Imperial College London

Real Time Digital Signal Processing

Lab 2 Report

Andrew Zhou, CID: 00938859

Jagannaath Shiva Letchumanan, CID: 00946740

Declaration: We confirm that this submission is our own work. In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the College examination offenses policy.
Signed: **Andrew Zhou, Jagannaath Shiva Letchumanan**

January 29, 2017

Contents

1 Answers to Questions

1.1 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?

Sample	Value
1	0.7071
2	0.999807
3	0.7070728
4	-3.84E-05
5	-0.7071272
6	-0.09999807
7	-0.7070456
8	7.69E-05
9	0.7071543
10	0.9999807
11	0.7070183
12	-1.15E-04
13	-0.7071815
14	-0.9999807
15	-0.7069911
16	1.53E-04
17	0.7072087

Table 1: 17 samples of the sine wave

Upon running the code with breakpoints right after the call to `sinegen()`, the following trace table was obtained for 17 samples of the sine wave. On carefully inspecting the table (rounding off a few entries), it can be seen that it takes just 8 samples to generate a whole cycle of a sine wave.

1.2 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?

The IIR filter implementation of the sine wave has coefficients calculated for a sine frequency of 1 kHz and a sampling frequency of 8 kHz. However, the calculations associated with them take place at a very high rate as the DSP chip is clocked at 225 MHz, which means that the chip could output values as and when they are ready, producing a frequency much higher than 1kHz.

This does not happen as the DAC is limited to reading samples at 8kHz since we have set the sampling rate while configuring the Codec.

```
/* 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                    */\
                         /**********************************************************************/
};
```

Figure 1: Configuration of Codec of data type DSK6713_AIC23_Config

The code we have implemented is a polling loop that waits until the `DSK6713_AIC23_write(H_Codec, ((Int32)(sample * L_Gain))` function returns a '1'. So, every 125 μs (corresponding to 8 kHz), the DAC reads new data from the buffer of the McBSP serial port. Now, the buffer is free to receive new data and the above function can successfully write to it, thereby ending the polling loop and allowing `sinegen()` to calculate the next value. If the data has not been extracted from the buffer, the polling loop will continue running.

As it takes 8 samples (as can be seen above) to generate an entire cycle of the sine wave, the frequency is in this manner throttled to 1 kHz.

1.3 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?

`(Int32)(sample * L_Gain)`
Although `sample` is a 32-bit floating point number limited to between -1 and 1, it is multiplied by another 32-bit integer (left and right audio channel gains) and the end result is parsed as a 32-bit integer. This is done to feed values to the audio ports that they can process. Hence, each sample is encoded using **32 bits**. Moreover, this can be confirmed by looking at number 7 in the `Config` structure (Figure 1) where we have set the number of bits to 32.

2 Working of Code

The code provided produces a sine wave by calculating values from a difference equation in real time i.e. it is a IIR filter realisation (explained in Appendix A). In contrast, the updated code calculates all required points of the sine wave initially, stores them in a lookup table and accesses these values when they are to be output.

In order to do this, a global symbolic constant `SINE_TABLE_SIZE` and a global variable `table` were initialised. `table` is an array of floats with `SINE_TABLE_SIZE` elements, and is filled with values taken at 256 equally spaced points across one period of a sine wave. This is done by the `sine_init()` function (Figure 2) which is called once in the main function, before the infinite while loop. The values are calculated by utilising the given global symbolic constant `PI` and the `sin()` function defined in the header file, `math.h`.

```
//fills table with values of 256 equally spaced points around the sine wave
void sine_init()
{
    int x;
    for(x=0; x<SINE_TABLE_SIZE; x++){
        table[x] = sin((2*PI*x)/SINE_TABLE_SIZE);
    }
}
```

