DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
IMPERIAL COLLEGE LONDON

EE3-19
REAL-TIME DIGITAL SIGNAL PROCESSING
LAB 3 REPORT

YASMIN BABA - 01196634
ZHI WEN SI - 01243716

15th of February, 2019

TABLE OF CONTENTS

1. Exercise 1

1.1 Bold Questions

Why is the full rectified waveform centred around 0V and not always above 0V as you may have been expecting?

A high pass filter exists at the output of the audio chip with the purpose of attenuating low frequency signals. Due to the capacitors highlighted in *Figure 1*, no DC components will be able to pass to the output. Thus, when the full-wave rectified signal passes through the filter, its DC offset is removed, shifting it down to centre around 0V.

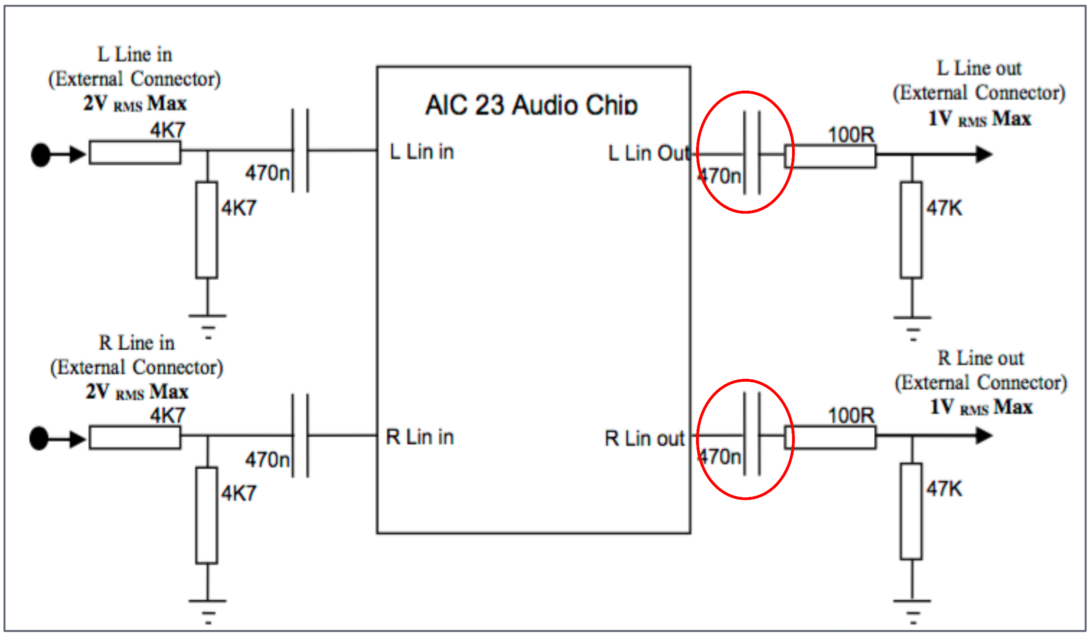


Figure 1: Audio Chip External Components

DC blocking is highly desired when outputting audio signals as DC offsets may cause the following problems [1]:

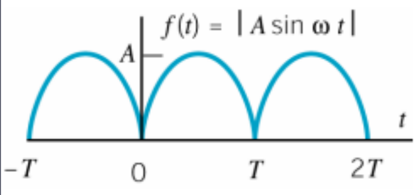
- Reduction in voltage headroom, potentially leading to clipping of the loudest components within the signal
- Intermittent crackling sounds known as ‘clicks’ may occur
- Low level distortion, which is usually inaudible but may cause issues if the wave is further sampled
- Amplification will increase the DC component, resulting in all the above concerns to worsen

Note that the output waveform will only be a full-wave rectified version of the input if the input from the signal generator is below a certain frequency. Why is this? You may wish to explain your answer using frequency spectra diagrams. What kind of output do you see when you put in a sine wave at around 3.8 kHz? Can you explain what is going on?

Signal frequencies between 10 Hz and 3.8 kHz were investigated. Many interesting discoveries were found within this range in which the output signal did not perfectly look or behave like a full-wave rectified version of the input. All worthy findings and edge cases will therefore be analysed.

1.1.1 THE NYQUIST FREQUENCY AND ALIASING

To understand the following results and discussions, it is useful to firstly recognise the behaviour of the full-wave rectified sine wave in the frequency domain. The waveform can be represented in terms of the following Fourier Series shown in *Figure 2.*



Full wave rectified sine wave: $\omega_0 = \dfrac{2\pi}{T}$

$$f(t) = \frac{2A}{\pi} - \frac{4A}{\pi} \sum_{n=1}^{\infty} \frac{\cos(n\omega_0 t)}{4n^2 - 1}$$

Figure 2: Fourier Series of a Rectified Sine Wave[1]

According to the equation, there are an infinite number of harmonics present in the rectified signal. These harmonics will have a decreasing amplitude and will be present at multiples of the rectified signal's frequency. Copies of this spectrum will be shifted by the DAC's sampling frequency of 8 kHz, resulting in overlapping and extra frequency components, potentially distorting the outputted signal.

