RTDSP Lab 3 Report

Lizhang Lin, CID: 00840705
Yihan Qi, CID: 00813873

In this lab session, we used interrupt driven code to sample a sinewave input at 8kHz and outputted it again after full-wave rectification. We explained question 1 and 2 in section one, and provided relevant scope traces and a summary table. Detailed code explanations for exercise 1 & 2 can be found in section 2. Our source code are also attached in the last section of this report.

Table of Contents

1. Questions

Note: through out this report, channel 1 (yellow trace) and channel 2 (blue trace) display the input and output to the DSP kit respectively.

1.1. Question 1

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

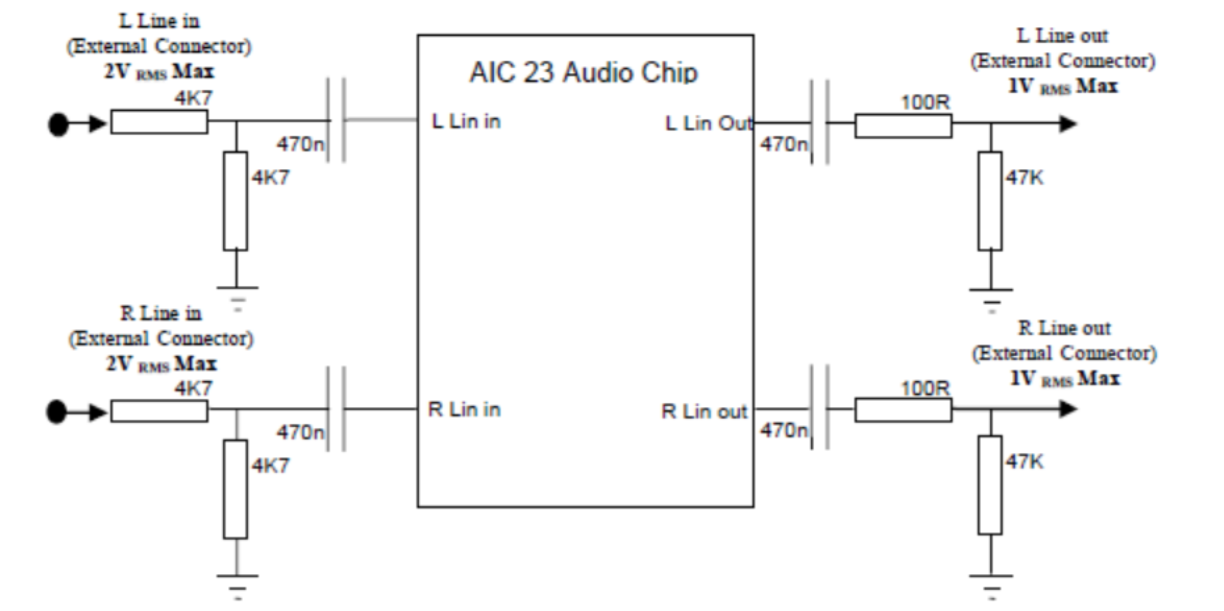


Figure 1: AIC23 Audio chip external components (adapted from TMS320C6713 Technical ref (page A-14, 2003 revision A))

From figure 2 below, we noticed that the output has a phase lag to input signal. It's understandable since it takes time for the DSP kit to process the input signal. Another important feature we discover is that the output is centered around 0V and not always above 0V. This is because the DC bias of the output from the audio chip is removed by a capacitor (shown in figure 1). Therefore, the output (blue trace) displayed in figure 2 is the outcome of AC coupling.

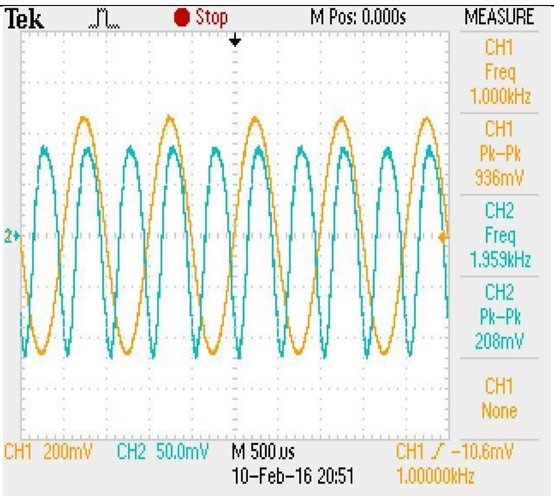


Figure 2: ch1: input signal generated by software of frequency 1000 Hz; ch2: full wave rectified

1.2. Question Two

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 3.8kHz? Can you explain what is going on?

Question 2 is answered in the following subsections of section 1.2.

1.2.1. Fourier Series and Fourier Transform of Full Wave Rectified Sinewave

The input signal is a sinewave. It is firstly sampled at 8000 Hz, then rectified and reconstructed into analogue signal at the output. Therefore, the signal becomes |sinewave| after rectification, which is equivalent to $sinewave\ *\ square\ wave$ if the frequency of the square wave is the same as that of the input sinewave.

Consider the fourier series of the square wave with the same frequency as the input sinewave, f**in**.

$$f(x) = \frac{4}{\pi} \sum_{n=1,3,5,\ldots}^{\infty} \frac{1}{n} \sin\left(\frac{n\pi x}{L}\right).$$

[1] , where $L = 1/\,(2\ *\ f\,in)$

Now let's consider the fourier series of full wave rectified sinewave which is the product of sinewave and square wave:

$$f(t) = \frac{2A}{\pi} - \frac{4A}{\pi}\sum_{n=1}^{\infty}\frac{\cos(n\omega_0 t)}{4n^2-1}$$ [2], where $\omega 0$ is the frequency of the full wave rectified sine wave, $\omega 0 = 2\pi * 2fin = 4\pi\, fin$, and A represents the amplitutde.

Therefore, the Fourier Transform of full wave rectified sine wave is formed by many impulses (harmonics) with fast decaying amplitude at higher frequency. The FT function is plotted below in frequency domain (figure 3).

$$\mathcal{F}\left(|\sin(\omega_o t)|_{decoupled}\right) \propto \sum_{n=1,2\ldots}^{\infty} \frac{\delta\left(2\omega_o - \frac{\omega}{n}\right) + \delta\left(\frac{\omega}{n} - 2\omega_o\right)}{4n^2 - 1}$$

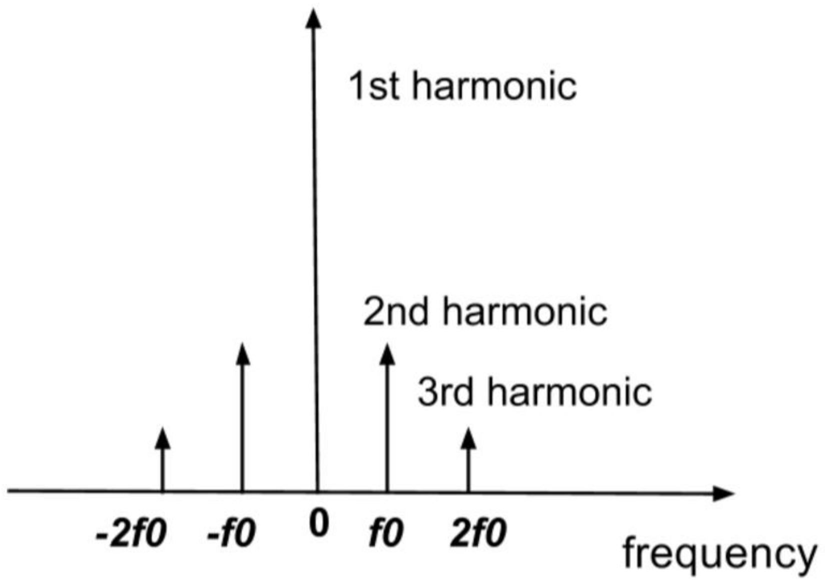


Figure 3: simplified plot of the FT of the full wave rectified sinewave (From the equation we obtained for the FT of full wave rectified sinewave, there should be infinite number of harmonics, however, we only display the first three harmonics in the following plots since the rest of the harmonics are significantly attenuated, which can be ignored).

1.2.2. Investigation into different frequency ranges

Having obtained the fourier series and FT as well as a simplified plot of full wave rectified sinewave, we then investigated and analysed the behaviour of the output signal in this lab session, which is not always a full wave rectified sinewave. Our following activities were performed at different frequencies of sinewave input, and were analysed in both time domain and frequency domain, theoretically and experimentally.

Always note that the output frequency is twice of the input signal frequency since the output is the full-wave rectified version of the input. The sampling frequency is always 8000 Hz in the following sections.

1.2.2.1 Input frequency above 4000Hz

The input signal must pass an anti-aliasing filter whose cut-off frequency is 4000 Hz since the sample frequency is 8000 Hz. Therefore, theoretically any input signal with frequency above 4000 Hz would be significantly attenuated and result in only noise at the output. And what we observed matches the above analysis (figure 6).

However, the anti-aliasing filter is not perfect and has a transition band (the frequency band between pass band and stop band). That's the reason why we obtained an output signal when the input signal has a frequency 4200 Hz, slightly above 4000 Hz (figure 4).

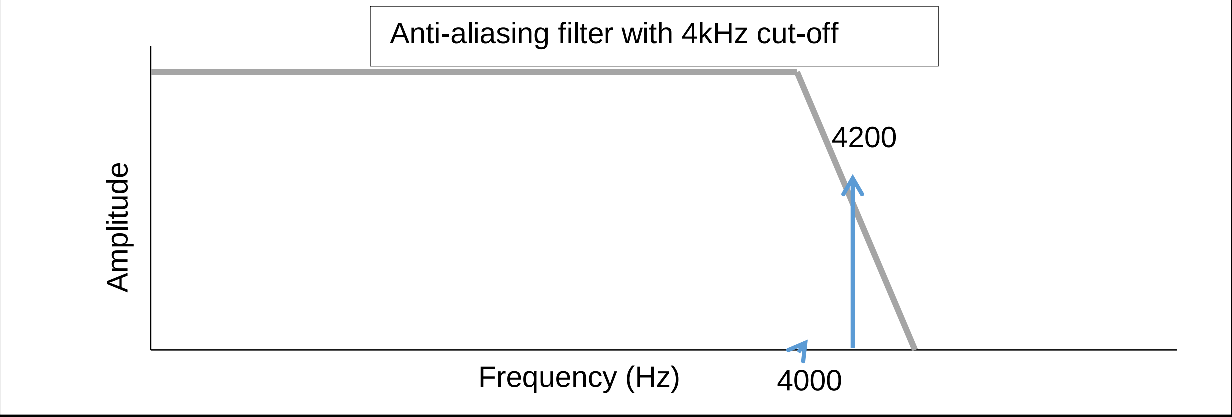


Figure 4: anti-aliasing filter at the input

Moreover, from the scope trace below (figure 5) we found that the output signal is not a full wave rectified version of the input. It's very interesting that the output signal has a frequency of 400 Hz when the input frequency is 4200 Hz. We supposed that this effect could be caused by aliasing, which would be further discussed in the following section in this report. And we concluded in this section that the output signal should either be noise or aliased version of the full wave rectified version of the input for any input frequency above Nyquist frequency. Namely, output signal is not a full wave rectified version of the input for any input frequency above 4000 Hz.