Figure 2: Initialization of the look-up table

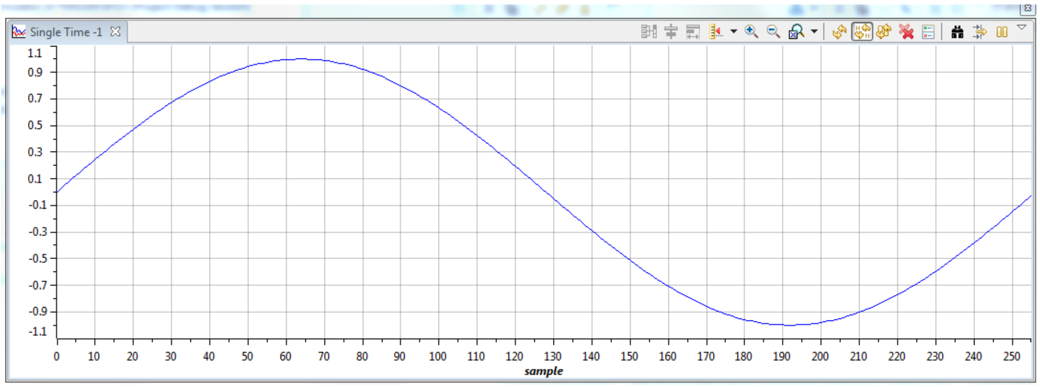


Figure 3: Graph of the array of points in `table`

Originally, the `sinegen()` function calculated and returned the result of the difference equation, whereas now, it returns the output by reading the corresponding values stored in `table` (Graphed in Figure 3).

```
float sinegen(void)
{
/*  This code produces a fixed sine of 1KHZ (if the sampling frequency is 8KHZ)
    using a digital filter.*/

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

    //index must skip samples in order to maintain correct interpreted frequency
    index += (SINE_TABLE_SIZE*sine_freq/sampling_freq); //offset definition
    if(index>=SINE_TABLE_SIZE)
    {
        //when index exceeds the table size, table size is subtracted from index
        index-=SINE_TABLE_SIZE ;
    }

    //set the output as the value of the sine wave, stored in table
    wave = table[(int)floor(index)];

    return(wave);

}
```

Figure 4: Sine Generation function

As before, `wave` is still the variable used as the output to be returned by `sinegen()`. `index` is a global float that tracks which memory location within `table` should be accessed. This is physically analogous to keeping track of which part of the sine wave is being passed to the output.

If we let the code run as it is now, it would produce a sine wave of frequency 31.25 Hz (8kHz/256) which is independent of the `sine_freq` and `sampling_freq` variables. In order to ensure that the wave is in fact controlled by these parameters, we use an offset of `SINE_TABLE_SIZE*sine_freq/sampling_freq` as this helps remove the dependency on the table size and sampling frequency (8000/256 is multiplied by 256/8000 times the required frequency). `index` is incremented by this offset each time in order to skip an appropriate number of addresses in the array, to maintain the frequency specified in the parameter `sine_freq`. The index and offset calculations are done as floating point numbers because if the offset is a fractional value, we do not want to truncate it while calculating the index. Instead the truncation is done by rounding down and parsing to integer while accessing the table: `wave = table[(int) floor(index)]`.

A repetitive check is performed by the `if` statement from lines 166 to 170, to maintain periodicity of the output waveform, i.e. when `index` exceeds `SINE_TABLE_SIZE`, the latter is subtracted from the former so that existing addresses of `table` are accessed (we circle back to the start). It would be erroneous to set `index` to 0 once it exceeds `SINE_TABLE_SIZE` as this would skip the tail end of the wave and restart at zero giving rise to distortions.

The `sinegen()` function is continually called in the infinite while loop in `main` so that a continuous sine wave is produced. These calls occur at the sampling frequency due to the two polling loops for the left and right audio ports (explained in section 1.2 above).

In this manner, we obtain a sine wave of the required frequency using the code (attached in Appendix B).

The output will not change with a change in sampling frequency (let us assume that it is doubled). This is a consequence of the fact that although the `sinegen()` function will read through the table at half the rate (offset is inversely dependant on the sampling frequency), the polling will occur at twice the rate and hence the two changes will balance out.