In order for there to be no loss of information when sampling the signal, the sampling frequency must be at least twice the maximum frequency present in the signal. Since the sampling frequency of the audio chip is fixed at 8 kHz, any signals of frequency higher than 4 kHz (the Nyquist frequency) will be wrongly outputted due to aliasing. However, since the inputted sine wave is rectified and has double the original input frequency, the Nyquist frequency is now halved and is 2 kHz.

Using the MATLAB code in *Listing 4* (see *Appendix 3.1*), the frequency spectrum of the rectified signal can be presented and aliasing can be observed clearly.

Notes:

- The following MATLAB plots will have a DC component, however, due to the previously discussed filter at the output of the audio chip, this DC component will not exist in reality.
- The DAC has an anti-aliasing filter with a cut-off frequency of 4 kHz (Nyquist Frequency) in order to avoid aliasing, thus any frequency components above this value will be removed.
- The screenshots from the scope contain noise that the MATLAB plots do not consider.

Input signals of 1 kHz and 3 kHz will now be analysed to highlight the effect of aliasing due to sampling a rectified signal, the Nyquist frequency and the anti-aliasing filter of the DAC. Before inputted into the DAC, the input sine waves are rectified and therefore have frequencies 2 kHz and 6 kHz respectively, producing the following magnitude responses.

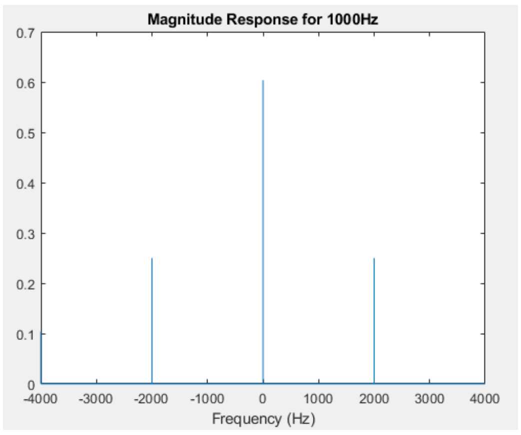


Figure 3: 1 kHz Signal Frequency

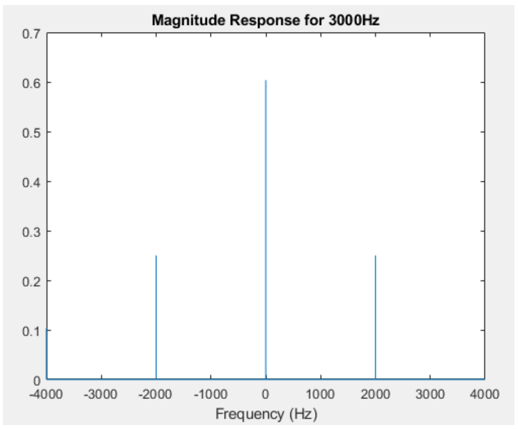


Figure 4: 3 kHz Signal Frequency

As shown in *Figures 3* and *4*, the frequency spectra are identical, making it impossible to correctly identify the input signal frequency based on the frequency spectra alone. At baseband, the rectified 6 kHz signal has an impulse at 6 kHz and -6 kHz. Due to the sampling frequency of 8 kHz, the spectrum is shifted and centred around 8 kHz, shifting the impulses to 14 kHz and 2 kHz respectively. Likewise, when centred around -8 kHz, the two impulses are shifted to -14 kHz and -2 kHz. Due to the DAC's anti-aliasing filter, only the impulses at 2 kHz and -2 kHz remain at the output. These impulses are then wrongly interpreted by the oscilloscope as the baseband spectrum of a 2 kHz sine wave.

An interesting finding is that the 1 kHz signal observed on the oscilloscope had the appearance of a regular sine wave, as illustrated in *Figure 5*. In order for the waveform to have the appearance of a rectified sine wave, all infinite harmonics must be present in the frequency spectra. However, due to the DAC's filter, all frequencies above 4 kHz are removed, resulting in the signals waveform to become more smooth. Moreover, as the filter is not perfect, the frequency component at 4 kHz will be attenuated, as illustrated in *Figure 6.* This thus leaves the 2 kHz component to be most prominent, resulting in an output wave that mimics a non-rectified 2 kHz sine wave.

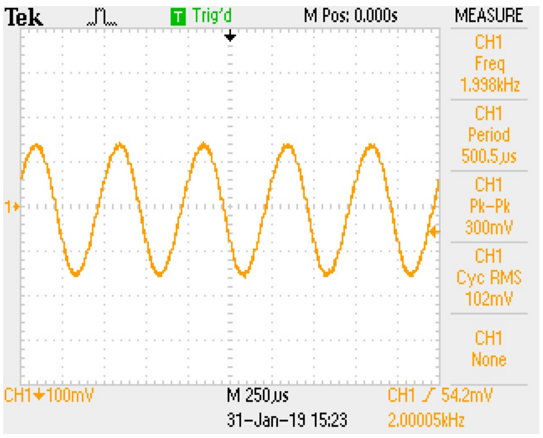


Figure 5: 1 kHz Input Signal

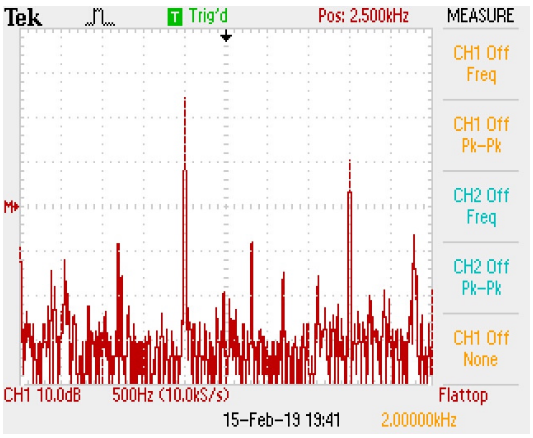


Figure 6: Frequency Spectrum

1.1.2 Amplitude Distortion

Input frequencies approaching the Nyquist frequency were observed to have fluctuating output waveforms. *Figure 7* illustrates the output of an input signal with a frequency of 1.9 kHz. It was clear that the output consisted of two signals:

- The rectified signal of 3.8 kHz
- An envelope signal of 400 Hz.

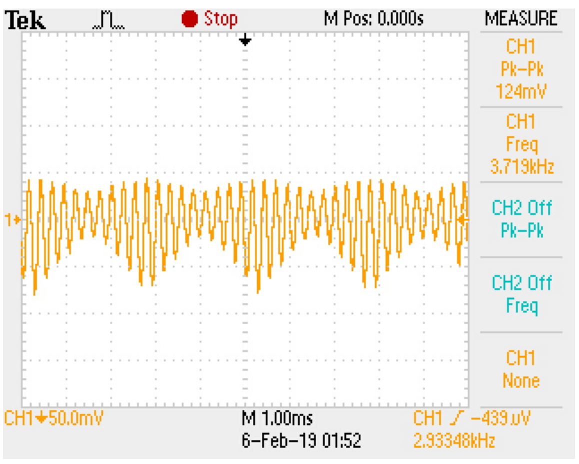


Figure 7: 1.9 kHz Input Frequency

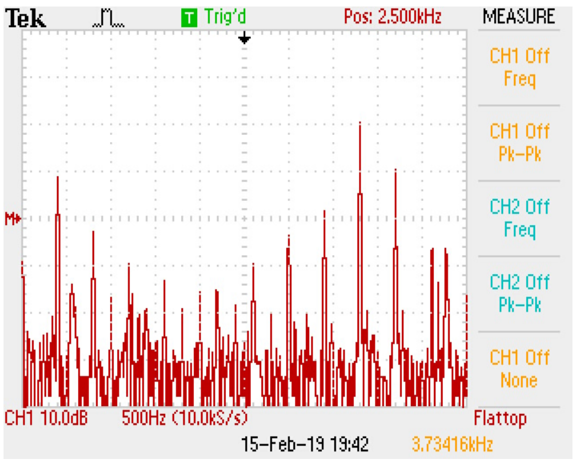


Figure 8: Frequency Spectrum

Again, this can be explained via the frequency spectrum of the output. As shown more clearly in *Figure 9,* a frequency component of 400 Hz exists. This is due to the baseband harmonic at -7.6 kHz, which has been shifted to 400 Hz when the signal was sampled.

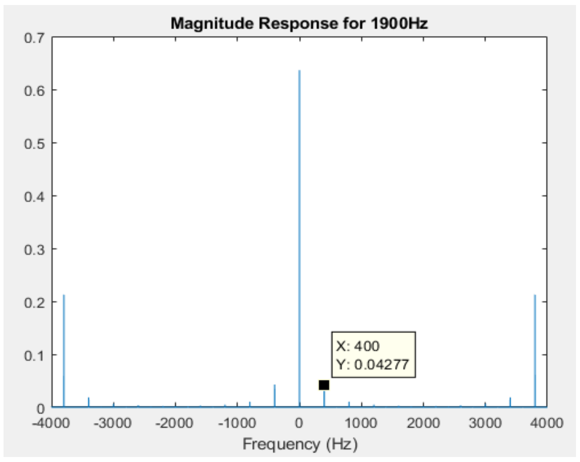


Figure 9: Frequency Spectrum of 1.9 kHz Signal Frequency

The amplitude of the 400 Hz component in addition to the distance from the 3.8 kHz component are significant enough for the oscilloscope to detect its presence, resulting in the misleading 'AM-looking' output signal. After further analysing the frequency spectra of other input signals, the same phenomenon occurs for signals of lower frequencies, however their aliased frequency components were not significant enough to be shown clearly on the oscilloscope.

1.1.3 A Signal Frequency of 3.8 kHz

Inputting a sine wave of frequency 3.8 kHz produces a full-rectified output waveform of frequency 400 Hz, as shown in *Figure 10*. This is due to aliasing as the signal lies above the Nyquist frequency.