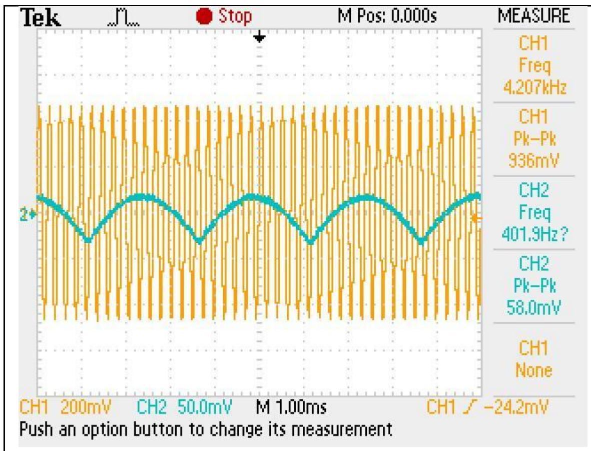


Figure 5: Scope trace of input and output signals with frequencies 4200 and 400 Hz respectively.

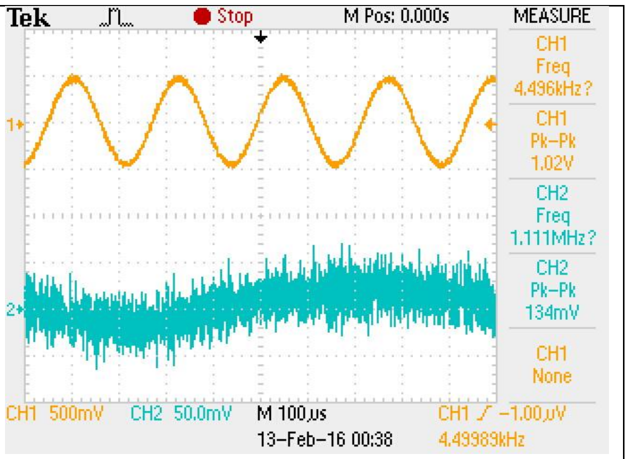


Figure 6: Scope trace of input and output signals. Input signal frequency is 4500 Hz and the output signal is only noise.

1.2.2.2 Input Frequency, *f* < 2000 Hz

In this section, we looked into the output signal behaviour when the frequency of input signal is below 2000 Hz. The reason for choosing 2000 Hz as the upper bound frequency for the input signal in this section is that the frequency of full wave rectified sinewave (double of the input signal frequency) is always below the Nyquist frequency, 4000 Hz.

We observed that when the input signal frequency is in the 10 – 500 Hz band (figure 9, 10, 11 & 12) and at particular values, such as 1000 Hz (figure 14) and 2000 Hz (figure 15), the output signal is a perfect full wave rectified version of the input without any amplitude modulation. However, for the rest of frequencies inside 10 to 2000 Hz frequency band, the output becomes an amplitude modulated and full wave rectified version of the input (figure 18). Detailed explanations are provided in the following paragraph.

Firstly, we attempted to find the reason why low frequency input signal (10 to 500 Hz) produces a perfect full wave rectified version of itself at the output. With the help of figure 7 and 8, we discovered that low frequency input gives a low frequency full rectified sinewave, therefore its harmonics centred at 0 Hz in the frequency domain doesn't affect the other harmonics sets around the integer multiple of the sampling frequency (ignore 4th, 5th, etc. harmonics). Thus it is perfectly reconstructed.

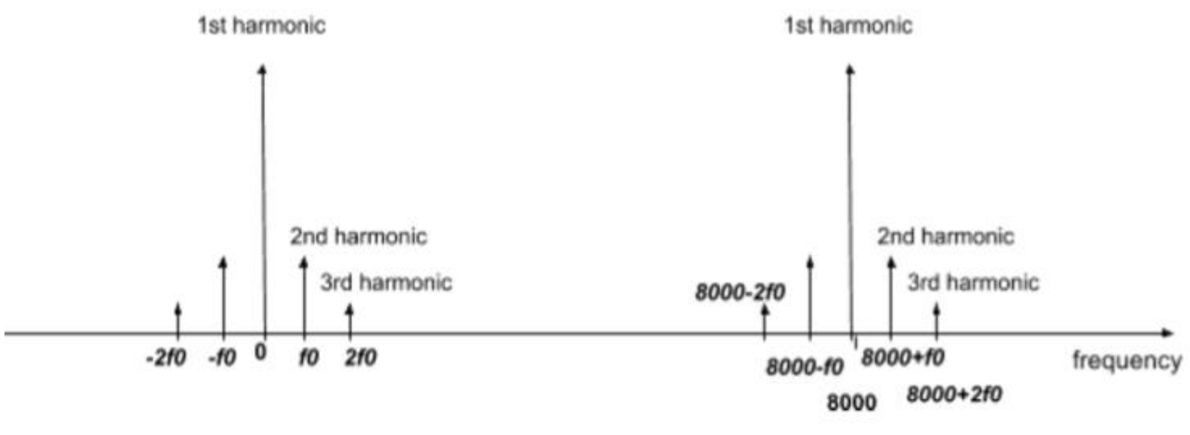


Figure 7: the plot of full wave rectified sine wave in frequency domain after sampling at 8000 Hz but before reconstruction (f0 is very small).

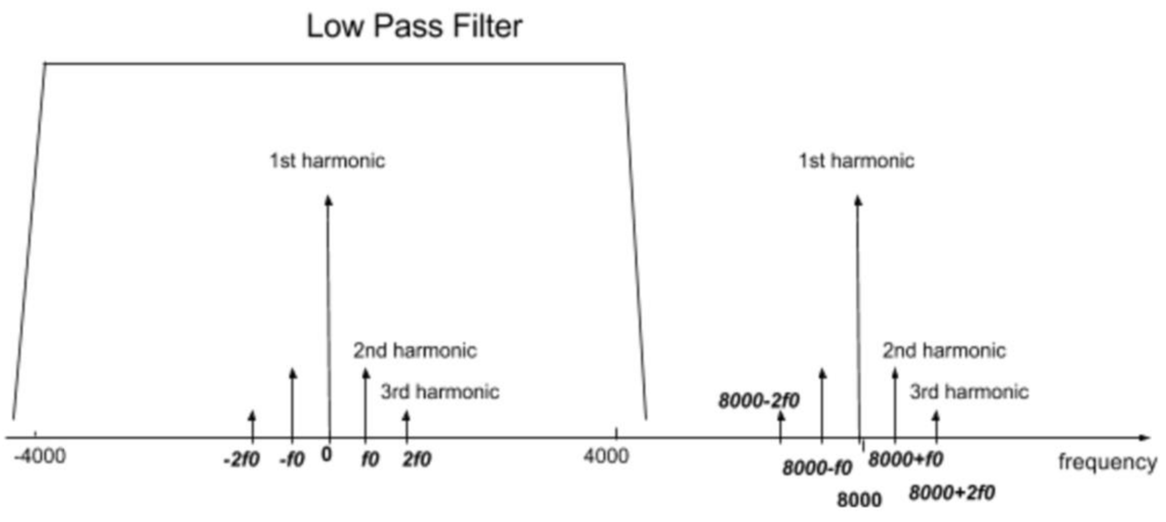


Figure 8: the plot of output signal in frequency domain after reconstruction (f0 is very small). The reconstructed signal appears at the output is the same as full wave rectified sinewave because its frequency at low input signal frequency.

A number of scope traces (figure 9, 10, 11 & 12) with input signal frequency from 0 to 500 Hz are provided below to justify our analysis.

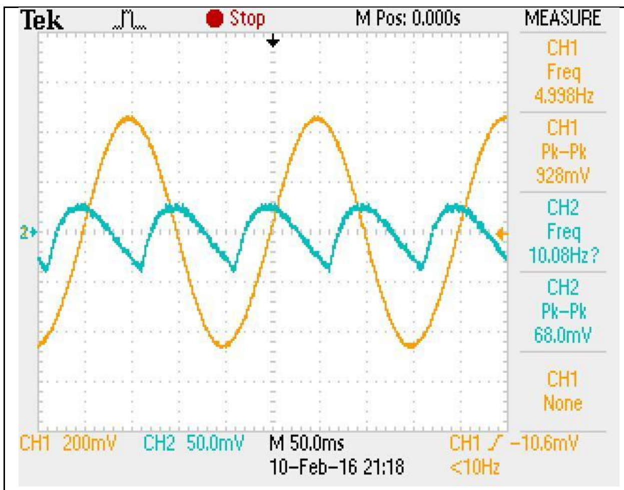


Figure 9: Scope trace of input and output signals with frequencies 5 and 10 Hz respectively.

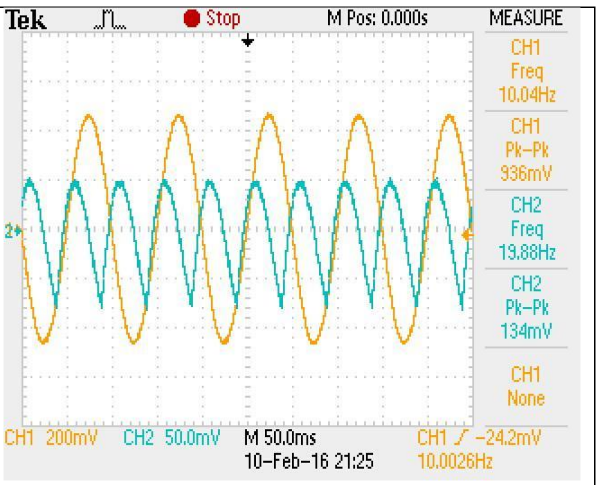


Figure 10: Scope trace of input and output signals with frequencies 10 and 20 Hz respectively.