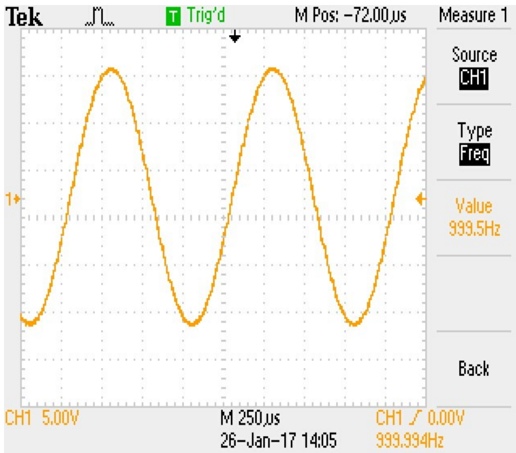
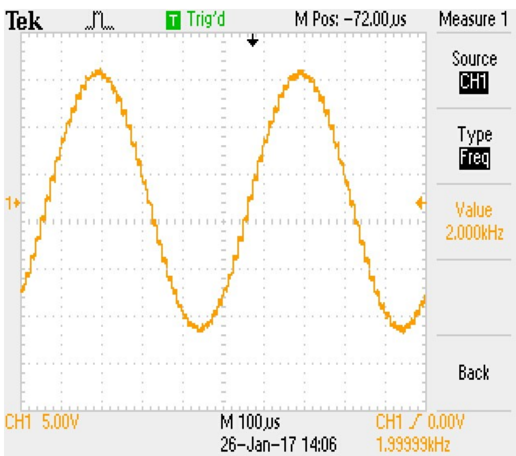
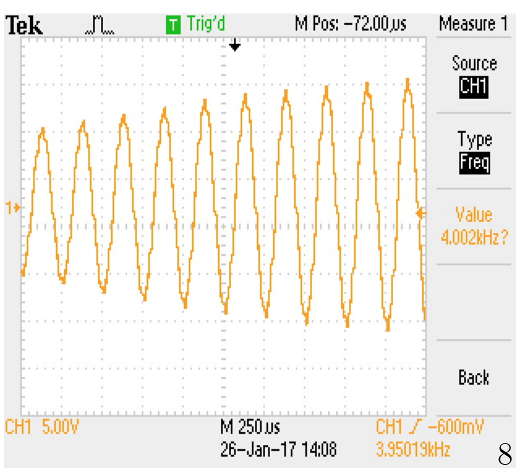
The code is quite **reusable** as all calculations and conditional statements, including loop counter limits, are performed in terms of the symbolic constants `SINE_TABLE_SIZE` and `PI`, and the parameters `sine_freq` and `sampling_freq`. So, any change in these values will not require the code to be rewritten.

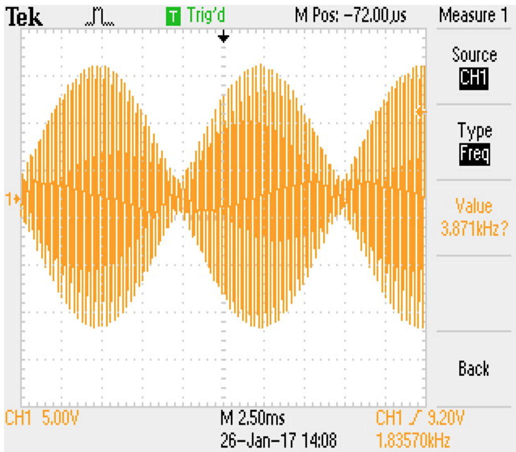
These are the changes introduced to output a sine wave of the specified frequency using a look-up table. The shell of the code, i.e. the polling loops, configuration of the Codec and initialisation of the hardware are the same as what was provided originally.

3 Scope traces illustrating operation of the code

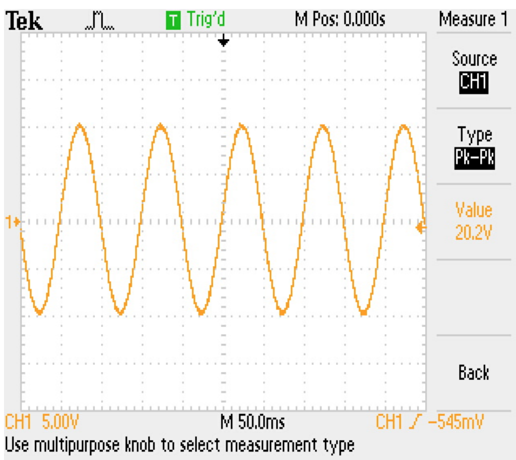
Figure 5 shows a scope trace of the output of the code when `sampling_freq` is set to 8kHz and `sine_freq` is set well below Nyquist sampling rate at 1kHz. The output is as expected, without any distortion or harmonics. Quantisation steps are apparent however, but unavoidable due to the digital nature of the code.

Figure 6 corresponds to `sine_freq` of 2 kHz, figure 7 corresponds to 3.95 kHz, figure 9 to 10 Hz and figure 10 to 5 Hz. The results that deviate from the expected are explained in a later section (section 5).

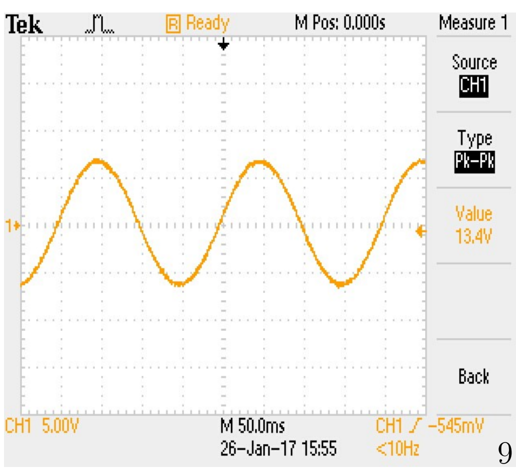
Images	Frequencies and Observations
	• **Figure 5** • `sine_freq` = 1 kHz • `sampling_freq` = 8 kHz • As expected • Peak-peak amplitude is 28V
	• **Figure 6** • `sine_freq` = 2 kHz • `sampling_freq` = 8 kHz • As expected • Peak-peak amplitude is 28V
	• **Figure 7** • `sine_freq` = 3.95 kHz • `sampling_freq` = 8 kHz • Amplitude Variations

