EE3-19 Real Time Digital Signal Processing Lab 2

February 17

2013

Declaration: I confirm that this submission is my 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 In it, I give references and citations whenever I refer to or use the published, or unpublished, work of others.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 examinations policy. RICHARD BENNETT

Lab 2

Lab 2

Question 1:

Trace Table for Sinegen

Values	A0	B0	X0	Y0	Y1	Y2
Before First Loop	1.4142	0.7071	1	0	0	0
After 1 Loop	1.4142	0.7071	0	0.7071	0.7071	0
After 2 Loops	1.4142	0.7071	0	0.9999807	0.9999807	0.7071
After 3 Loops	1.4142	0.7071	0	0.7070728	0.7070728	0.9999807
After 4 Loops	1.4142	0.7071	0	-3.8445e-05	-3.8445e-05	0.7070728
After 5 Loops	1.4142	0.7071	0	-0.7071272	-0.7071272	-3.8445e-05
After 6 Loops	1.4142	0.7071	0	-0.9999807	-0.9999807	-0.7071272
After 7 Loops	1.4142	0.7071	0	-0.7070456	-0.7070456	-0.9999807
After 8 Loops	1.4142	0.7071	0	7.688999e-05	7.688999e-05	-0.7070456
After 9 Loops	1.4142	0.7071	0	0.7071543	0.7071543	7.688999e-05
After 10 Loops	1.4142	0.7071	0	0.9999807	0.9999807	0.7071543

This takes 8 loops (samples) to complete a whole cycle.

Question 2:

The output of the sine wave is fixed at 1kHz as it takes 8 loops of the sinegen function to generate one wave, so with a sample frequency of 8kHz this means the output can't be faster than 1kHz.

Samples are not output as fast as possible due to time delays introduced by polling the audio channels to check if they are ready to receive data. However, outputting the samples as fast as possible would be unproductive as the difference between samples wouldn't be constant and wouldn't create a smooth sine wave. The registers throttle it to 1kHz.

Question 3:

32 bits are used to encode each sample that is sent to the audio port – this can be seen in the main code loop below, as the sample multiplied by gain is cast to a 32bit integer.

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

Code Operation

To explain the operation of the code in the sine.c file, we should start with the main code loop and explain each function as we come to it.

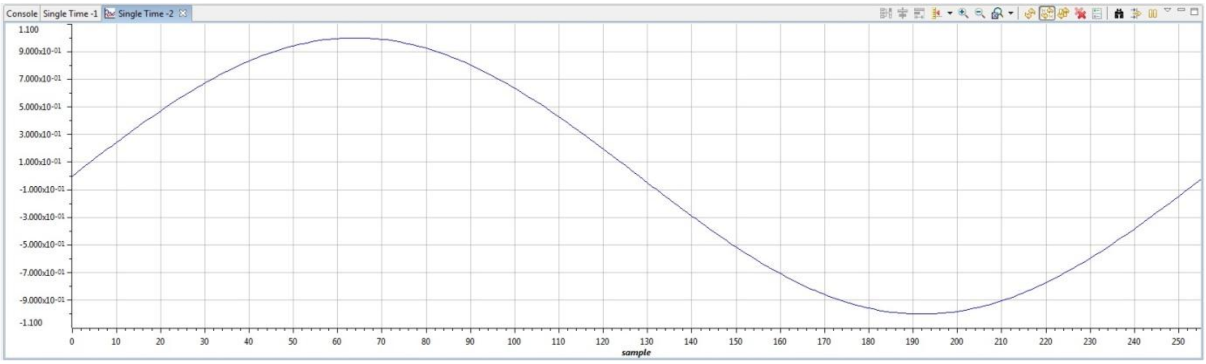
The first function to be called is “init_hardware” which contains code provided to us in the lab. First, the board support library is initialised then the audio codec is started using the “Config” settings. Next the number of bits in the word used by MSBSP for communication with AIC23 is defined and finally the sampling frequency of the audio port is set. This function returns void.

The second function to be called is “sine_init” which fills a lookup table with 256 sine wave data points for one cycle. This function uses a FOR loop which populates the table with sine wave data points based on this equation:

sin(((2*PI)/SINE_TABLE_SIZE)*i)

the use of the #define SINE_TABLE_SIZE enables us to easily change it’s value when if we want to create a larger table to increase resolution.