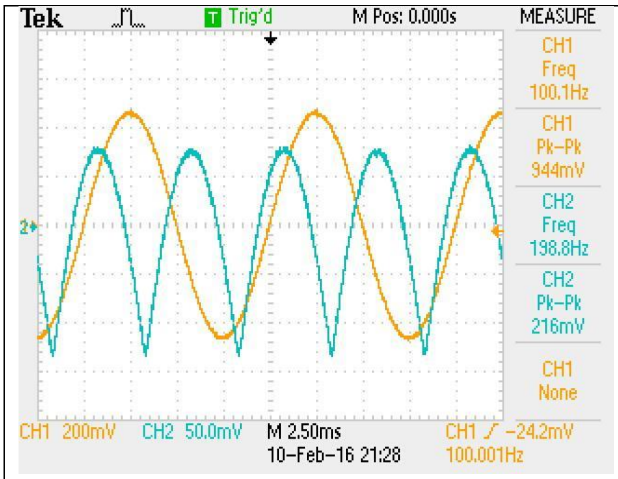


Figure 11: Scope trace of input and output signals with frequencies 100 and 200 Hz respectively.

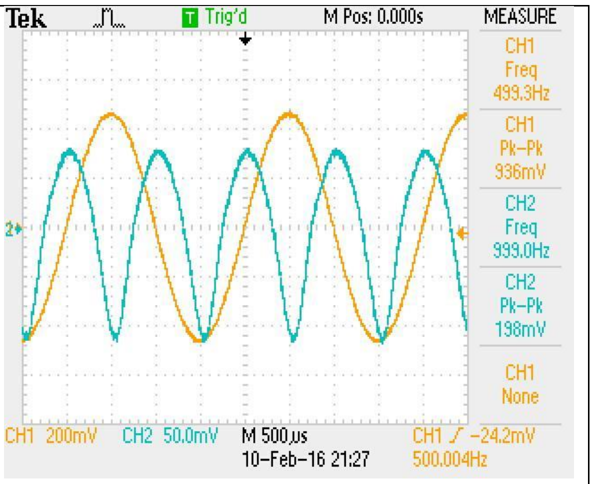


Figure 12: Scope trace of input and output signals with frequencies 500 and 1000 Hz respectively.

Secondly, we investigated into a special case when the input signal frequency is large, 1000 Hz (figure 14), but the output signal is still a perfect full wave rectified version of the input. When the input signal frequency is 1000 Hz, shown in the scope traces below (figure 14), the output signal is a full wave rectified sinewave of frequency 2000 Hz. We looked into the FFT of full wave rectified sinewave (figure 13), and found that when the input signal frequency is 1000 Hz, the FFT of the output has the same set of harmonics separated at integer multiples of the full wave rectified sinewave frequency though slightly changed in amplitude, which doesn't affect the output waveform in a significant manner. Therefore, the output we obtained at 1000 Hz input signal frequency is a perfect full wave rectified version of the input.

Based on what we just found, we concluded that at any input signal frequency of integer factors of 4000 Hz, such as 800, 1000, 1333, and 2000 Hz, the output signal is always a perfect full wave rectified version of the input signal. This is because at 800, 1000, 1333 and 2000 Hz, the duplicates of harmonics sets overlaps such that no additional harmonic at different frequency is produced.

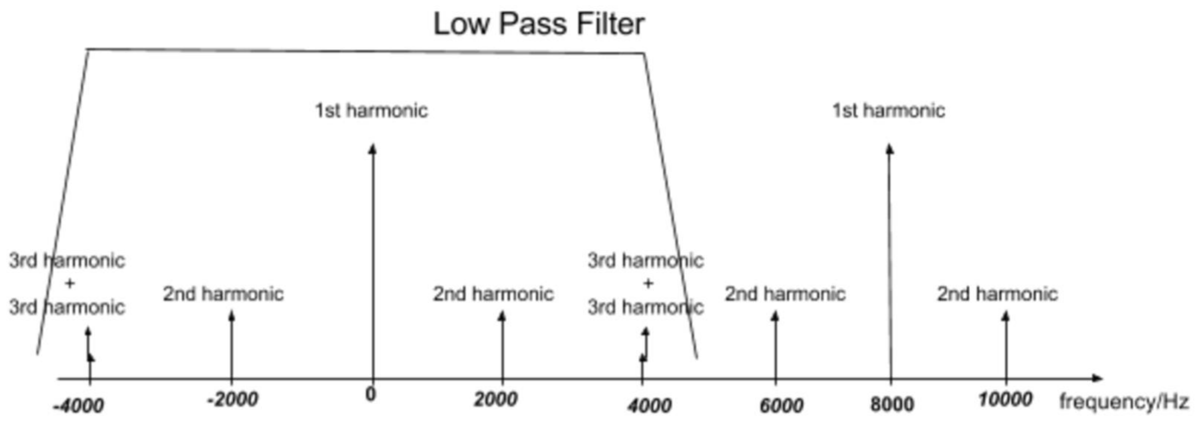


Figure 13: FFT of the output signal after reconstruction (fin=1000 Hz, f0=2000 Hz).

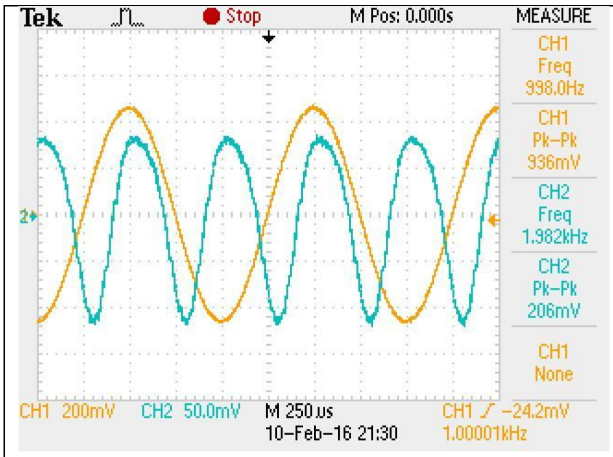


Figure 14: Scope trace of input and output signals with frequencies 1000 and 2000 Hz respectively.

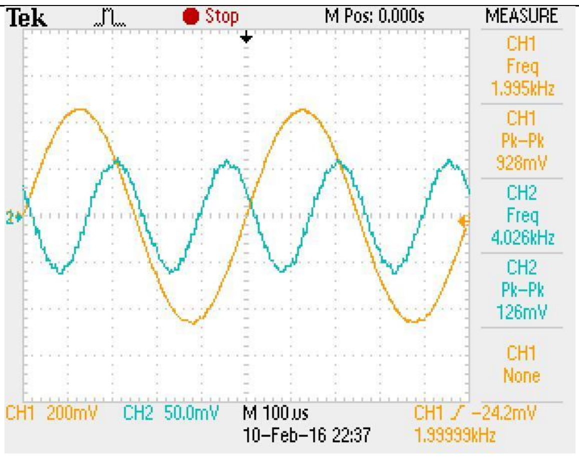


Figure 15: Scope trace of input and output signals with frequencies 2000 and 4000 Hz respectively.

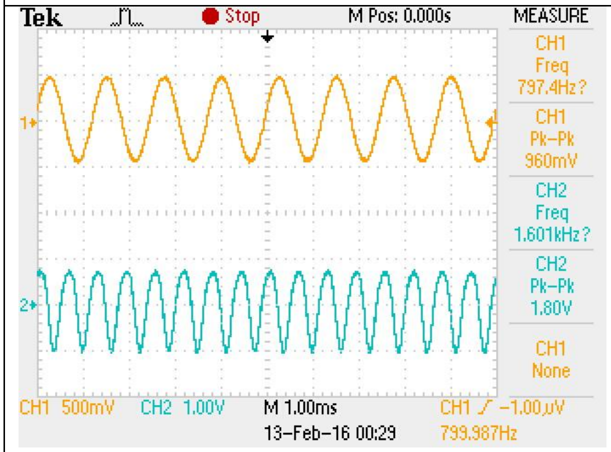


Figure 16: Scope trace of input and output signals with frequencies 800 and 1600 Hz respectively.

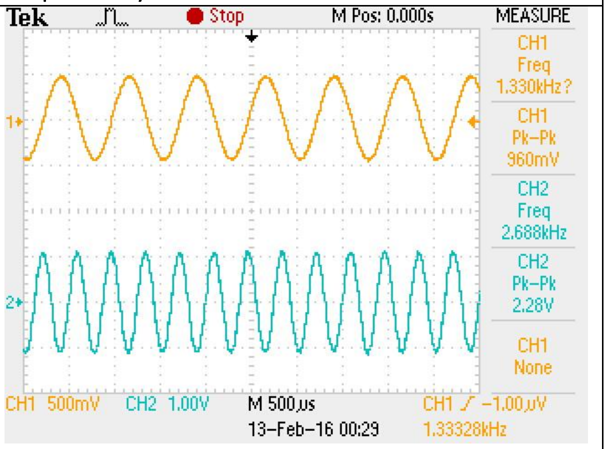


Figure 17: Scope trace of input and output signals with frequencies 1333 and 2666 Hz respectively.

However, when the input signal frequency is 1900 Hz, an amplitude modulation of 400 Hz with a base frequency of 3800 Hz was observed at the output, shown in the scope trace below (figure 18), which was expected to be simply a 3800 Hz rectified sinewave. We looked into the FFT of the output signal (figure 19), which has many additional harmonics. The frequencies of these newly introduced harmonics are always the integer multiples of 400 Hz. According to what we found in the FFT of full wave rectified sinewave in section 1.2.1, we thus concluded that these newly introduced harmonics form a full wave rectified sinewave of frequency 400 Hz (also see figure 20). Hence we observed an amplitude modulation of 400 Hz with a base frequency of 3800 Hz. Similarly, we found the frequency of envelope for input signal frequency *fin,* which is express in the equation below.