<table>
<tr><th>Images</th><th>Frequencies and Observations</th></tr>
<tr><td>
</td><td>

- **Figure 8**
- `sine_freq` = 3.95 kHz
- `sampling_freq` = 8 kHz
- Horizontal axis is zoomed out further compared to Figure 7

</td></tr>
<tr><td>
</td><td>

- **Figure 9**
- `sine_freq` = 10 Hz
- `sampling_freq` = 8 kHz
- Peak to peak amplitude is 20V in comparison to 28V as seen in previous figures

</td></tr>
<tr><td>
</td><td>

- **Figure 10**
- `sine_freq` = 5 Hz
- `sampling_freq` = 8 kHz
- Peak to peak amplitude is 13.4V in comparison to 28V as seen in previous figures

</td></tr>
</table>

4 Improving Resolution

The symmetrical properties of a sine wave can be exploited to increase resolution of the output without increasing `SINE_TABLE_SIZE`. As the first and second halves of a sine wave are identical in magnitude, `sine_init()` can be altered to take 256 equally spaced points across only half a sine wave, effectively doubling resolution over half a wave.

```
table[x] = sin((PI*x)/SINE_TABLE_SIZE);
```

Figure 11: Improving resolution by a factor of 2

To produce a full sine wave, `sinegen()` must be modified to reverse the sign of the output after every half wave, effectively after `table` has been traversed fully and this is done using the `sign` global variable. The offset must be doubled to again ensure that the frequency of the sine wave is controlled by `sine_freq` and `sampling_freq`. The resolution will improve as now we have a larger range of values (incrementing by smaller steps) to step through. This improvement can mainly be observed for sine wave frequencies that result in fractional offsets.

```
//reverses sign if index exceeds 256
if(index>=SINE_TABLE_SIZE)
{
        //when index exceeds the table size, subtract table size from index to continue the wave
        index-=SINE_TABLE_SIZE;
        sign = -sign;
}

//set the output as the value of the sine wave, stored in table, and set sign.
wave = sign*table[(int)floor(index)];

return(wave);
```