Graph of data in the array “table”



The third function to be called is “sine_gen” which produces a variable frequency sine wave using a sine lookup table. First we initialise a variable to store the sinewave data point values in, next using the value of step (a global variable initialised to 0) we assign the value of our sine wave table addressed by step to wave. The number of datapoints required to create the wave is decided by this equation:

(SINE_TABLE_SIZE/(sampling_freq/sine_freq))

so the number of data points produced varies with the sampling frequency and sine frequency which enables the processor to output varying frequency sinewaves accurately. Finally, an IF loop is used to prevent the step value trying to address a value greater than the sine table size.

The resolution of the output could be increased by utilising the fact that sine waves are symmetrical, and can be split up into 4 identical sections, which are reflections about the x and y axis. This has been implemented in the sinegen4 function.

Breaking down the sine wave into four quadrants, positive y-axis increasing and decreasing and negative y-axis increasing and decreasing, we can see that these waves are all identical. From this observation it is possible to use just a quarter sine wave look up table. To keep the same resolution and save memory we could just use a 64-value look up table, however if we want to increase resolution we could use the same table size but take values across a quarter sine wave.

To create the quarter sine wave lookup table we just need to adjust the previous sine wave creation formulae to one quarter of the range - **sin(((0.5*PI)/SINE_TABLE_SIZE)*i)**

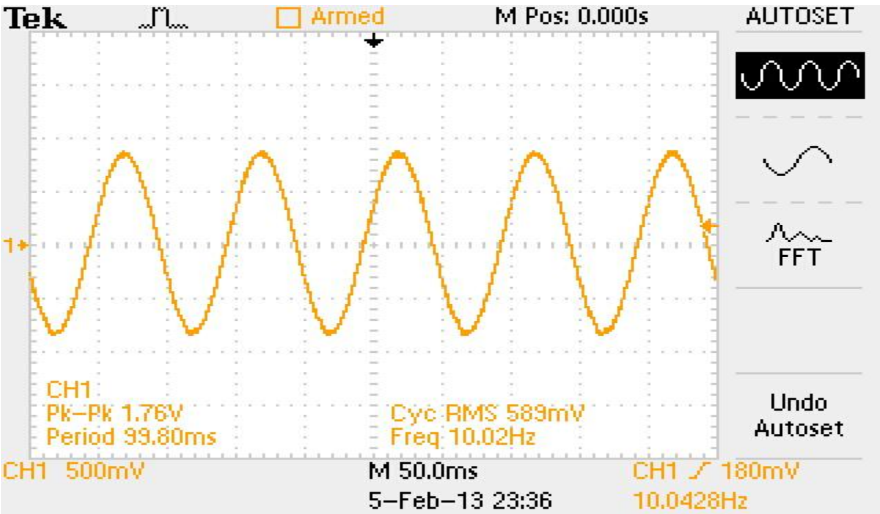
Generating the sine wave from a quarter sine wave look up table requires a few more operations than from a full sine wave look up table as we must account for the reflection and inversion of values. We consider the able to be 4 times longer and then adjust the step value accordingly, based on the sine wave quadrant.

A further way to be able to improve the resolution of the sine wave would be to perform linear interpolation on the values using this formula -

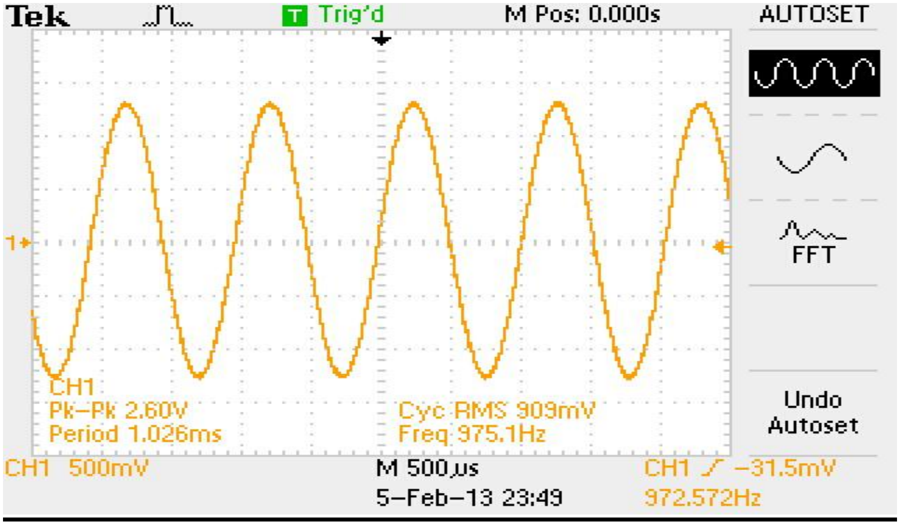
$$y = y_0 + (y_1 - y_0)\frac{x - x_0}{x_1 - x_0}$$