$Envelope\ (AM)\ Frequency = 8000 - 4 * fin,\ for\ 500 < fin < 2000\ and\ fin \neq$ $800, 1000, 1333, 2000.$

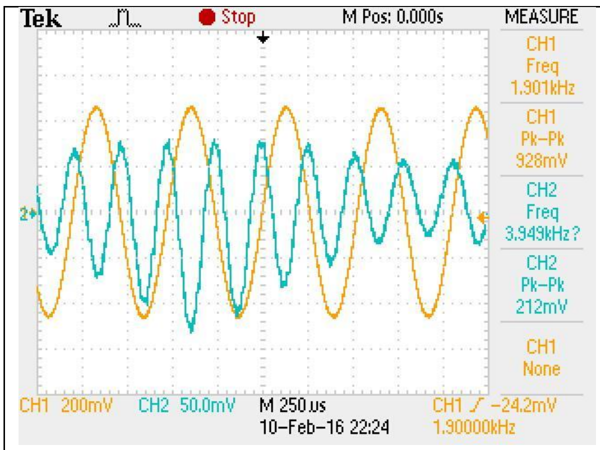


Figure 18: Scope trace of input and output signals with frequencies 1900 and 3800 Hz respectively.

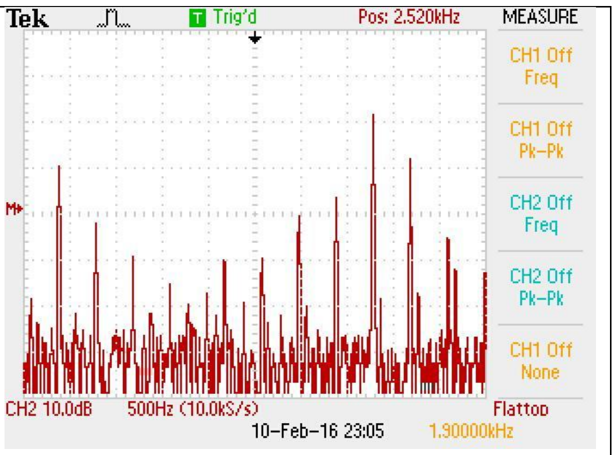


Figure 19: FFT of the output signal

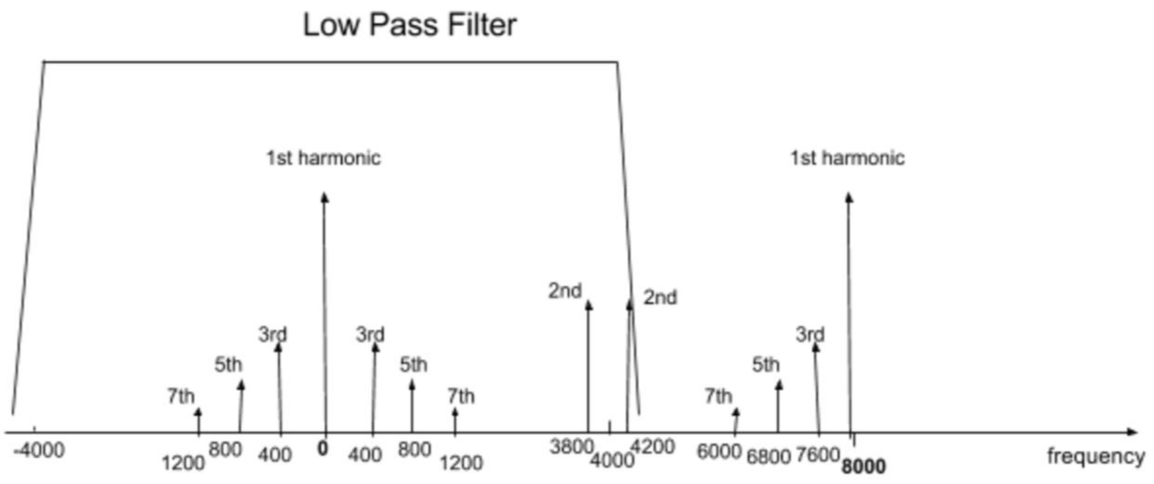


Figure 20: frequency response of the output after sampling at 8000 Hz when the input sinewave frequency is 1900 Hz

1.2.2.3 Input Frequency 2000 Hz < *f* < 4000 Hz

In this section, we were expecting aliasing effect of the output signal since the frequency of full wave rectified sinewave is larger than the Nyquist frequency, 4000 Hz. Thus we mainly focused on observing and analysing the aliasing effect of the output signal, and worked towards only one particular case where the input signal frequency is 3800 Hz. We concluded that with any input sinewave frequency above 2000 Hz and below 4000 Hz, the output signal is not a full wave rectified version of the input. A number of scope traces are also provided in this section in order to prove our conclusion.

Theoretically, the frequency response of the output signal should behave as displayed in figure 22. The reason for it is that the frequency of the full wave rectified sinewave is 7600 Hz (2*3800 Hz).

Before sampling, the theoretical frequency response of the rectified output behaves as displayed in figure 21. After sampling at 8000 Hz, its frequency response before sampling is duplicated and centred at the integer multiples of the sampling frequency, 8000 Hz. The result is illustrated in figure 22. Therefore, after reconstruction the output signal becomes a full wave rectified sinewave of frequency 400 Hz, which is not a full wave rectified version of the input. This effect is called aliasing.

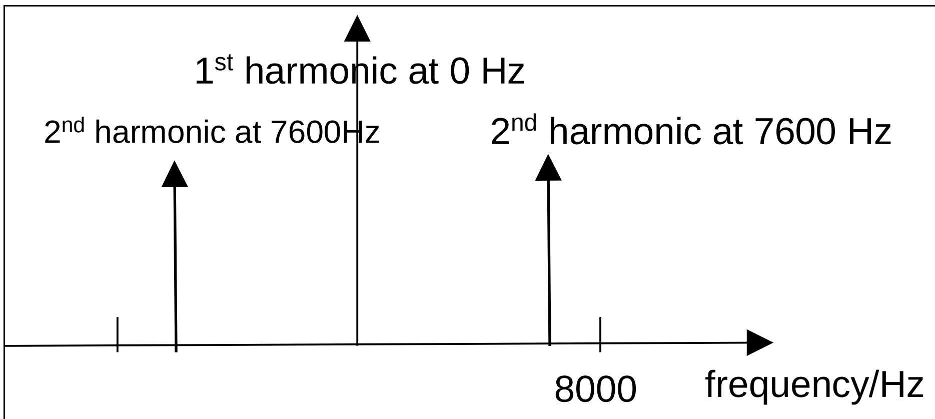


Figure 21: frequency response of full wave rectified sinewave before sampling (**only 1st and 2nd harmonics are displayed here)**

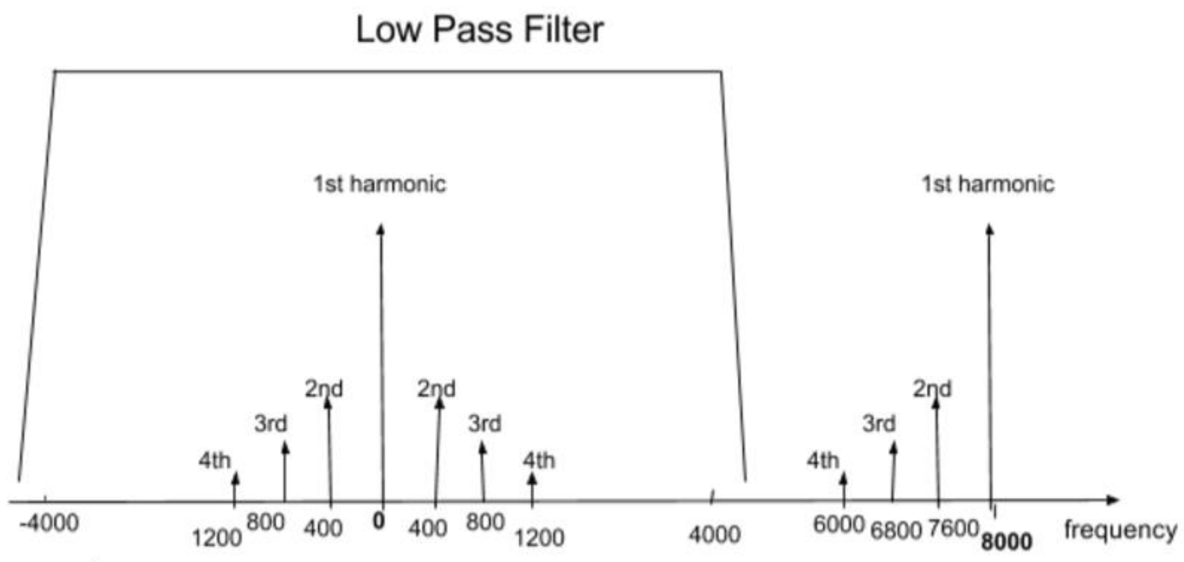


Figure 22: frequency response of the output after sampling at 8000 Hz when the input sinewave frequency is 3800 Hz

Similarly, we concluded that as the frequency of the input signal rises from 2000 Hz to 4000 Hz, the frequency of the output signal appeared in the oscilloscope would decrease from 4000 Hz to almost 0. In the 2000 to 4000 Hz input signal frequency band, the frequency of the output signal can be expressed by the equation below.

$$fo = fs - 2 * fin$$

A number of scope traces are provided below in order to prove the conclusion we just drew.

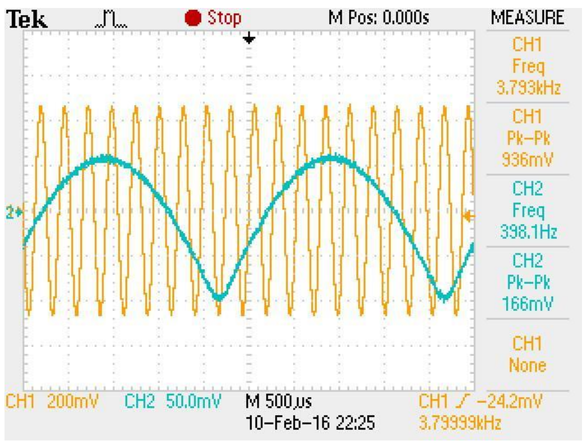


Figure 23: Scope trace of input and output signals with frequencies 3800 and 400 Hz respectively.