Figure 12: Modified if statement and return value for improving resolution

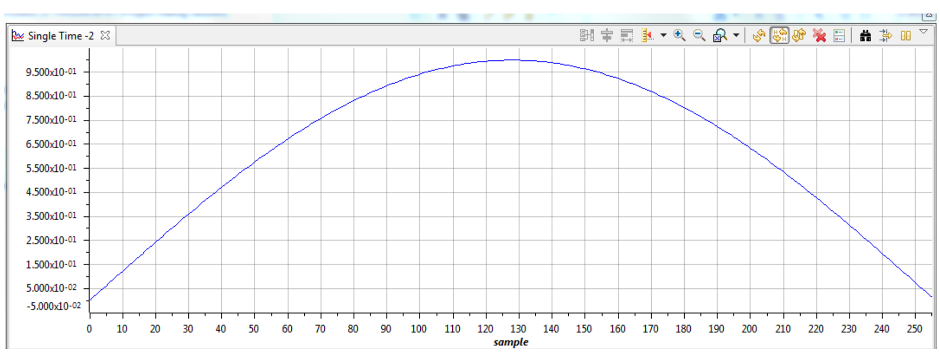


Figure 13: Graph of the new array of points in `table`

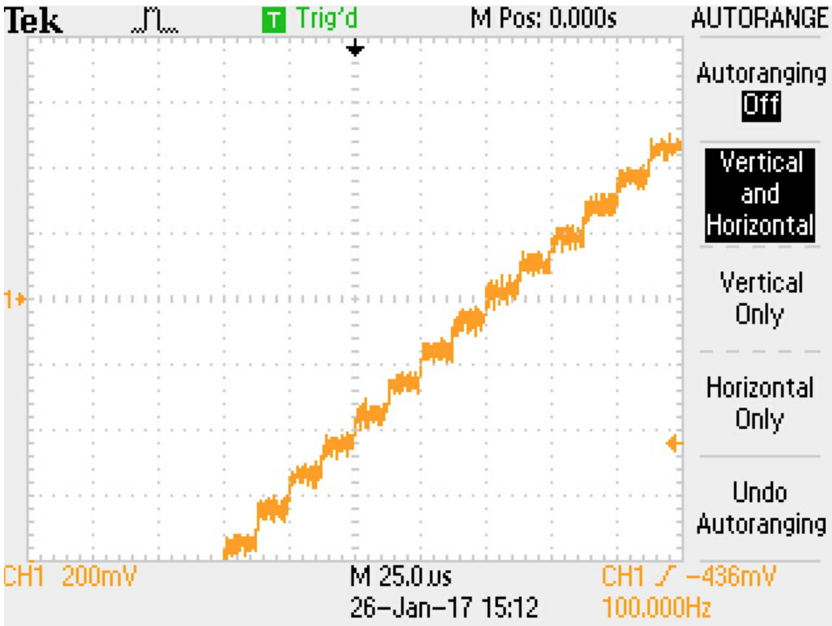


Figure 14: Original quantisation step for 100 Hz

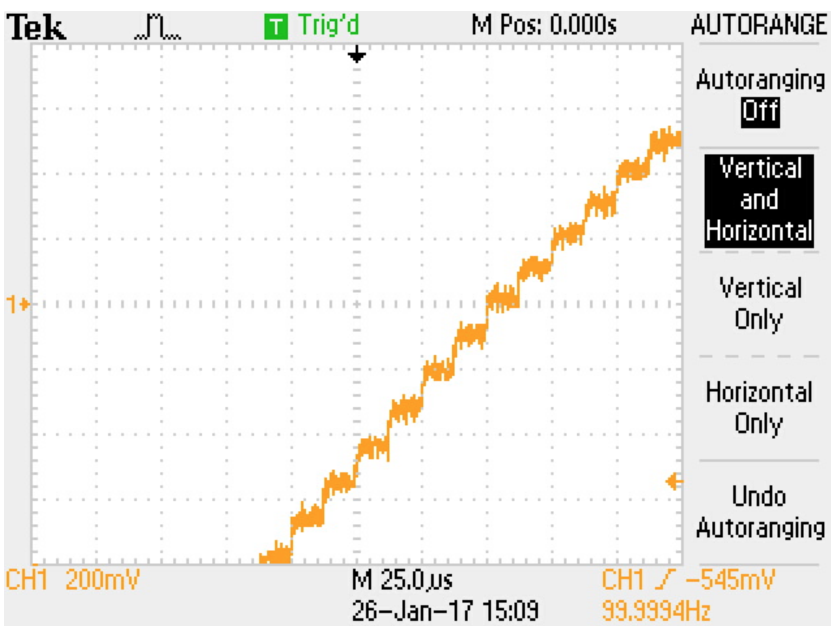


Figure 15: Improved quantisation step for 100 Hz

Under close examination of the two oscilloscope traces above, it can be noted that the latter image has a smaller amount of quantisation noise about the steps confirming that the resolution has in fact been improved. This is more pronounced in frequencies that result in very small offsets such as 100Hz.

Resolution can be further improved by taking equally spaced values across only a quarter of a sine wave exploiting the symmetry further and increasing the offset by a factor of 4 instead. Then the direction that index varies in can be alternated every time `table` has been traversed fully and the sign of the output can be reversed at half this frequency. A possible implementation is shown in the code below:

```
float sinegen(void)
{
/*  This code produces a fixed half sine wave of 2KHZ (if the sampling frequency is 8KHZ)
    using a digital filter, with the sign of the wave reversed after each cycle(half wave).
    This is done with the purpose of increasing resolution*/

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

    //index must skip samples in order to maintain correct interpreted frequency
    index += inc_sign*(SINE_TABLE_SIZE*4*sine_freq/sampling_freq); //offset definition

    //reverses sign if index exceeds 256
    if(index>=SINE_TABLE_SIZE)
    {
            //when index exceeds the table size, subtract table size from index to continue the wave
            index = 2*SINE_TABLE_SIZE - index - 1;
            inc_sign = -1;
    }
    if(index<=0)
    {
        index = -index;
        inc_sign = 1;
        sign = -sign;
    }

    //set the output as the value of the sine wave, stored in table, and set sign.
    wave = sign*table[(int)floor(index)];

    return(wave);

}

//fills table with values of 256 equally spaced points around half of the sine wave
void sine_init()
{
    int x;
    for(x=0; x<SINE_TABLE_SIZE; x++){
        table[x] = sin(((PI/2)*x)/SINE_TABLE_SIZE);
    }
}
```

Figure 16: Implementation of a quarter wave 256 element look up table

Beyond a factor of 4, it is not possible to exploit sine wave symmetry any further. However, resolution can still be improved by interpolating between the points in case of fractional index (which is quite likely). The simplest interpolation would be linear but non-linear interpolations that take into account the curvature of the sine wave would be better. These interpolations improve the resolution by further reducing the quantisation noise.

