RTDSP Lab 2 Report

Edward Stables, CID: 01220379
Jack Waller, CID: 01217931

January 2019

1 Questions

1.1 Question 1

Cycle	y[0]	y[1]	y[2]
0	0	0	0
1	b	b	0
2	1	1	b
3	b	b	1
4	0	0	b
5	-b	-b	0
6	-1	-1	-b
7	-b	-b	-1
8	0	0	b

Table 1: Table of Difference Equation Values, $b = \frac{1}{\sqrt{2}}$

The trace table shows the pattern of values held in the **y** array. Note that due to the buffer being advanced after a new value is assigned to **y[0]**, **y[0]** and **y[1]** always contain the same value. This does not impact the calculation of the sequence. The IIR filter results in a sequence of length 8 samples, giving an approximation of a sinusoid.

1.2 Question 2

The output is updated with the line of code:

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

This code will cause an infinite loop while the DSK6713_AIC23_write function returns false. In the init_hardware function, the sampling frequency is set at whatever value is defined by the user (default of 8kHz). Therefore DSK6713_AIC23_write will return true (meaning a sample has been output) at the sampling frequency. As was shown in the previous question, there are 8 'samples' per cycle, therefore with the sampling rate of 8kHz will result in 1000 periods of the signal every second, ie. a signal frequency of 1kHz.

1.3 Question 3

In the same while loop referenced in the previous question, it can be seen that the sample is passed to the output, first being multiplied by the gain and cast as an Int32. Therefore the output sample is a 32-bit integer.

2 Code Operation

The main principle of operation of the sinegen() function is to constantly track the index of the lookup table with a high precision. When a value has to be accessed, the index is rounded down for a valid index value. This output sample is then returned as the next sample.

```
index_round = (int) round(index);
table_sample = table[index_round];
```

This requires a value of index to be tracked over each call to the sinegen function, therefore it is declared as a global variable.

Index is increased every time the function is called, with the increment value depending on the values of the signal and sampling frequencies. If the value of index is equal or greater than 256; then it must be wrapped around to give a valid index to the lookup table. As the default modulo operator only operates on integers, the initial approach was to make use of the fmod function. However, it was found that the calculation performed by fmod was too slow when operating at the highest sampling frequency (96kHz). Therefore this was changed for a simple while loop:

```
increment=(float)SINE_TABLE_SIZE*sine_freq/(float)sampling_freq;
index=index+increment;
while(index >= SINE_TABLE_SIZE)
{
    index = index - SINE_TABLE_SIZE;
}
```

The requirement for tracking index as a float came about due to the behaviour of the code for low frequencies. Previously the function used an integer value for index, but this resulted in increment having a value of 0 for any desired frequencies smaller than $\frac{sampling\_freq}{SINE\_TABLE\_SIZE}$ (meaning that the only output value was constantly 0, the first indexed value in the look up table). The solution to this was to give index a higher precision, and to perform the rounding operation as late into the process as possible.

Increasing the resolution is only necessary when two adjacent samples have the same value (which will occur for when the sampling frequency is substantially higher than the signal frequency). A solution to this would be to perform interpolation between the sample and the next.

3 Scope Traces

These examples were chosen to demonstrate operation at high and low frequencies across a wide range of sampling frequencies. Due to the filter present on the output of the DSP board, the lower frequencies are attenuated. This filter has a corner frequency of 7Hz, shown in figure 1, therefore the attenuation is only observed on the 10Hz signal.

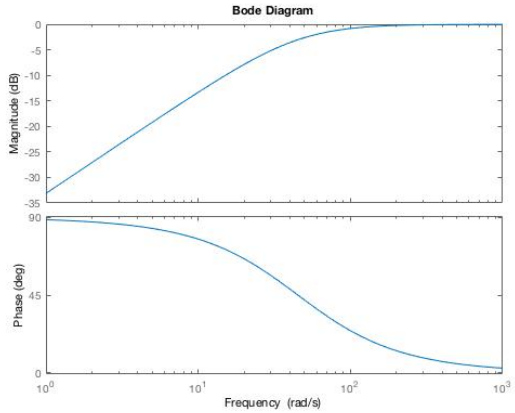

Figure 1: Bode Plot of Analogue Output Filter

3.1 8kHz Sampling Frequency

The trace in figure 2 shows a 10Hz signal, demonstrating the accurate reproduction of a low frequency sine wave.

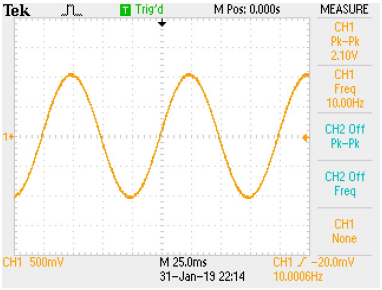

Figure 2: $f_s = 8kHz,\ f = 10Hz$

The 3.5kHz wave in figure 3 shows the increase in distortion as the signal frequency approaches the Nyquist limit.

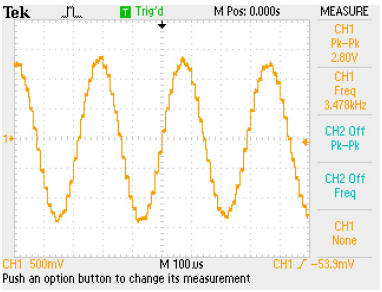

Figure 3: $f_s = 8kHz,\ f = 3.5kHz$

This trace in figure 4 shows a clean 2kHz sine wave, however this is actually a result of an input of 10kHz. As this is significantly above the Nyquist limit it is a good example of the effects of aliasing.