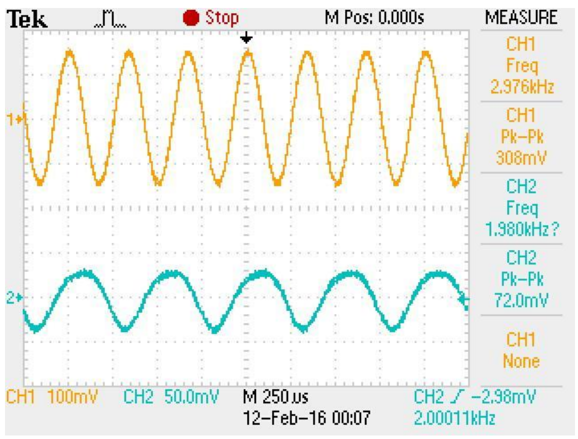


Figure 24: Scope trace of input and output signals with frequencies 3000 and 2000 Hz respectively.

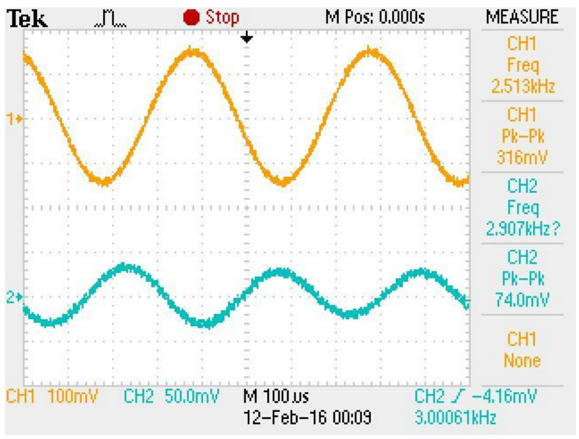


Figure 25: Scope trace of input and output signals with frequencies 2500 and 3000 Hz respectively.

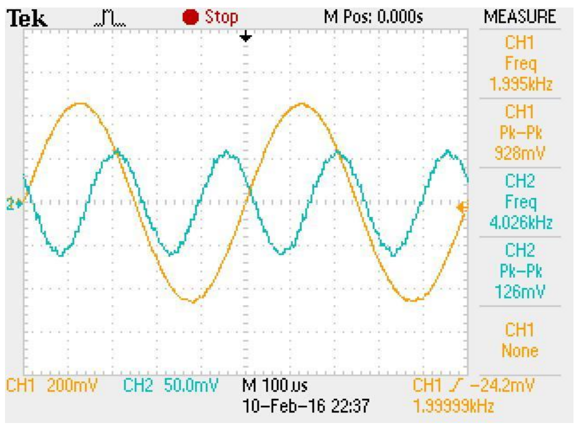


Figure 26: Scope trace of input and output signals with frequencies 2000 and 4000 Hz respectively.

From figure 26, we found that only at 2000 Hz of input signal frequency the output is a full wave rectified version of the input.

To summarise, in this section we found that aliasing effects always exist between 2000 and 4000 Hz input frequency band. Therefore, the output signal is not a full wave rectified version of the input for any frequency of input signal above 2000 Hz.

1.2.2.4. summary

From our investigated in different frequency ranges of input signal, we noticed various effects and thus summarised our findings in the table below.
Notation: f_{in} is input sinewave frequency and f_s is the sampling frequency, 8000 Hz.

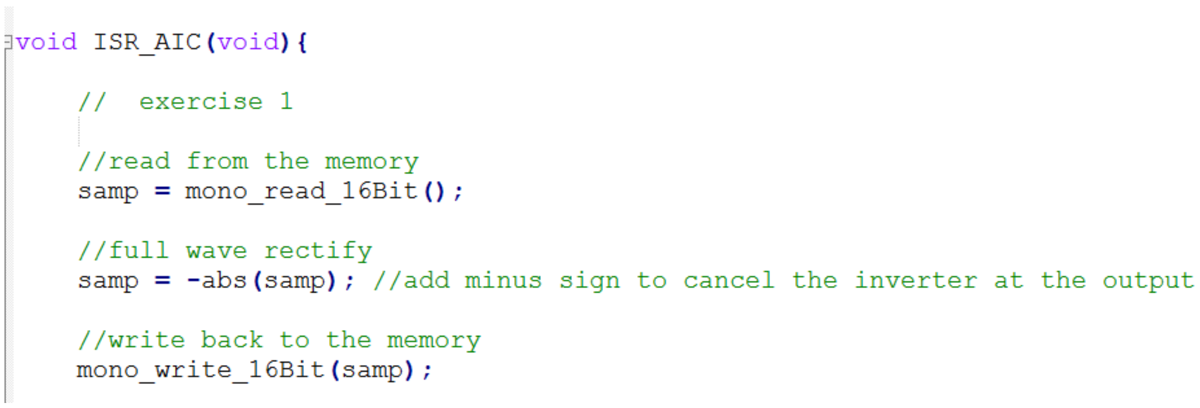
Input signal Frequency	**Output**
≤2000 Hz	• The output signal is a full wave rectified version of the input with frequency f_s, which is $2 \times f_{in}$. • However, amplitude modulation occurs for large f_{in} (500 to 2000 Hz) except special cases where f_{in} is the integer factor of 4000 Hz, such as 1000 Hz. The frequency of envelope equals to $(f_s - 2 * f_{in})$, and the base frequency is f_s, which is $2 \times f_{in}$.
2000-4000Hz	• Output is a rectified sinewave of frequency $(f_s - 2 * f_{in})$ in an envelope of frequency $(4 * f_{in} - f_s)$. Therefore it is not a full wave rectified version of the input since the output signal frequency $(f_s - 2 * f_{in}) \neq 2 * f_{in}$ for $f_{in} \geq 2000$.
Above 4000Hz	• Noise since the input signal must pass an anti-aliasing filter before being sampled • However, anti-aliasing filter is not perfect. Therefore, f_{in} slightly above 4000 Hz, such as 4200 Hz results in output signal, which is a rectified sinewave of frequency $(2 * f_{in} - f_s)$.

Therefore, we answered question two with the summary above. The output waveform will only be a full-wave rectified version of the input if the input from the signal generator is below 2000 Hz.

2. Code Explanation

2.1. Exercise One

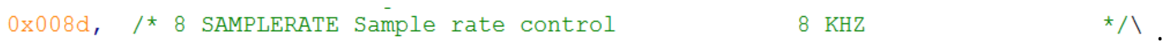
In exercise one, we wrote a function that services the interrupt, which is shown below.



```
void ISR_AIC(void){

    //  exercise 1

    //read from the memory
    samp = mono_read_16Bit();

    //full wave rectify
    samp = -abs(samp); //add minus sign to cancel the inverter at the output

    //write back to the memory
    mono_write_16Bit(samp);
```

The input signal is generated using a software called signal generator.
Essentially, our code manipulates rectifications of samples from the codec by performing reading and writing to the memory. It reads in a sample of input signal from the codec using `mono_read_16Bit();` function, which results in a 16 bit integer. The sample read is an average value of samples from the left and right audio ports. Then it changes the sample to negative instead of positive using `samp = -abs(samp);` in order to rectify the sample. We used a negative sign because there is an inverter placed at the output. After rectification, the full wave rectified sample is written back to the codec using `mono_write_16Bit(samp);` function. This function simply takes the argument *samp* and sends it to the left and right audio ports to achieve a mono output.

Please note that the function **ISR_AIC()** gets called 8000 times a second (sampling frequency 8000 Hz) by the setting,

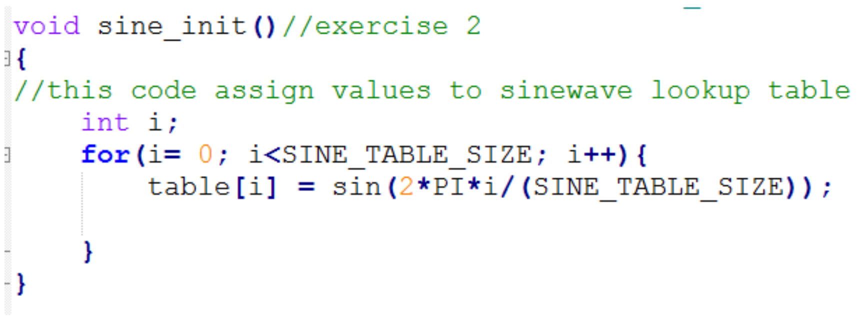


```
  0x008d,  /* 8 SAMPLERATE Sample rate control           8 KHZ                 */\
```
.

2.2. Exercise Two

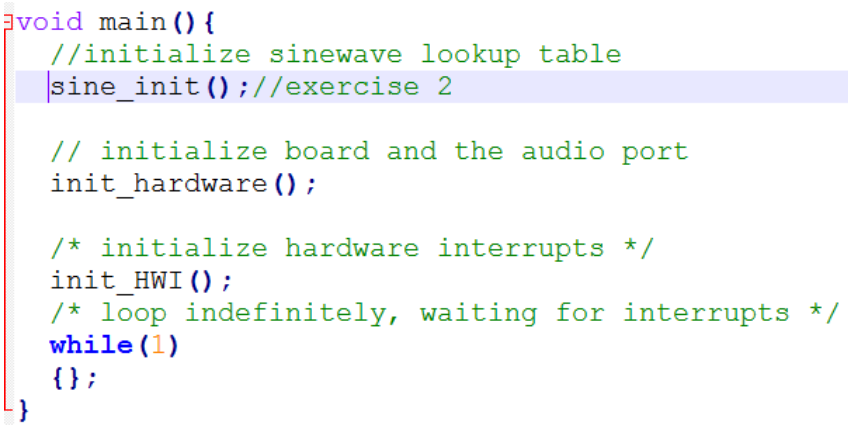
The second exercise generates a sinewave input using the lookup table method from lab 2, which no longer requires reading from the codec but only writing to it.

A sinewave table is initiated using the function below, which stores 256 values in the lookup table in the form of vector of size 256.



