])/2)):(table[(int)iterato
r]);
	iterator += f*( SINE_TABLE_SIZE
)/sampling_freq;
	if (iterator > SINE_TABLE_SIZE)
{iterator -= SINE_TABLE_SIZE;}
	return result;
}
float sinegen3(float* table, float freq){
	float step= 360.00*((float)
freq/(float)sampling_freq);

	static float phase=0;
	float result=0;
	if(phase<90){

	result=table[(int)((phase/90.0)*SINE_T
ABLE_SIZE)];
	}
	if(phase<180&&phase>90){
		result=table[SINE_TABLE_SIZE-
(int)(((phase-90)/90.0)*SINE_TABLE_SIZE)];
	}
	if(phase<270&&phase>180){
		result=-table[(int)(((phase-
180.0)/90.0)*SINE_TABLE_SIZE)];
	}
	if(phase<360&&phase>270){
		result=-table[SINE_TABLE_SIZE-
(int)(((phase-270)/90.0)*SINE_TABLE_SIZE)];
	}
	if(phase==90)
		result=table[SINE_TABLE_SIZE-
1];
	if(phase==270)
		result=-table[SINE_TABLE_SIZE-
1];
	if(phase==360||phase==180)
		result=table[0];
	phase+=step;
	if(phase>=360){
	phase=phase-360;
	}
	return result;
			}
```