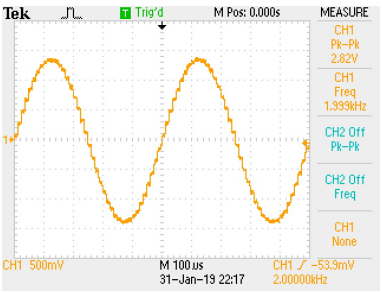

Figure 4: $f_s = 8kHz,\ f = 10kHz$

3.2 32 kHz

The trace in figure 5 displays the output signal for a 10kHz signal operating at a 32kHz sampling frequency.

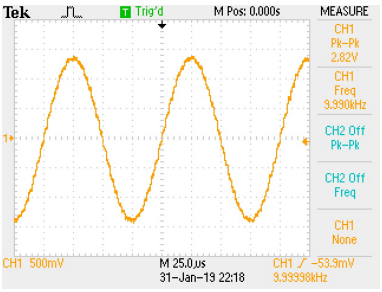

Figure 5: $f_s = 32kHz,\ f = 10kHz$

Figure 6 is the same again showing a waveform of a frequency close to the Nyquist limit, again showing the effects of aliasing.

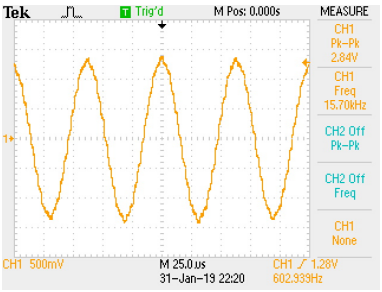

Figure 6: $f_s = 32kHz,\ f = 15.5kHz$

3.3 96kHz

In figure 7 is a 10Hz signal operating at a 96khz sampling rate, demonstrating that the sampling rate has little impact on the accuracy of the output signal, as long as the signal frequency is below the Nyquist limit. However, it should be noted that a particularly high sampling rate can cause issues for the output as the sine generation code may not be efficient enough to produce samples on time.

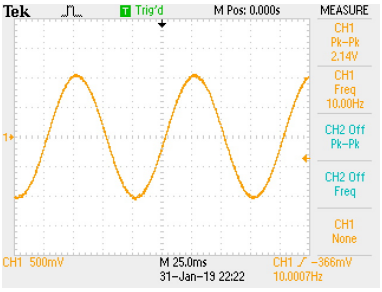

Figure 7: $f_s = 96kHz,\ f = 10Hz$

Finally, in figure 8 is a 44kHz signal, as this is close to the Nyquist limit, distortion can be seen taking place. This waveform shows the upper limit of output frequency that can be reached using this method.

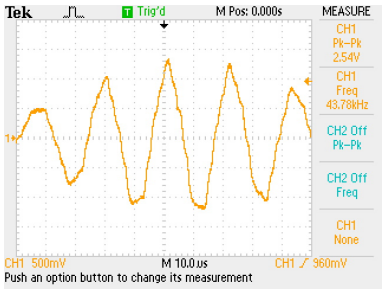

Figure 8: $f_s = 96kHz$, $f = 44kHz$

4 Limits

A practical limit to the output frequency for on all test cases is seen as the frequency approaches the Nyquist limit. In all cases this can be observed at a few kHz below the Nyquist limit. The low frequency operation is limited by the high pass filter on the output of the DSP. This filter has a corner frequency of 7Hz (figure 1), causing attenuation as the frequency falls below this value.

5 Appendix

```
/*************************************************************
         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
 *************************************************************/

/***************** 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

//Size of lookup table defined here
#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. */
/* 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;

/* 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 careful
    that you do not set it above the current nyquist frequency! */
float sine_freq = 1000.0;



/* Global variables added during the Lab */
//Array that containing the lookup table values
float table[SINE_TABLE_SIZE];
//Used for tracking the current index of the lookup table
float index = 0;
/****************** Function prototypes *********************/
void init_hardware(void);
float sinegen(void);
/*Function added during the Lab*/
void sine_init(void);

/********************** Main routine ************************/
void main()
{

    // initialize board and the audio port
    init_hardware();

    // initialize sine 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()
{
    /* 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)
{
    /*
    The function calculates the increment required for the index
    of the lookup table (fractionally) for a given sample rate
    and desired frequency. This index is then rounded down to
    ensure that it is valid for lookup. The corresponding look
    up value is then returned
    */

    float table_sample;
    float increment;
    int index_round;
```

```
    //the current index is rounded to a valid index
    index_round = (int) floor(index);
    //the required sample is recovered from the lookup table
    table_sample = table[index_round];

    //the increment value of the index is calculated
    increment = (float) SINE_TABLE_SIZE * sine_freq / (float) sampling_freq;

    //the index is incremented
    index = index + increment;

    //if index is equal or greater to 256 it must be wrapped around
    while(index >= SINE_TABLE_SIZE)
    {
        index = index - SINE_TABLE_SIZE;
    }

    return table_sample;
}


/* Function added during labs*/
void sine_init(){
    int i;

    /* The for loop runs 256 times, generating a value for each index in the defined
     table */
    for(i = 0; i < SINE_TABLE_SIZE; i++)
    {
        /* The argument passed to sin is formed from the current loop index */
        double sin_arg = (double)i / (double)SINE_TABLE_SIZE;

        /* The corresponding sin value is calculated */
        double sin_val = sin(2 * PI * sin_arg);

        /* The sin value is stored in table */
        table[i] = sin_val;
    }
}
```