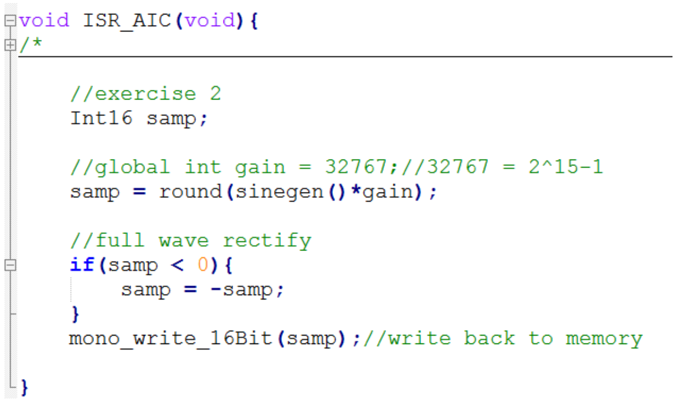
```
void sine_init()//exercise 2
{
//this code assign values to sinewave lookup table
    int i;
    for(i= 0; i<SINE_TABLE_SIZE; i++){
        table[i] = sin(2*PI*i/(SINE_TABLE_SIZE));

    }
}
```

The sinewave table generating function is called at the beginning of the main function (shown below) before interrupt is initialised.



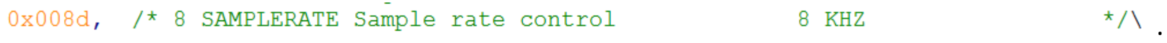
```
void main(){
    //initialize sinewave lookup table
    sine_init();//exercise 2

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

    /* initialize hardware interrupts */
    init_HWI();
    /* loop indefinitely, waiting for interrupts */
    while(1)
    {};
}
```

When the interrupt is initialized, Interrupt service routine ISR_AIC(void) executes. It firstly assigns a 16 bit integer variable called *samp*, which is a sample from the sinewave lookup table. The sample is then multiplied by a gain of 32767, which is (2^15 – 1). This is because the integer variable *samp* is a 16 bit signed integer, and the sample value from ***sinegen()*** function varies from -1 to 1. (A detailed explanation about ***sinegen()*** function is provided later.)
Therefore, we must take full advantage of this 16 bit integer by multiplying its maximum gain to fully use all of its bits, thus get an appropriate voltage at the output. Then we detect whether the sample is positive or negative using an ***if*** statement. If it's negative, we change it to positive. After that, we wrote the *samp* back to the left and right audio port using function mono_write_16Bit(samp);.



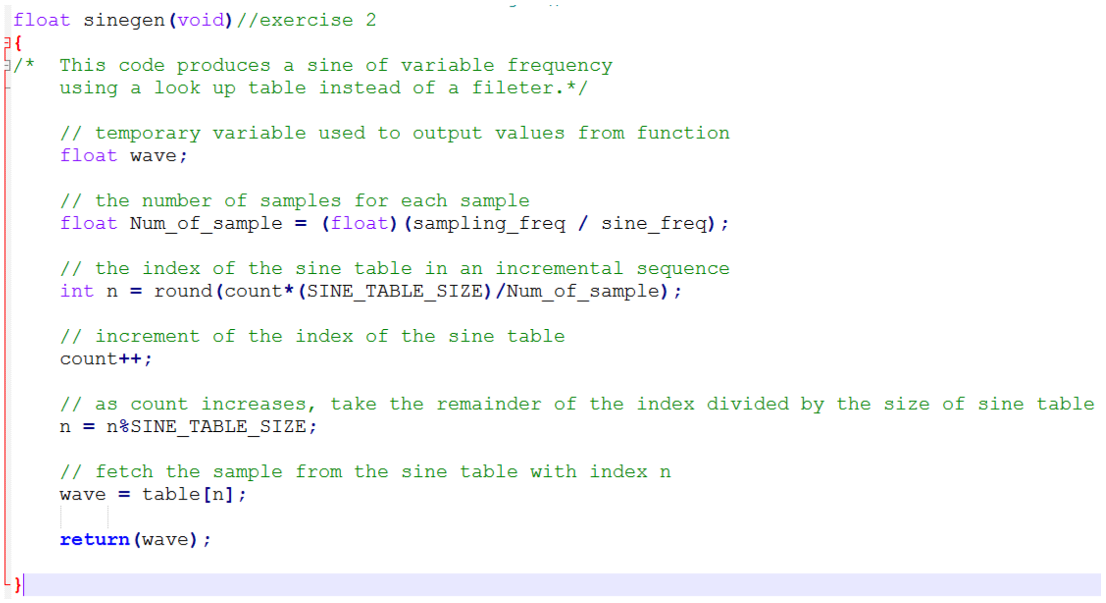
```
void ISR_AIC(void){
/*

    //exercise 2
    Int16 samp;

    //global int gain = 32767;//32767 = 2^15-1
    samp = round(sinegen()*gain);

    //full wave rectify
    if(samp < 0){
        samp = -samp;
    }
    mono_write_16Bit(samp);//write back to memory

}
```

We should note that the ISR ***ISR_AIC()*** gets called 8000 times a second (sampling frequency 8000 Hz) by the setting,



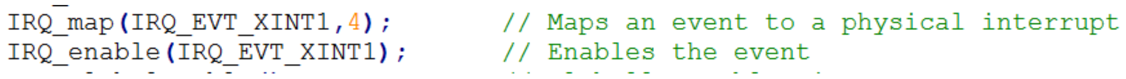
```
0x008d,  /* 8 SAMPLERATE Sample rate control            8 KHZ                   */\
```

Now let’s look at ***sinegen()*** function. It returns a sample from the sinewave lookup table. Its index increases, every time the function is called, in a similar way as we discussed in lab 2 report. In order to avoid loss of data, we treated everything as float. In addition, we ensured that loop handling is done correctly in the end by using ***n= n %SINE_TABLE_SIZE***.



```
float sinegen(void)//exercise 2
{
/*  This code produces a sine of variable frequency
    using a look up table instead of a fileter.*/

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

    // the number of samples for each sample
    float Num_of_sample = (float)(sampling_freq / sine_freq);

    // the index of the sine table in an incremental sequence
    int n = round(count*(SINE_TABLE_SIZE)/Num_of_sample);

    // increment of the index of the sine table
    count++;

    // as count increases, take the remainder of the index divided by the size of sine table
    n = n%SINE_TABLE_SIZE;

    // fetch the sample from the sine table with index n
    wave = table[n];

    return(wave);

}
```

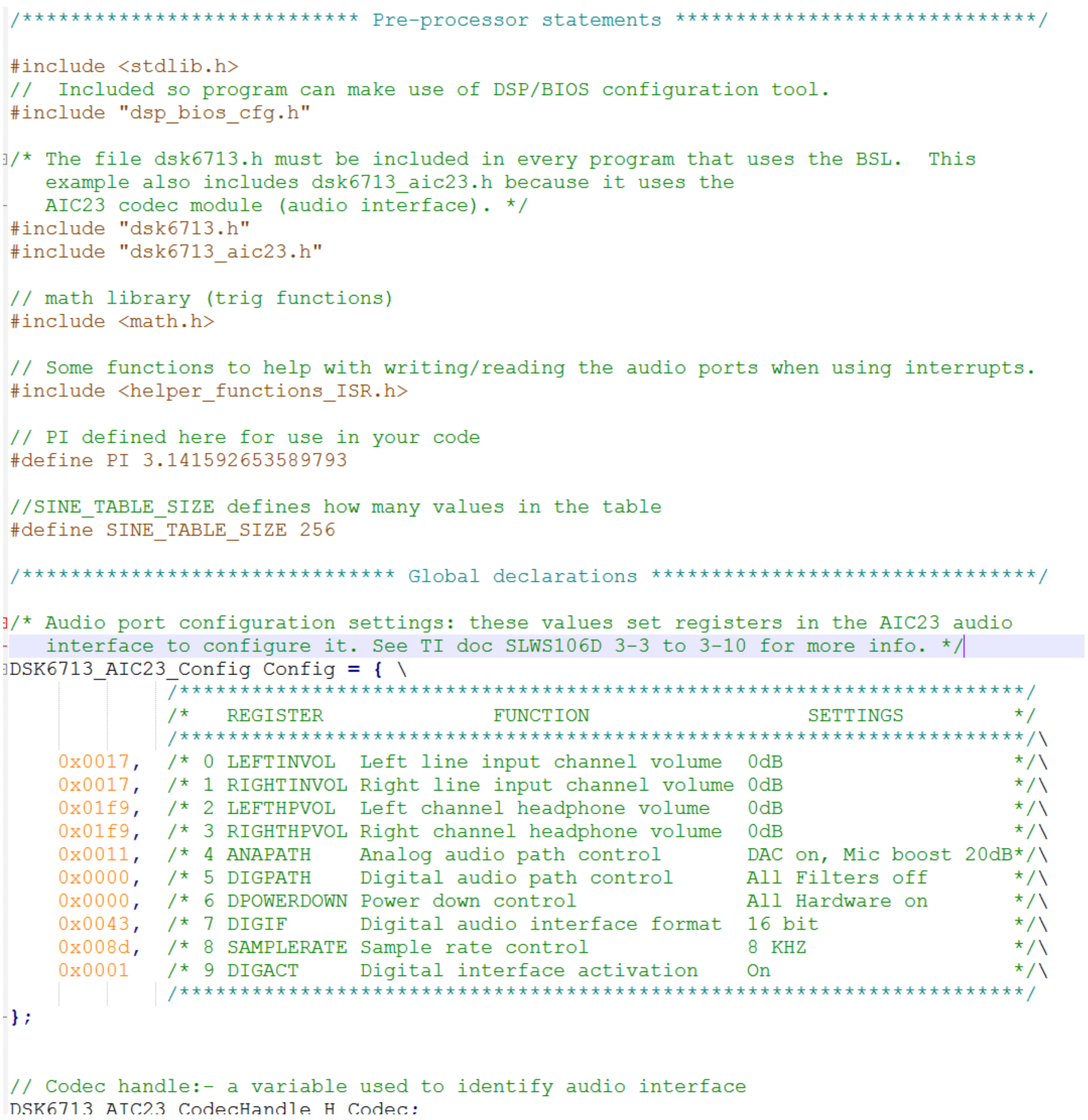
Moreover, we also modified the configuration file. Function *int_HWI* is also modified to reflect this change of configuration. The edited lines of *int_HWI* is displayed below. Simply, we just changed ***IRQ_EVT_RINT1*** to ***IRQ_EVT_XINT1***.