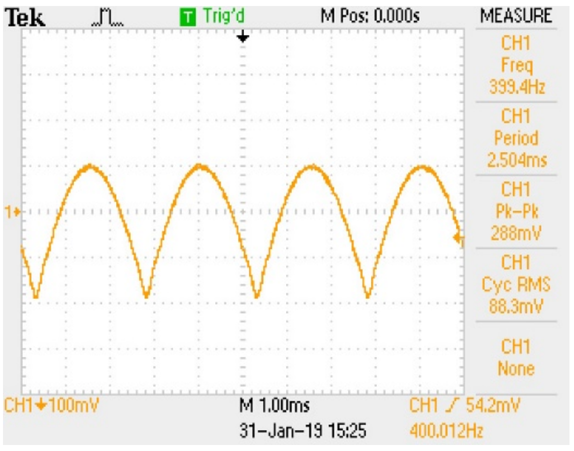


Figure 10: 3.8 kHz Signal Frequency

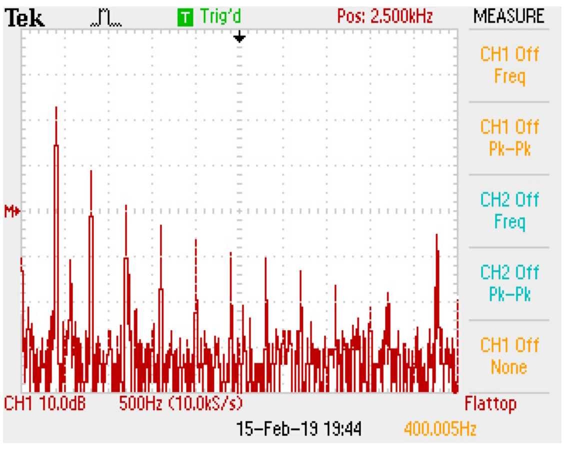


Figure 11: Frequency Spectrum

Firstly, the rectified signal will have a fundamental frequency of 7.6 kHz and infinitely many amplitude-decreasing harmonics as discussed in Section 1.1.1. The DAC then reads this rectified signal and samples it at a rate of 8 kHz producing positive and negative frequency components at 400 Hz, 15.6 kHz, etc. (in addition to multiples of the infinite harmonics of lesser amplitudes). The DAC's anti-aliasing filter then filters out any components above 4 kHz, thus leaving a prominent 400 Hz and the less prominent components due to the harmonics. This is illustrated both in *Figures 11* and *12*.

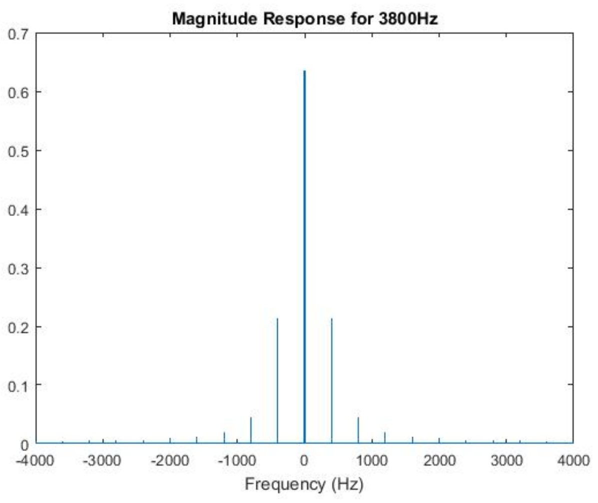


Figure 12: Frequency Spectrum of a 3.8 kHz Input Wave

In other words, due to aliasing, the input wave of 3.8 kHz is mimicking an input wave of 200 Hz. After further investigation of input signals above the Nyquist frequency, it is concluded that inputted waves of frequencies 2 kHz to 3.8 kHz will ideally output waves mimicking those of 2 kHz to 400 kHz waves respectively.

1.1.4 Low Input Frequencies

As expected, when a signal of frequency 10 Hz was inputted, the outputted signal was a full-rectified wave with frequency 20 Hz, as shown in *Figure 13.* However, the waveform was noticeably skewed. This is due to the high pass filter at the output of the audio chip which has a non-linear phase response. The group delay of the signal is frequency dependent resulting in each harmonic to experience different delay times. As the sum of these harmonics give the waveform its shape, the mismatched components will no longer be able to produce the correct waveform.

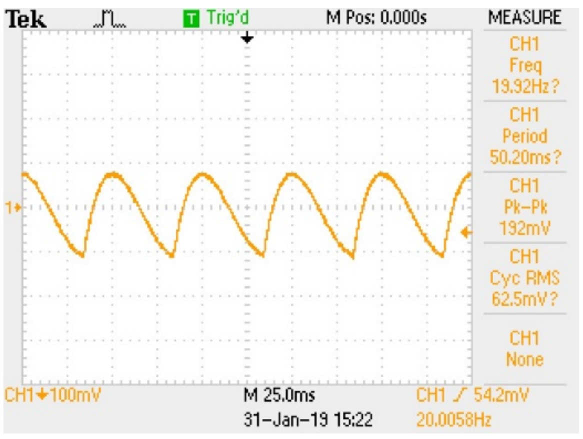


Figure 13:10Hz Signal Frequency

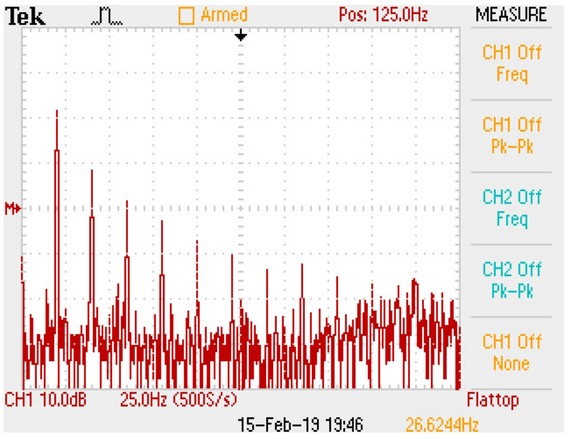


Figure 14: Frequency Spectrum

Furthermore, the wave was found to be attenuated compared to signals of higher frequencies. This is also due to the high pass filter. The corner frequency of the filter is,

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi(100 + 47\times10^3)(270\times10^{-9})} = 7.19\ Hz$$

Ideally, this high pass filter would cut off all frequencies exactly below 7.19 Hz, producing a magnitude response that directly falls to minus infinite gain at the corner frequency. However, in reality, such an infinite shoot cannot be implemented and the gain will actually roll off such that attenuation will begin to occur before the corner frequency (see *Figure 15*).