Source: http://en.wikipedia.org/wiki/Linear_interpolation

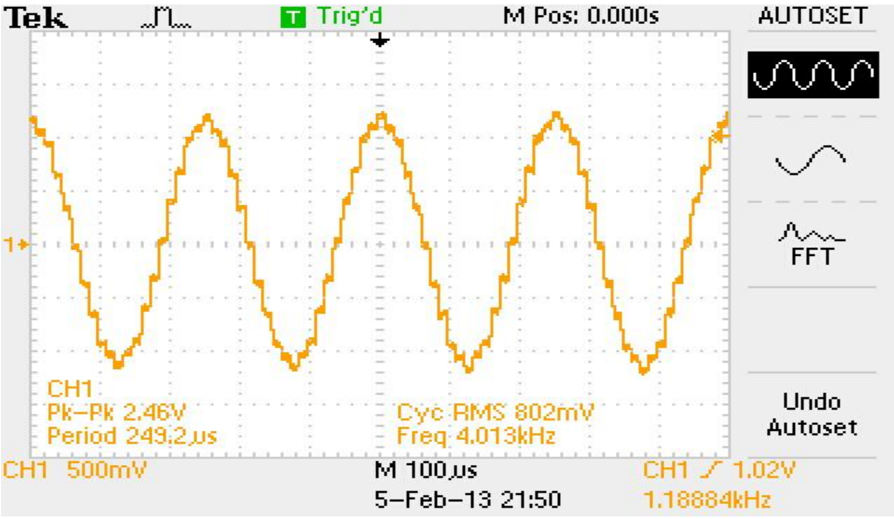
Scope Traces for Full Sine Wave Look up table



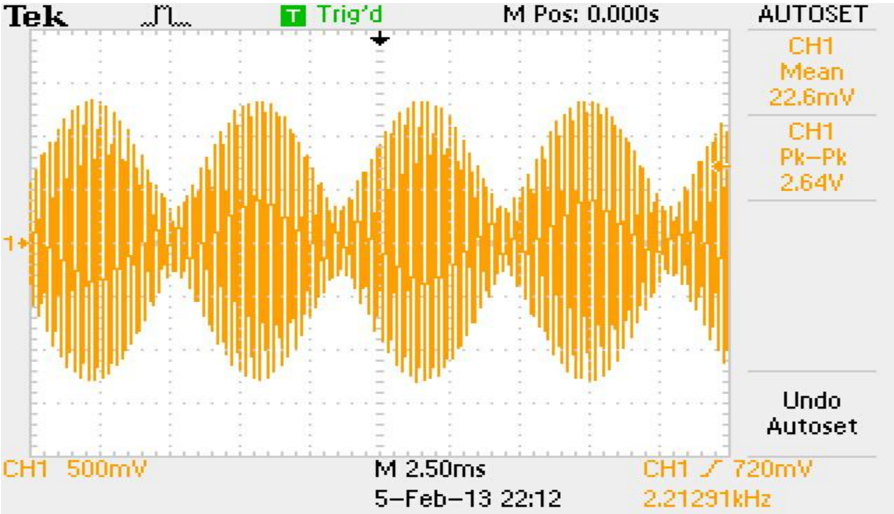
Sampling rate 8kHz Sine wave frequency 10HZ



Sampling Rate 8kHz Sine Wave Frequency 7kHz – as we can see the code can't cope with frequencies over the nyquist frequency.