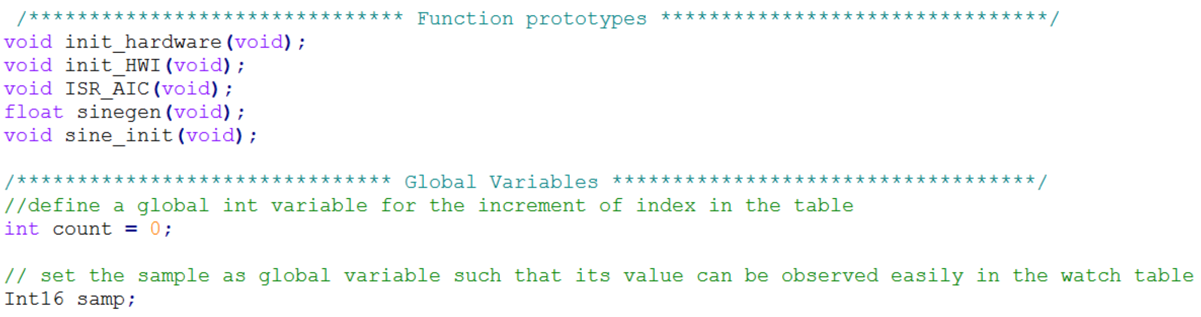
```
IRQ_map(IRQ_EVT_XINT1,4);        // Maps an event to a physical interrupt
IRQ_enable(IRQ_EVT_XINT1);       // Enables the event
```

3. Source Code

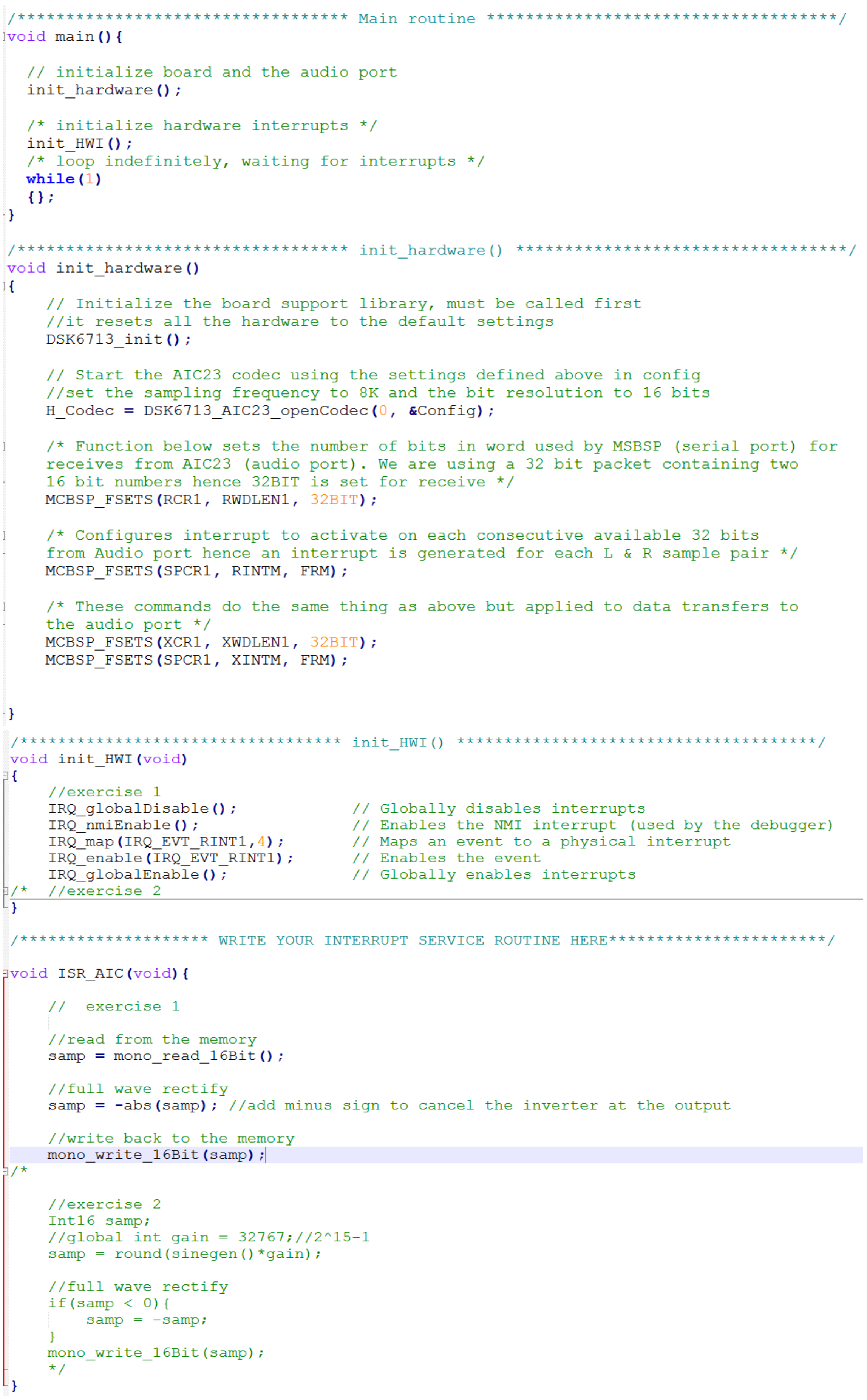
3.1. Exercise One Code



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

//SINE_TABLE_SIZE defines how many values in the table
#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;
```

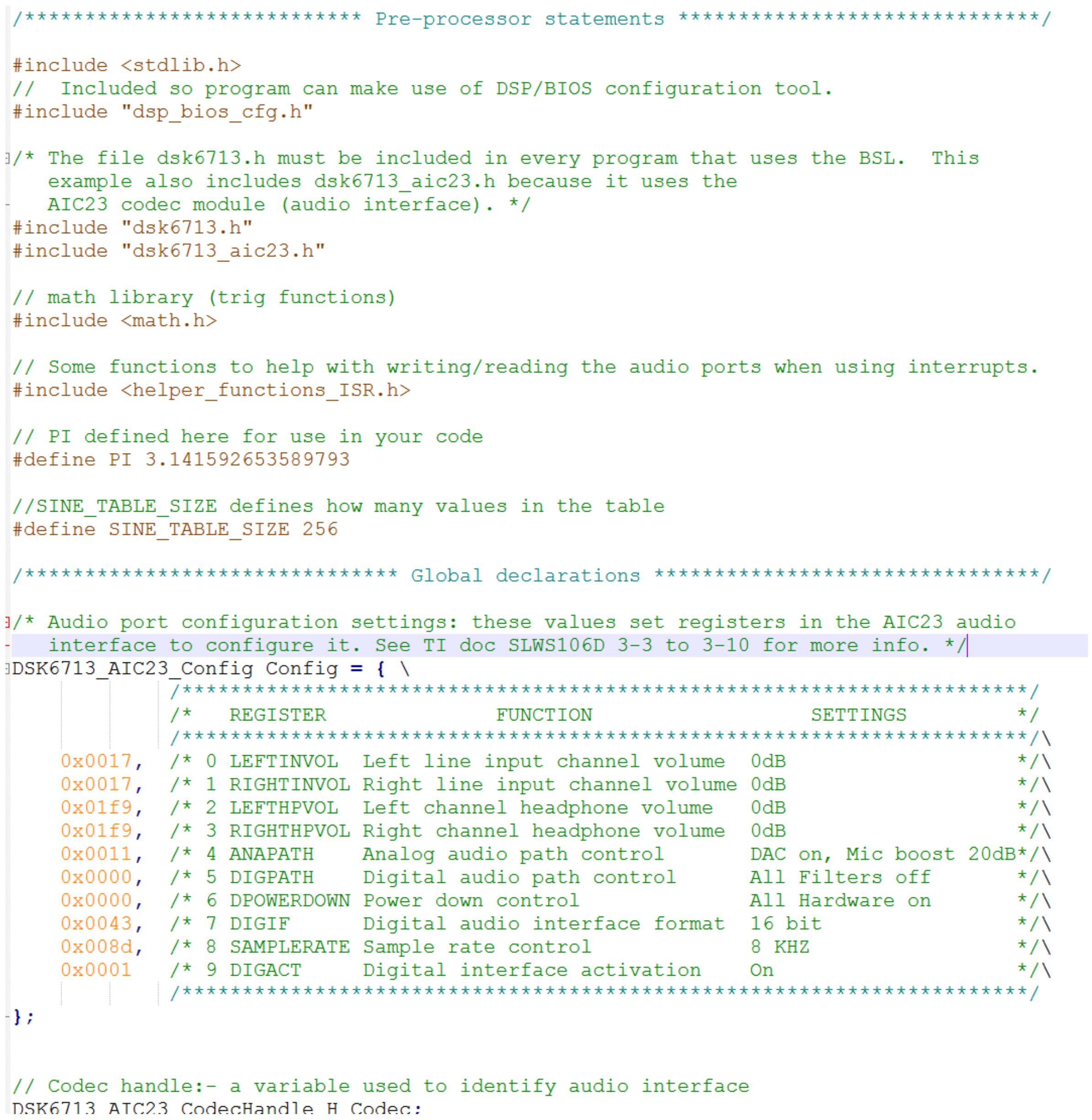


```
 /****************************** Function prototypes *******************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
float sinegen(void);
void sine_init(void);

/****************************** Global Variables ***********************************/
//define a global int variable for the increment of index in the table
int count = 0;