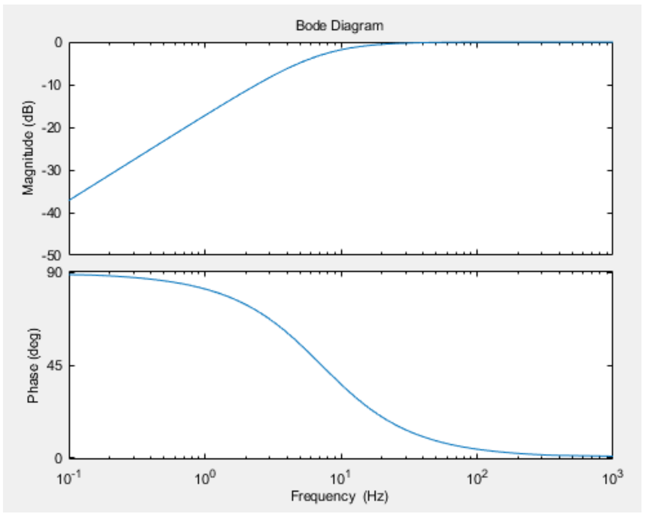
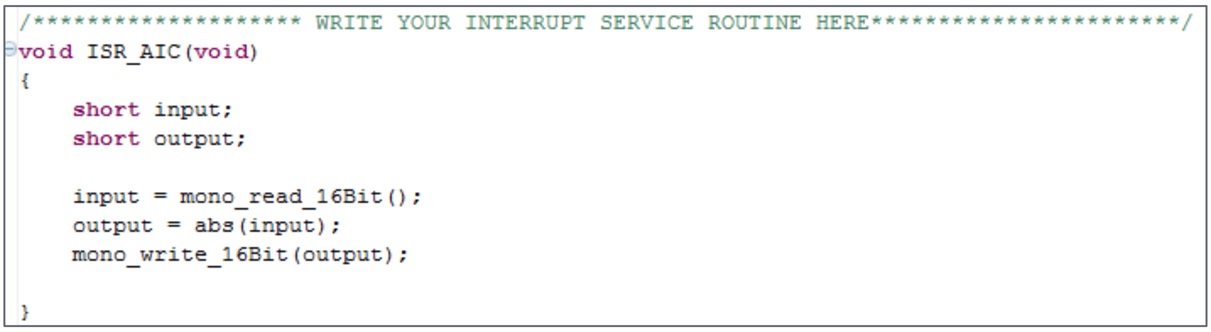


Figure 15: Frequency Response of the High Pass Filter

1.2 Code Explanation

The following code reads a sample from the codec, using the function **mono_read_16Bit** and stores it into the variable **input.** This function reads a Left and Right sample from the audio port, divides their amplitudes by two and sums the samples together to provide an average mono input of size 16-bits. This ‘input signal’ is then fully rectified using the **abs** function, converting all negative samples to positive ones, hence doubling the signal frequency. Finally, the fully rectified signal is written to the DAC using the **mono_write_16Bit** function. The output must not exceed 16 bits thus the variable types are set as **short**.



```
/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
void ISR_AIC(void)
{
    short input;
    short output;

    input = mono_read_16Bit();
    output = abs(input);
    mono_write_16Bit(output);

}
```

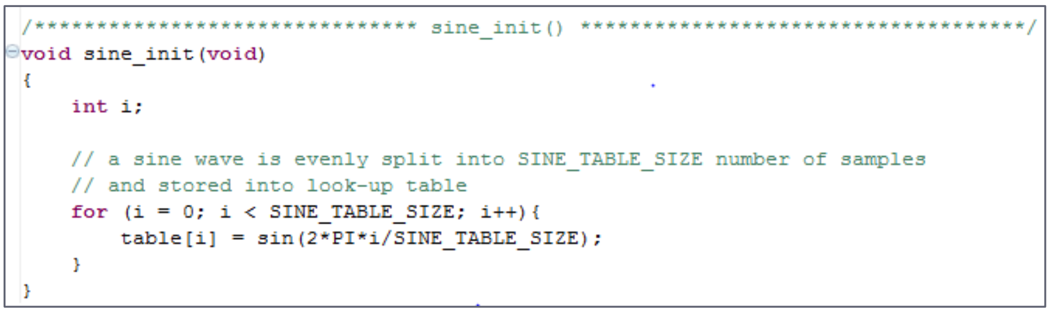
Listing 1: ***ISR_AIC*** *Function*

2. Exercise 2

2.1 Code Explanation

The **ISR_AIC** function now generates a sine wave via the look up table technique explored in Lab 2. The look-up table stores evenly spaced data points, or samples, of one cycle of a sine wave which will be sent to the output. In this case, the look-up table is of size 256.