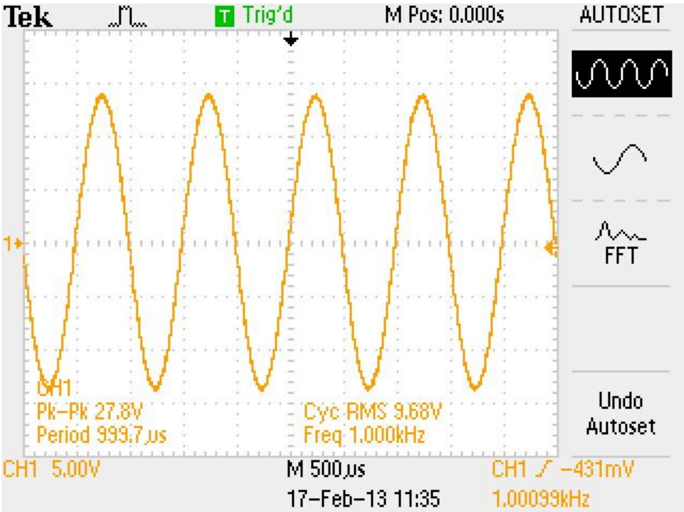


Sampling Rate 8kHz Sine Wave Frequency 4kHz. (Nyquist frequency)

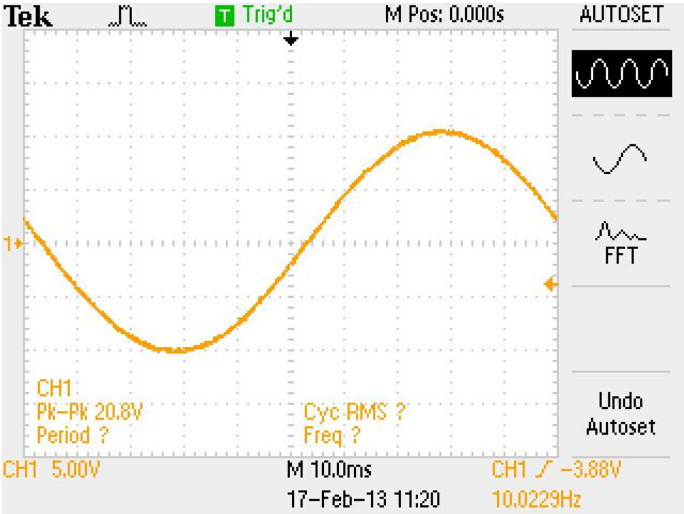


Sampling Rate 8kHz, Sine Wave Frequency 3.95kHz,

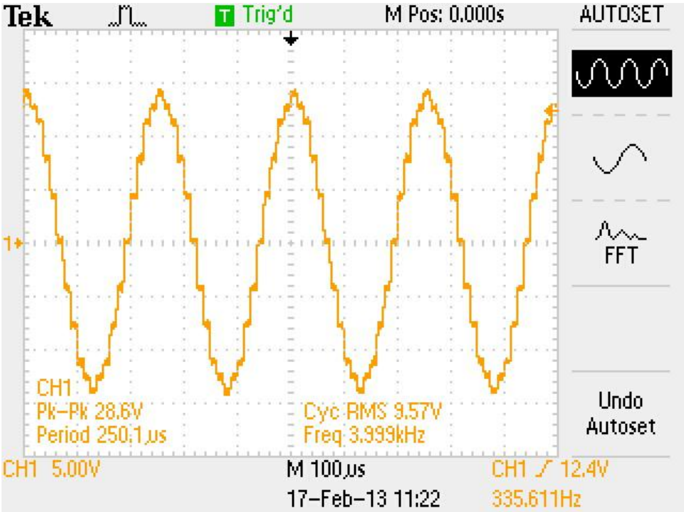
Scope Traces for quarter Sine Wave Look up table