// set the sample as global variable such that its value can be observed easily in the watch table
Int16 samp;
```



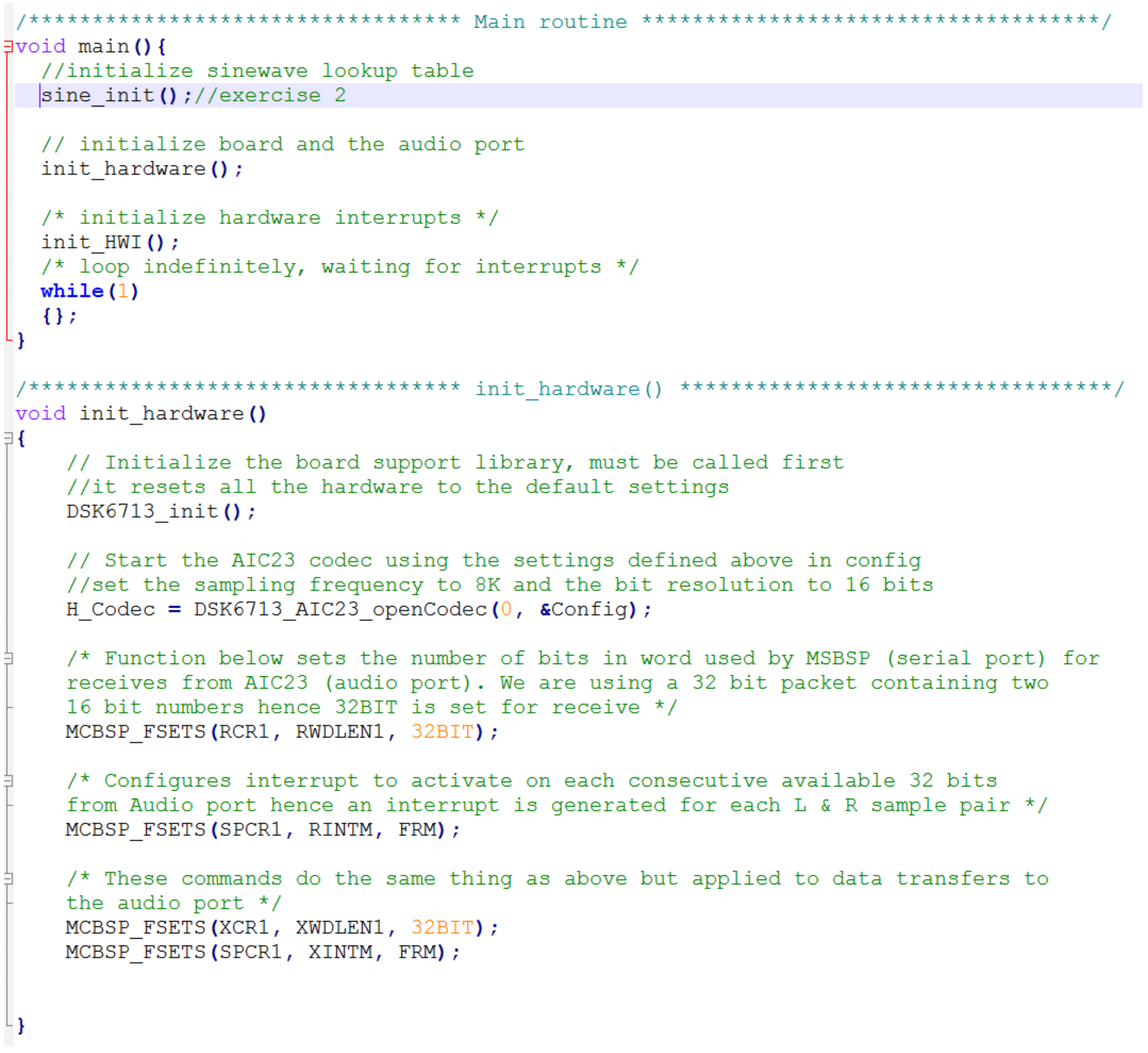
```c
/*********************************** Main routine *************************************/
void main(){

  // 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
    //it resets all the hardware to the default settings
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    //set the sampling frequency to 8K and the bit resolution to 16 bits
    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)
{
    //exercise 1
    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
/*  //exercise 2
}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/

void ISR_AIC(void){

    //  exercise 1

    //read from the memory
    samp = mono_read_16Bit();

    //full wave rectify
    samp = -abs(samp); //add minus sign to cancel the inverter at the output

    //write back to the memory
    mono_write_16Bit(samp);
/*

    //exercise 2
    Int16 samp;
    //global int gain = 32767;//2^15-1
    samp = round(sinegen()*gain);

    //full wave rectify
    if(samp < 0){
        samp = -samp;
    }
    mono_write_16Bit(samp);
    */
}
```

3.2. Exercise Two code

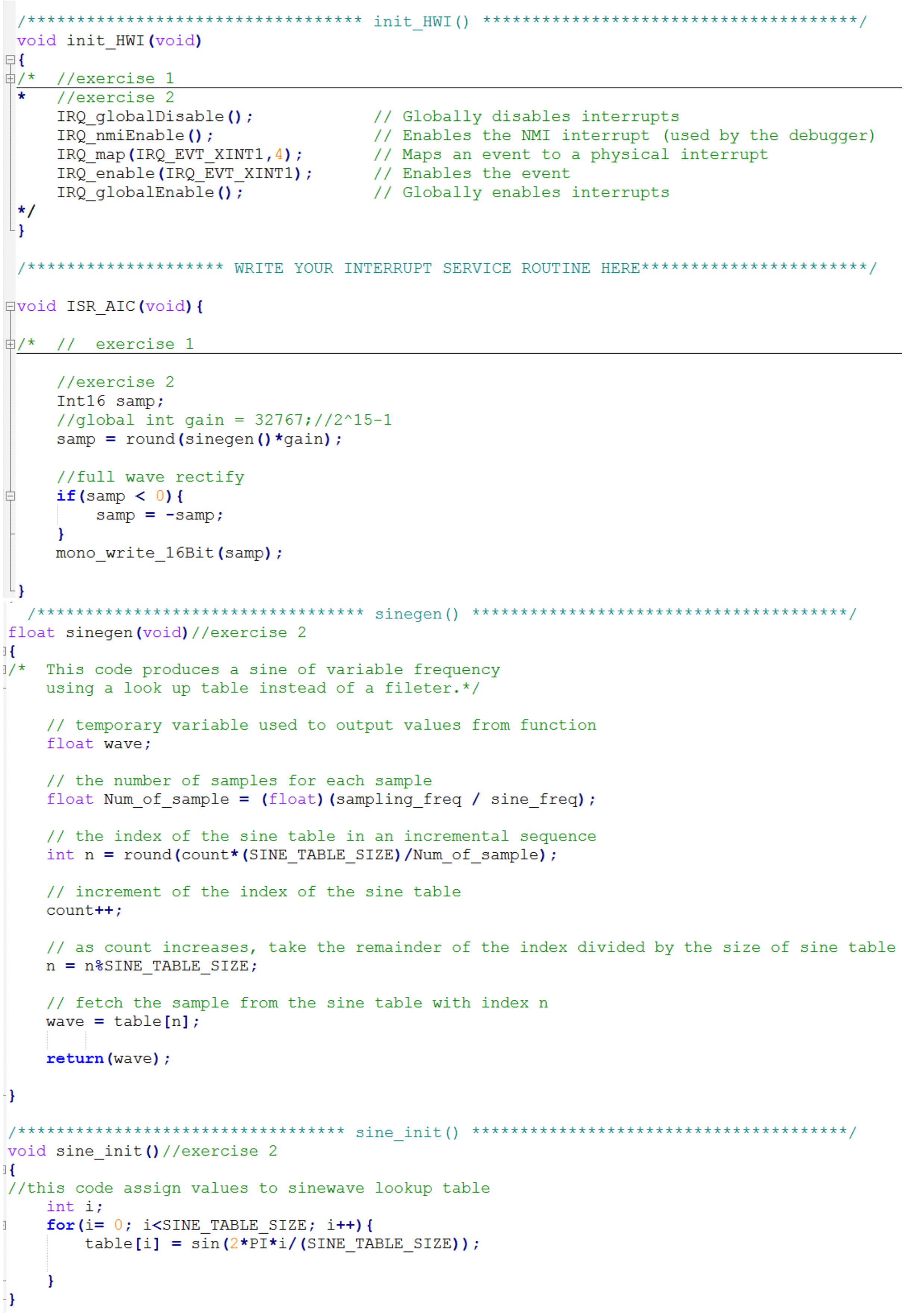


```c
/***************************** Pre-processor statements ******************************/

#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

//SINE_TABLE_SIZE defines how many values in the table
#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;
```

```c
 /******************************** Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
float sinegen(void);
void sine_init(void);

/******************************** Global Variables **********************************/
//define a global int variable for the increment of index in the table
int count = 0;

// set the sample as global variable such that its value can be observed easily in the watch table
Int16 samp;

//Array of floats that contains SINE_TABLE_SIZE elements
float table [SINE_TABLE_SIZE];//exercise 2

/* Sampling frequency in HZ. Must only be set to 8000, 16000, 24000
32000, 44100 (CD standard), 48000 or 96000  */
int sampling_freq = 8000; //exercise 2

/* 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;    //exercise 2

//signal gain for exercise 2
//set the gain as global variable such that its value can be changed easily in the watch table
int gain = 32767;//2^15-1
```



```c
 /********************************** Main routine ************************************/
void main(){
  //initialize sinewave lookup table
  sine_init();//exercise 2

  // 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
    //it resets all the hardware to the default settings
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    //set the sampling frequency to 8K and the bit resolution to 16 bits
    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)
{
/*  //exercise 1
*   //exercise 2
    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
*/
}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/

void ISR_AIC(void){

/*  //  exercise 1

    //exercise 2
    Int16 samp;
    //global int gain = 32767;//2^15-1
    samp = round(sinegen()*gain);

    //full wave rectify
    if(samp < 0){
        samp = -samp;
    }
    mono_write_16Bit(samp);

}
  /********************************** sinegen() ****************************************/
float sinegen(void)//exercise 2
{
/*  This code produces a sine of variable frequency
    using a look up table instead of a fileter.*/

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

    // the number of samples for each sample
    float Num_of_sample = (float)(sampling_freq / sine_freq);

    // the index of the sine table in an incremental sequence
    int n = round(count*(SINE_TABLE_SIZE)/Num_of_sample);

    // increment of the index of the sine table
    count++;

    // as count increases, take the remainder of the index divided by the size of sine table
    n = n%SINE_TABLE_SIZE;

    // fetch the sample from the sine table with index n
    wave = table[n];

    return(wave);

}

/********************************** sine_init() ****************************************/
void sine_init()//exercise 2
{
//this code assign values to sinewave lookup table
    int i;
    for(i= 0; i<SINE_TABLE_SIZE; i++){
        table[i] = sin(2*PI*i/(SINE_TABLE_SIZE));

    }
}
```

4. Reference

[1]: http://mathworld.wolfram.com/FourierSeriesSquareWave.html

[2]: http://www.calpoly.edu/~fowen/me318/FourierSeriesTable.pdf