The following function **sine_init** initialises the look-up table by using a for loop, assigning each cell of the table with a different data point.

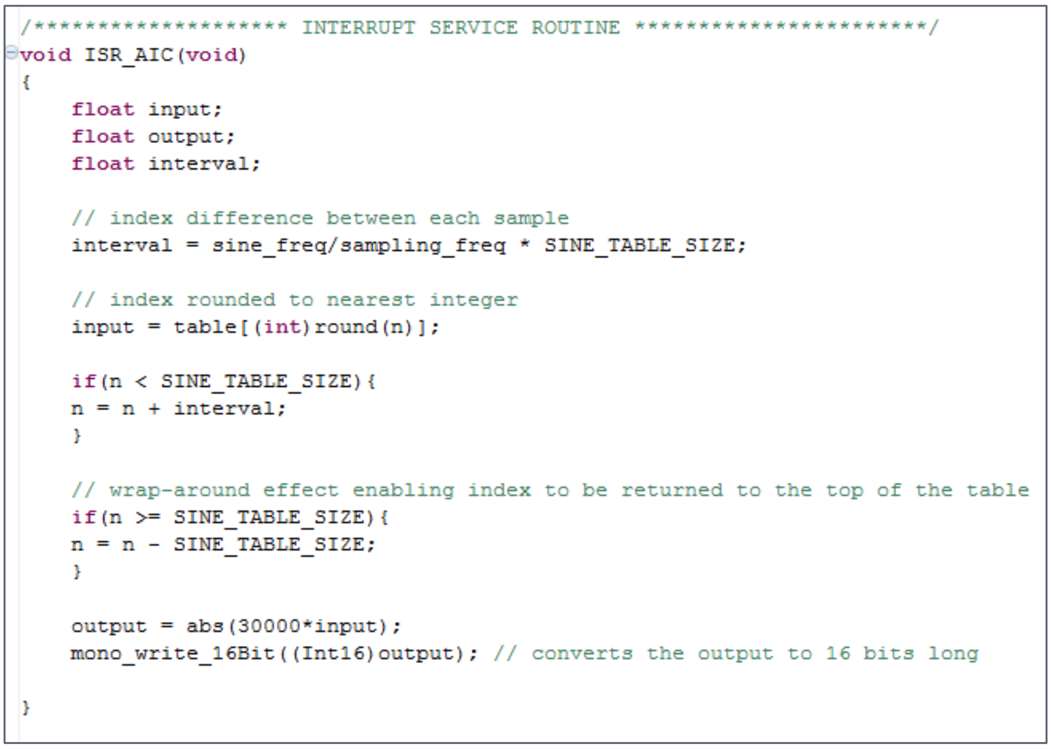


```c
/******************************** sine_init() ***********************************/
void sine_init(void)
{
    int i;

    // a sine wave is evenly split into SINE_TABLE_SIZE number of samples
    // and stored into look-up table
    for (i = 0; i < SINE_TABLE_SIZE; i++){
        table[i] = sin(2*PI*i/SINE_TABLE_SIZE);
    }
}
```

Listing 2: **sine_init** *Function*

In order to retrieve the samples and send them to the output, the **ISR_AIC** function has been modified, as shown in *Listing 3*.



```c
/******************** INTERRUPT SERVICE ROUTINE ***********************/
void ISR_AIC(void)
{
    float input;
    float output;
    float interval;

    // index difference between each sample
    interval = sine_freq/sampling_freq * SINE_TABLE_SIZE;

    // index rounded to nearest integer
    input = table[(int)round(n)];

    if(n < SINE_TABLE_SIZE){
    n = n + interval;
    }

    // wrap-around effect enabling index to be returned to the top of the table
    if(n >= SINE_TABLE_SIZE){
    n = n - SINE_TABLE_SIZE;
    }

    output = abs(30000*input);
    mono_write_16Bit((Int16)output); // converts the output to 16 bits long

}
```

Listing 3: **ISR_AIC** *Function*

The float variable ***interval*** is used as the index difference between the current sample and the next. The float ***n***, a global variable initially set to 0, represents the current index value itself. Finally, the float ***input*** stores the sample value taken from the look-up table at the current index and is the variable sent back in to main to be outputted.

The variable ***n*** must be declared as a float in order to avoid calculation errors when later added with ***interval***. However, when passed as an index into ***table***, it is rounded to the nearest integer as the table is indexed by whole numbers.

Since a continuous sine wave is to be generated, and the fact that the look-up table consists of a finite 256 samples, a wrap-around effect is implemented via the if statements. If the index is below 256, the next index ***n*** is incremented by the value ***interval***. On the other hand, if the index is equal to or greater than 256, it will be subtracted by 256. It is not set back to zero in case the first index does not start at zero – using this method, a phase difference may be implemented if desired.

Finally, during rectification via the **abs** function, the signal was amplified by a gain of 30,000 in order to achieve a readable measurement on the oscilloscope. Some output observations are shown below for completeness.