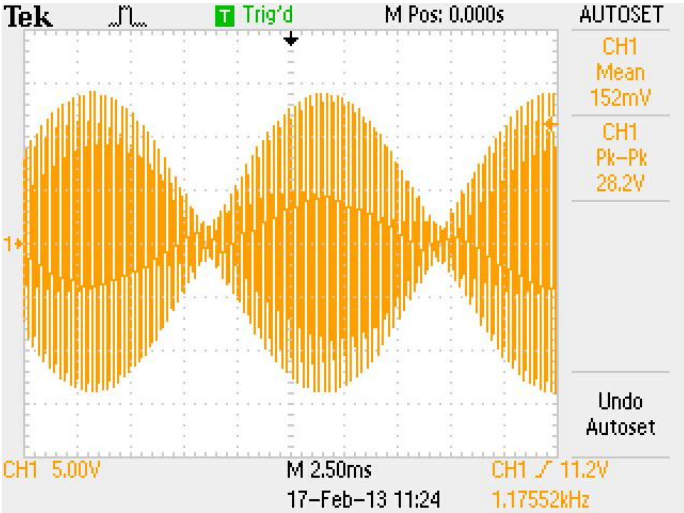
Sampling Rate 8kHz, Sine Wave Frequency 1kHz



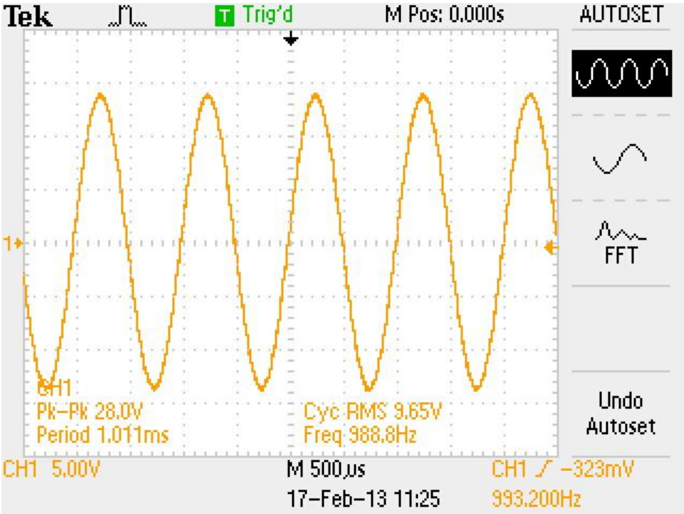
Sampling Rate 8kHz, Sine Wave Frequency 10Hz



Sampling Rate 8kHz, Sine Wave Frequency 4kHz



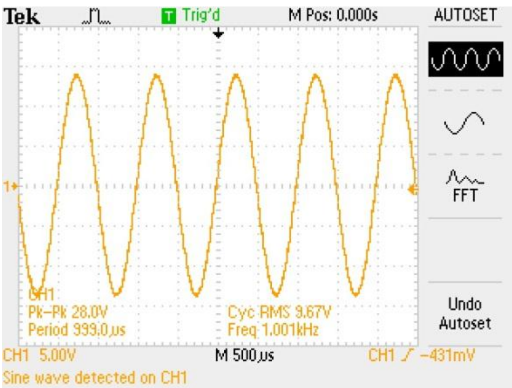
Sampling Rate 8kHz, Sine Wave Frequency 3.95Hz



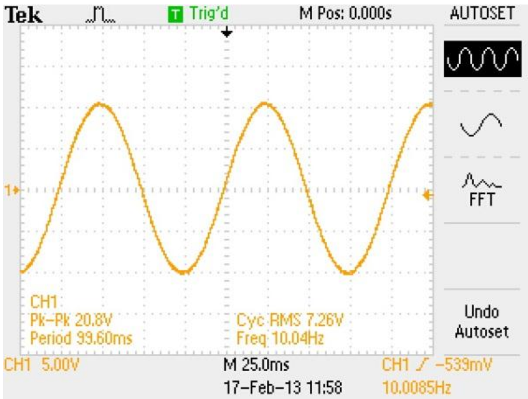
Sampling Rate 8kHz, Sine Wave Frequency 7kHz

Limitations

At low frequencies noise is introduced to the wave and the Peak-Peak and RMS voltages are decreased, as can be seen below, this is caused by a high pass filter (DC blocking capacitor) at the output of the DSK board.