As possible implementation of a linear interpolator would be:

```
//calculation for interpolation
deltay = table[(int)ceil(index)] - table[(int)floor(index)];
deltax = index - floor(index);
wave = table[(int)floor(index)] + (deltay*deltax);
```

Figure 17: Linear Interpolator Implementation

However, interpolation is not always good. This is due to the fact that the chip has to do more processing and the step size may not be unifrom, which could cause the computations to slow down. Nevertheless, this would not matter as long as the overall computation performed can be completed before the McBSP serial port asks for the next data (computations must be done faster than the sampling frequency).

5 Range of Frequencies

As can be seen in the scope traces above, the amplitude of the sine wave output is constant at around 28V (most likely determined by the left and right audio channel gains). However, there seem to be upper and lower bounds of operation outside of which the behaviour is not as expected.

For frequencies very close to the Nyquist, a sine wave of similar amplitude, i.e. 28V is expected to be produced. Unexpectedly, testing showed that an amplitude modulated signal with an envelope of much smaller frequency was in fact produced. For instance, if we try to output a sine wave at 3.95 kHz with a sampling frequency of 8kHz, we will see a 3.95 kHz AM signal (Figure 7) with a 50Hz envelope (Figure 8).

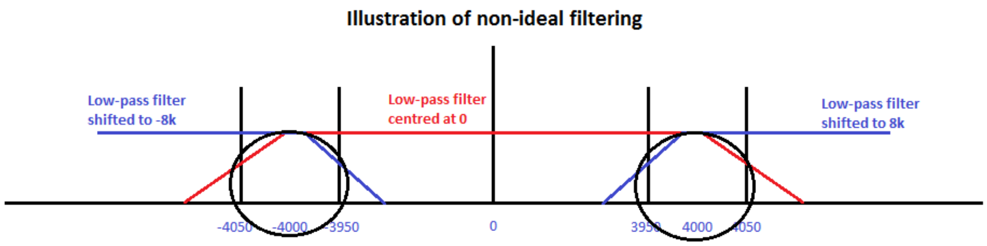


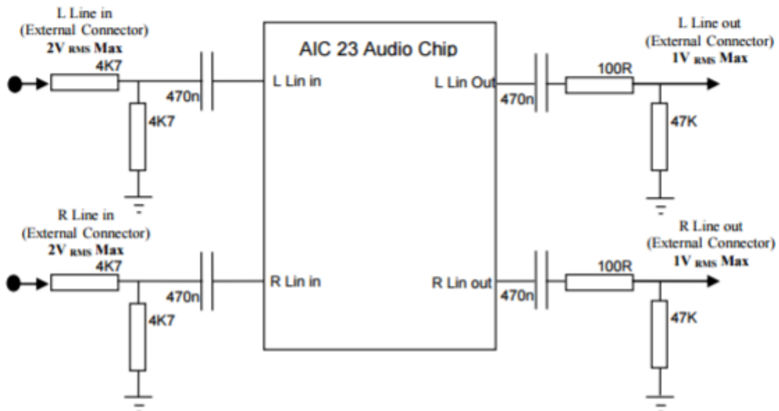
Figure 18: Illustration of non-ideal filtering

This **aliasing** effect is caused by the fact that the low pass filters used for reconstruction are not ideal, i.e. their cut-off is not vertical. If we consider, once again, the case of the 3.95 kHz sine wave, it's spectrum will be two Dirac deltas at +/- 3.95 kHz. However, since we sample at 8 kHz, we will get components at +/- 4.05 kHz as well (+/-(8kHz-3.95kHz)) and this will repeat, centred at all multiples of the sampling frequency. Since the low pass reconstruction filter (in red) is not ideal, the output will include components at +/- 4.05 kHz as well. As a result, the output has frequencies at 4000+/-50 Hz, which results in an AM signal with the 'carrier' at 3.95 kHz.

For the case of the lower bound, the AIC 23 Audio Chip has filters at both the input and output which serve the purpose of filtering any DC signals or offsets from going through to the output. This is desirable as it prevents a constant hum from the output and also avoids damage to speakers (if any) from overheating.