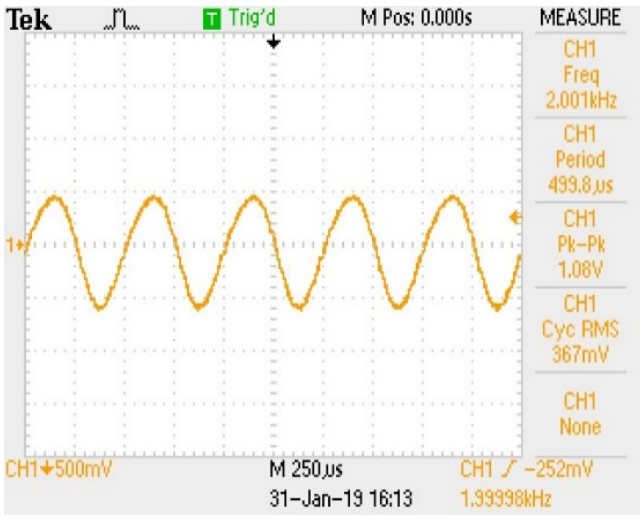
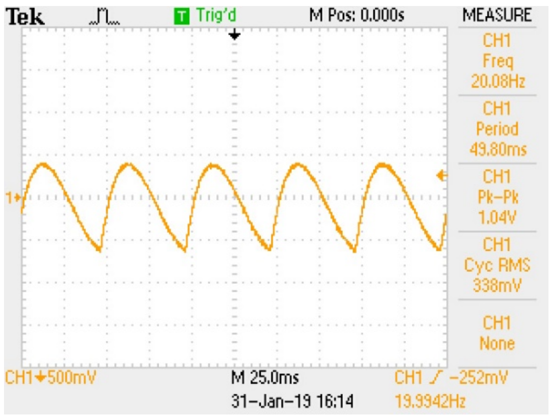


Figure 16: Input Frequency of 1 kHz



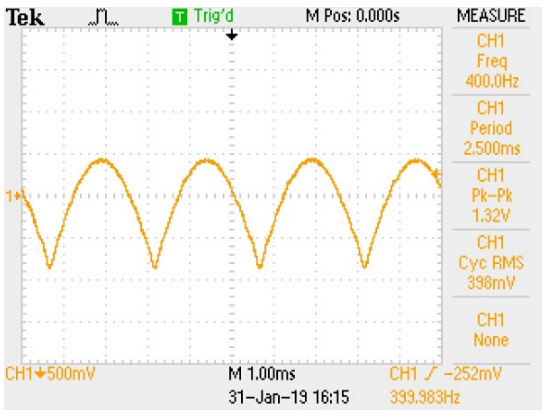
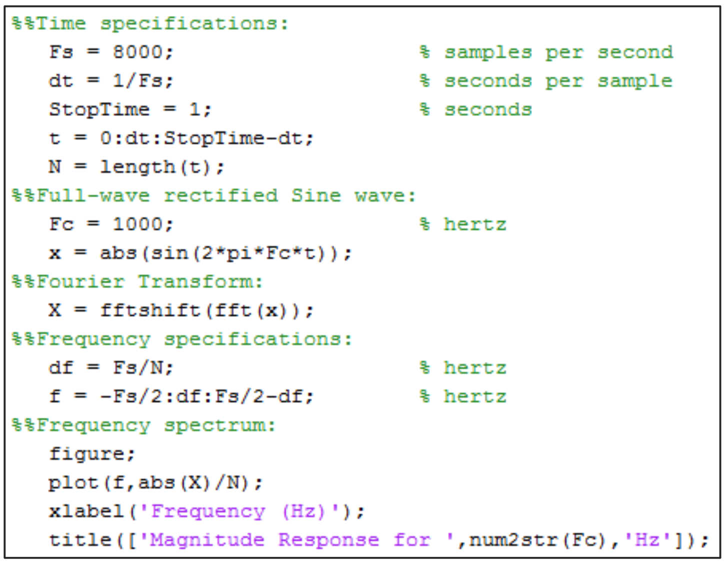


Figure 17: Input Frequency of 10 Hz

Figure 18: Input Frequency of 3.8 kHz

3. APPENDIX

3.1 MATLAB CODE



```
%%Time specifications:
   Fs = 8000;                      % samples per second
   dt = 1/Fs;                      % seconds per sample
   StopTime = 1;                   % seconds
   t = 0:dt:StopTime-dt;
   N = length(t);
%%Full-wave rectified Sine wave:
   Fc = 1000;                      % hertz
   x = abs(sin(2*pi*Fc*t));
%%Fourier Transform:
   X = fftshift(fft(x));
%%Frequency specifications:
   df = Fs/N;                      % hertz
   f = -Fs/2:df:Fs/2-df;           % hertz
%%Frequency spectrum:
   figure;
   plot(f,abs(X)/N);
   xlabel('Frequency (Hz)');
   title(['Magnitude Response for ',num2str(Fc),'Hz']);
```