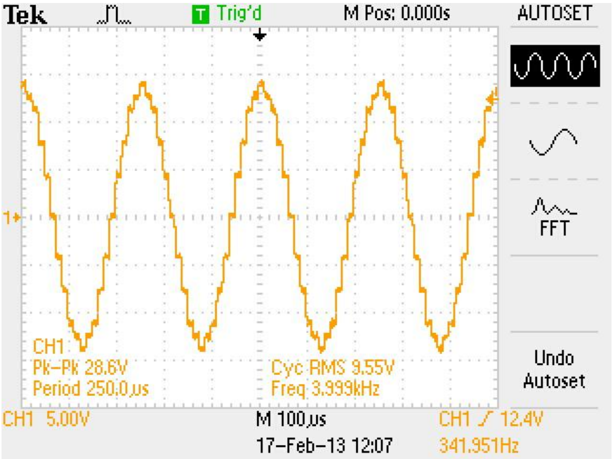


Sampling Frequency 8kHz Sine Frequency 1kHz



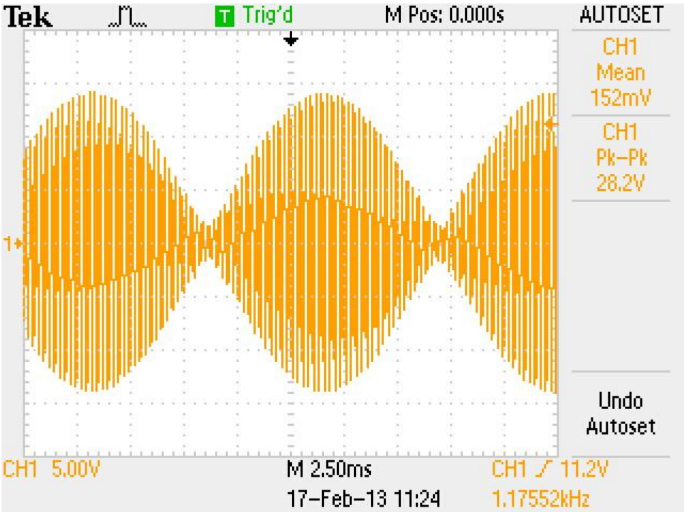
Sampling Frequency 8kHz Sine Frequency 10Hz

For high frequencies – at the nyquist frequency we can see a stepping effect. This is due to the minimum possible values being taken to create a sine wave at this point, which means the DSP has to approximate the lines in between the points – as there are less points the more visible steps are.



Sampling Frequency 8kHz Sine Frequency 4kHz

At values just below the nyquist frequency, e.g. 3.95kHz, an amplitude modulated wave is observed.