The output filter is a high pass filter with a cut-off frequency at 7.19 Hz.

$$f = \frac{1}{2\pi * RC} = \frac{1}{2\pi * (47000 + 100) * 470 * 10^{-9}} = 7.189Hz$$



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

Figure 19: Filters at input and output of Audio Chip

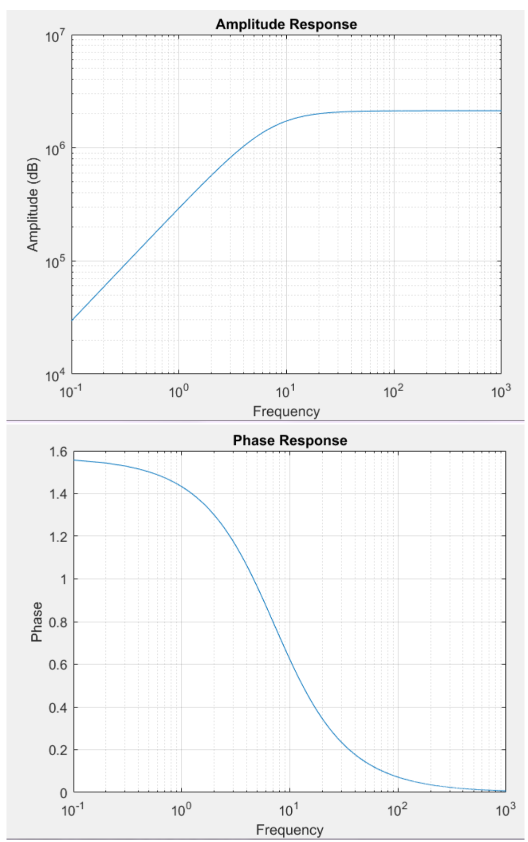


Figure 20: Amplitude and Phase Response of Output Filter

From the Amplitude response above, it can be seen that for any frequency below 20 Hz (approximately), the gain of the filter is a lot lower and can damp the amplitude of the output sine wave which can be seen in Figures 9 and 10. This frequency is actually set so that it is right below the minimum audible frequency of an average human ear which is 20 Hz. As a result, it is just the sub-sonic frequencies that are filtered out.

Hence, the range of frequencies of the system, i.e. the frequencies over which the output is of constant amplitude, is around **20 Hz to 95% of the Nyquist rate** (half the sampling frequency).

References

[1] Erik Cheever. Laplace and z transforms. http://lpsa.swarthmore.edu/LaplaceZTable/LaplaceZFuncTable.html, 2005. [Online; accessed 2017-01-28].

Appendices

A IIR Sine Wave Difference Equation

A sine wave has two poles that are located on the unit circle, hence the function is marginally stable. The transfer function for a system that creates a sine wave of amplitude 1 (obtained from a table of z-transforms [1]) is:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{z^2 sin(\omega_0)}{z^2 - 2cos(\omega_0)z + 1} \text{ where } \omega = \frac{2\pi * f_0}{f_s}$$

$$f_0 = \text{frequency of the sine wave}$$

$$f_s = \text{sampling frequency}$$

Dividing throughout by z^2, we get:

$$\frac{Y(z)}{X(z)} = \frac{sin(\omega_0)}{1 - 2cos(\omega_0)z^{-1} + z^{-2}}$$

Multiplying out, we get:

$$Y(z) - 2cos(\omega_0)z^{-1}Y(z) + z^{-2}Y(z) = sin(\omega_0)X(z)$$

Taking the inverse z-transform:

$$y[n] - 2cos(\omega_0) * y[n-1] + y[n-2] = sin(\omega_0) * x[n]$$

Rearranging:

$$y[n] = 2cos(\omega_0) * y[n-1] - y[n-2] + sin(\omega_0) * x[n]$$
$$y[n] = a_0 * y[n-1] - a_1 * y[n-2] + b_0 sin(\omega_0) * x[n]$$

For the case of a 1 kHz sine wave with a sampling frequency of 8 kHz,

$$\omega_0 = \frac{2\pi * 1000}{8000} = \frac{\pi}{4}$$
$$a_0 = 2 * cos(\omega_0) = \sqrt{2}$$
$$a_1 = 1 \; b_0 = sin(\omega_0) = \frac{1}{\sqrt{2}}$$

This is the IIR (Infinite Impulse Response) difference equation for a sine wave, which is what was used by the `sine.c` file provided originally.