Listing 4: MATLAB Code to Illustrate Frequency Spectra

3.2 Exercise 1 Full Code Listing

```
/*************************************************************************************
                         DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                                   IMPERIAL COLLEGE LONDON

                         EE 3.19: Real Time Digital Signal Processing
                              Dr Paul Mitcheson and Daniel Harvey

                                     LAB 3: Interrupt I/O

                               ********* I N T I O. C **********

  Demonstrates inputing and outputing data from the DSK's audio port using interrupts.

 ************************************************************************************
                              Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
                              Updated for CCS V4 Sept 10
 ************************************************************************************/
/*
 *      You should modify the code so that interrupts are used to service the
 *  audio port.
 */
/**************************** Pre-processor statements ******************************/

#include <stdlib.h>
//  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 writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

/******************************* 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       */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 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;

 /****************************** Function prototypes *******************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);

/********************************** Main routine ************************************/
```

```
void main(){


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

  /* initialize hardware interrupts */
  init_HWI();

  /* loop indefinitely, waiting for interrupts */
  while(1)
  {};

}

/********************************** init_hardware() **********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

       /* Function below sets the number of bits in word used by MSBSP (serial port) for
       receives from AIC23 (audio port). We are using a 32 bit packet containing two
       16 bit numbers hence 32BIT is set for  receive */
       MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

       /* Configures interrupt to activate on each consecutive available 32 bits
       from Audio port hence an interrupt is generated for each L & R sample pair */
       MCBSP_FSETS(SPCR1, RINTM, FRM);

       /* These commands do the same thing as above but applied to data transfers to
       the audio port */
       MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
       MCBSP_FSETS(SPCR1, XINTM, FRM);


}

/********************************** init_HWI() **************************************/
void init_HWI(void)
{
       IRQ_globalDisable();                  // Globally disables interrupts
       IRQ_nmiEnable();                      // Enables the NMI interrupt (used by the
debugger)
       IRQ_map(IRQ_EVT_RINT1,4);             // Maps an event to a physical interrupt
       IRQ_enable(IRQ_EVT_RINT1);            // Enables the event
       IRQ_globalEnable();                   // Globally enables interrupts

}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
void ISR_AIC(void)
{
       short input;
       short output;

       input = mono_read_16Bit();
       output = abs(input);
       mono_write_16Bit(output);

}
```

Listing 5: Full Code Listing for Exercise 1

3.2 Exercise 2 Full Code Listing

```
/*************************************************************************************
                     DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                                   IMPERIAL COLLEGE LONDON

                          EE 3.19: Real Time Digital Signal Processing
                              Dr Paul Mitcheson and Daniel Harvey

                                     LAB 3: Interrupt I/O

                               ********* I N T I O. C **********

  Demonstrates inputing and outputting data from the DSK's audio port using interrupts.

 ************************************************************************************
                  Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
                                  Updated for CCS V4 Sept 10
 ************************************************************************************/
/*
 *      You should modify the code so that interrupts are used to service the
 *  audio port.
 */
/***************************** Pre-processor statements ******************************/

#include <stdlib.h>
//  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 writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

// PI defined here for use in your code
#define PI 3.141592653589793

// size of look-up table for values of a sine wave
#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      */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 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;

/* 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;

// create an array to store samples of sine wave
float table [SINE_TABLE_SIZE];

// look-up table index
float n = 0;

 /******************************** Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
void sine_init(void);
void ISR_AIC(void);
/*********************************** Main routine ************************************/
void main(){


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

  /* initialize hardware interrupts */
  init_HWI();

  // initialize look up table
       sine_init();

  /* loop indefinitely, waiting for interrupts */
  while(1)
  {};

}

/*********************************** init_hardware() *********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

       /* Function below sets the number of bits in word used by MSBSP (serial port) for
       receives from AIC23 (audio port). We are using a 32 bit packet containing two
       16 bit numbers hence 32BIT is set for  receive */
       MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

       /* Configures interrupt to activate on each consecutive available 32 bits
       from Audio port hence an interrupt is generated for each L & R sample pair */
       MCBSP_FSETS(SPCR1, RINTM, FRM);

       /* These commands do the same thing as above but applied to data transfers to
       the audio port */
       MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
       MCBSP_FSETS(SPCR1, XINTM, FRM);


}

/*********************************** init_HWI() **************************************/
void init_HWI(void)
{
       IRQ_globalDisable();                 // Globally disables interrupts
       IRQ_nmiEnable();                     // Enables the NMI interrupt (used by the
debugger)
       IRQ_map(IRQ_EVT_XINT1,4);            // Maps an event to a physical interrupt
       IRQ_enable(IRQ_EVT_XINT1);           // Enables the event
       IRQ_globalEnable();                  // Globally enables interrupts

}
/******************************** sine_init() ************************************/
void sine_init(void)
{
       int i;

       // sine wave evenly split into SINE_TABLE_SIZE number of samples and stored into look-
up table
       for (i = 0; i < SINE_TABLE_SIZE; i++){
               table[i] = sin(2*PI*i/SINE_TABLE_SIZE);
       }
}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
void ISR_AIC(void)
{
       float input;
       float output;
       float interval;

       // index difference between each sample
       interval = sine_freq/sampling_freq * SINE_TABLE_SIZE;

       // index rounded to nearest integer
       input = table[(int)round(n)];

       if(n < SINE_TABLE_SIZE){
    n = n + interval;
       }

       // wrap-around effect enabling index to be returned to the top of the table
       if(n >= SINE_TABLE_SIZE){
       n = n - SINE_TABLE_SIZE;
       }

       output = abs(30000*input);
       // full-wave rectification

       mono_write_16Bit((Int16)output); // converts the output to 16 bits long

}
```

Listing 6: Full Code Listing for Exercise 2

4. References

[1] https://manual.audacityteam.org/man/dc_offset.html

[2] https://web.calpoly.edu/~fowen/me318/FourierSeriesTable.pdf