As the sampling window at the nyquist frequency is not perfect, we get 2 frequencies from either side of our nyquist frequency appearing in our sine wave. As the frequencies are very close to each other they constantly go in and out of phase which causes them to appear in our waveform. As the amplitudes at these points are not identical and are constantly changing we get a maximum when the two largest amplitudes combine and a minimum when the two smallest amplitudes combine. The frequency of the amplitude modulated wave generated is the frequency at which they go out of phase with each other.

```
/*************************************************************************
***********
            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_TABLE_SIZE - number of values in the sine lookup table - defined
here for use in the code
#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;

/* 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;
```

```
// An array of floats containing SINE_TABLE_SIZE elements
float table[SINE_TABLE_SIZE];

// Step variable for use in the sinegen function
float step = 0;

// Step variable for use in the quarter sinegen function
float step4 = 0;

// Modified step value to within Sine wave table values - for use in
quarter sine wave table
float table_value;


/*
// Sine Value returned Linear-Interpolation function
int interpolate = 0;
*/
/******************************** Function prototypes*******************/
void init_hardware(void); // Hardware initialisation
float sinegen(void); // Sinewave generation
void sine_init(void);   // Initalising sinewave lookup table.
float sinegen4(void); // Sinewave generation from quarter sinewave
void sine_init4(void);  // Initalising quarter sinewave lookup table.
//int linear_interpolate(void); // Linear interpolation for quarter
sinewave function

/*********************************** Main routine **********************/
void main()
{
      // initialize board and the audio port
      init_hardware();

      // initialise table of sinewave data
      //sine_init();

      // initialises table of one quarter sinewave data
      sine_init4();

    // Loop endlessley generating a sine wave
   while(1)
    {
            /* Calculate next sample from full sinewave data table
            sample = sinegen();*/

            // Calculate next sample from quarter sinewave data table
            sample = sinegen4();

      /* 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 variable frequency sine wave, using a sine
lookup table.

      float wave; // Initialise a variable to store the sine wave datapoint
values in.
      wave = table[(int)step]; // Assign wave a value from the sine lookup
table
      // assign step a value based on sampling frequency and desired output
frequency to calculate next table value required.
      step = (SINE_TABLE_SIZE/(sampling_freq/sine_freq)) + step;

//To prevent step containing values greater than SINE_TABLE_SIZE-1
      if (step > (SINE_TABLE_SIZE-1))
      {
            step = step - (SINE_TABLE_SIZE-1);
      }

      return wave;
}

/********************************** sinegen4() ************************/
float sinegen4(void)
{
      /*  This code produces a variable frequency sine wave, using a
quarter-sine-wave lookup table.
       *
       * */
```

```
	float wave4 = 0; // Initialise a variable to store the sine wave 
datapoint values in.

	// To create a sine wave from the quarter sinewave table data.
	//For values in the first sinewave quadrant - no adjustment to the 
step value needs to be made.
	if (step4 < (SINE_TABLE_SIZE))
	{
		table_value = step4;
	//	wave4 = linear_interpolate();
		wave4 = table[(int)step4];
	}

	//Second quadrant - step value must be adjusted to bring the value 
back into the range 0-255
	else if (step4 < (2*SINE_TABLE_SIZE) && (step4 >= SINE_TABLE_SIZE))
	{
		table_value = ((SINE_TABLE_SIZE-1)-(step4-SINE_TABLE_SIZE));
	//	wave4 = linear_interpolate();
		wave4 = table[(int)((SINE_TABLE_SIZE-1)-(step4-
SINE_TABLE_SIZE))];
	}
	//Third quadrant - step value must be adjusted to bring the value 
back into the range 0-255 and the wave value negated
	else if (step4 < (3*SINE_TABLE_SIZE) && (step4 >= 
(2*SINE_TABLE_SIZE)) )
	{
		table_value = (step4-(2*SINE_TABLE_SIZE));
	//	wave4 = -linear_interpolate();
		wave4 = -table[(int)(step4-(2*SINE_TABLE_SIZE))];
	}

	//Fourth quadrant - step value must be adjusted to bring the value 
back into the range 0-255 and the wave value negated
	else if (step4 < (4*SINE_TABLE_SIZE) && (step4 >= 
(3*SINE_TABLE_SIZE))  )
	{
		table_value = ((SINE_TABLE_SIZE-1)-(step4-
(3*SINE_TABLE_SIZE)));
	//	wave4 = -linear_interpolate();
		wave4 = -table[(int)((SINE_TABLE_SIZE-1)-(step4-
(3*SINE_TABLE_SIZE)))];
	}

	// assign step a value based on sampling frequency and desired output 
frequency to calculate next table value required.
	step4 += ((4*SINE_TABLE_SIZE)/(sampling_freq/sine_freq));

	//To prevent step containing values greater than 4*SINE_TABLE_SIZE-1 
which would cause the operation to overflow.
	if (step4 > ((4*SINE_TABLE_SIZE-1)))
	{
		step4 = step4 - (4*SINE_TABLE_SIZE-1);
	}

	return wave4;
}

/********************************** sine_init()************************/
```

```
void sine_init()
{
      // Fill the lookup table with 256 sine data points  across one cycle.
      int i;
      for(i=0; i < SINE_TABLE_SIZE; i++)
      {
            table[i] = sin(((2*PI)/SINE_TABLE_SIZE)*i);
      }

}

/********************************** sine_init4()************************/

void sine_init4()
{
      // Fill the lookup table with 256 sine data points  across one
quarter cycle.
      int j;
      for(j=0; j < SINE_TABLE_SIZE; j++)
      {
            table[j] = sin(((0.5*PI*j)/SINE_TABLE_SIZE));
      }

}

/****************************linear_interpolate()****************
//Perform linear interpolation on the quarter sine wave.

int linear_interpolate()
{


            int floor_value = floor(table_value);
            int ceiling_value = ceil(table_value);


            float floor_diff = table_value - floor_value;
            int value_diff = ceiling_value - floor_value;
            if (ceiling_value > SINE_TABLE_SIZE)
                  ceiling_value = 0;

            interpolate = table[floor_value] + (((floor_diff *
table[ceiling_value]) - (floor_diff * table[floor_value])) / value_diff);

      return interpolate;



}

*/
```