B Full `sine.c` code (excluding improvements to resolution)

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

// Holds the value of the current sample
float sample;

//Index number
float index;

/* 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;

//Table containing values of sine wave
float table[SINE_TABLE_SIZE];

/******************************** Function prototypes ********************************/
void init_hardware(void);
float sinegen(void);
void sine_init();
/*********************************** Main routine ************************************/
void main()
{
index = 0;
    // initialize board and the audio port
    init_hardware();

    // initialise the sine 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()
{
    // 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.*/

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

    //index must skip samples in order to maintain correct interpreted frequency
    index += (SINE_TABLE_SIZE*sine_freq/sampling_freq); //offset definition
    if(index>=SINE_TABLE_SIZE)
    {
        //when index exceeds the table size, table size is subtracted from index
        index-=SINE_TABLE_SIZE ;
    }

    //set the output as the value of the sine wave, stored in table
    wave = table[(int)floor(index)];

    return(wave);

}

//fills table with values of 256 equally spaced points around the sine wave
void sine_init()
{
    int x;
    for(x=0; x<SINE_TABLE_SIZE; x++){
        table[x] = sin((2*PI*x)/SINE_TABLE_SIZE);
    }
}
```

Listing 1: `Full sine.c` code

C sine.c with frequency resolution (half wave)

```c
/*************************************************************************************
                   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
#define SINE_TABLE_SIZE 256

/******************************** Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
```

```c
   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;

// Holds the value of the current sample
float sample;

//Index number
float index;

//Sign dictating the positive or negative cycle of the sine wave
int sign=1;

/* 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;

//Table containing values of sine wave
float table[SINE_TABLE_SIZE];

/******************************** Function prototypes ********************************/
void init_hardware(void);
float sinegen(void);
void sine_init();
/************************************ Main routine ************************************/
void main()
{
index = 0;
    // initialize board and the audio port
    init_hardware();

    // initialise the sine 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()
{
    // 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 half sine wave of 2KHZ (if the sampling frequency is 8KHZ)
    using a digital filter, with the sign of the wave reversed after each cycle(half wave).
    This is done with the purpose of increasing resolution*/

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

    //index must skip samples in order to maintain correct interpreted frequency
    index += (SINE_TABLE_SIZE*2*sine_freq/sampling_freq); //offset definition

```

```c
    //reverses sign if index exceeds 256
    if(index>=SINE_TABLE_SIZE)
    {
        //when index exceeds the table size, subtract table size from index to continue the wave
        index-=SINE_TABLE_SIZE;
        sign = -sign;
    }


    //set the output as the value of the sine wave, stored in table, and set sign.
    wave = sign*table[(int)floor(index)];

    return(wave);

}

//fills table with values of 256 equally spaced points around half of the sine wave
void sine_init()
{
    int x;
    for(x=0; x<SINE_TABLE_SIZE; x++){
        table[x] = sin((PI*x)/SINE_TABLE_SIZE);
    }
}


```

Listing 2: `Full sine.c` code with half-wave resolution improvement

D sine.c with frequency resolution (quarter wave)

```
/*************************************************************************************
                  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
#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;

// Holds the value of the current sample
float sample;

//Index number
float index;

//Sign dictating the positive or negative cycle of the sine wave
int sign=1;

//sign dictating the direction in which index increments
int inc_sign=1;

/* 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;

//Table containing values of sine wave
float table[SINE_TABLE_SIZE];

/******************************** Function prototypes ********************************/
void init_hardware(void);
float sinegen(void);
void sine_init();
/*********************************** Main routine ************************************/
void main()
{
index = 0;
    // initialize board and the audio port
    init_hardware();
```

```
    // initialise the sine 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()
{
    // 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 half sine wave of 2KHZ (if the sampling frequency is 8KHZ)
    using a digital filter, with the sign of the wave reversed after each cycle(half wave).
    This is done with the purpose of increasing resolution*/

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

```c
    //index must skip samples in order to maintain correct interpreted frequency
    index += inc_sign*(SINE_TABLE_SIZE*4*sine_freq/sampling_freq); //offset definition

    //reverses sign if index exceeds 256
    if(index>=SINE_TABLE_SIZE)
    {
            //when index exceeds the table size, subtract table size from index to continue the wave
            index = 2*SINE_TABLE_SIZE - index -1
            inc_sign = -1
    }
    if(index<=0)
    {
        index = -index
        inc_sign = 1
        sign = -sign;
    }

    //set the output as the value of the sine wave, stored in table, and set sign.
    wave = sign*table[(int)floor(index)];

    return(wave);

}

//fills table with values of 256 equally spaced points around quarter of the sine wave
void sine_init()
{
    int x;
    for(x=0; x<SINE_TABLE_SIZE; x++){
        table[x] = sin((PI*x/2)/SINE_TABLE_SIZE);
    }
}
```

Listing 3: `Full sine.c` code with quarter-wave resolution improvement