Sebastian Grubb sg3510 24th February 2013

RTDSP Lab 3

In the following lab we look at the various aspects of a software full wave rectifier of an input signal, by using a real-time DSP kit which samples a signal at 8kHz and outputs it as well as a case where a rectified sinewave is generated then outputted.

Contents

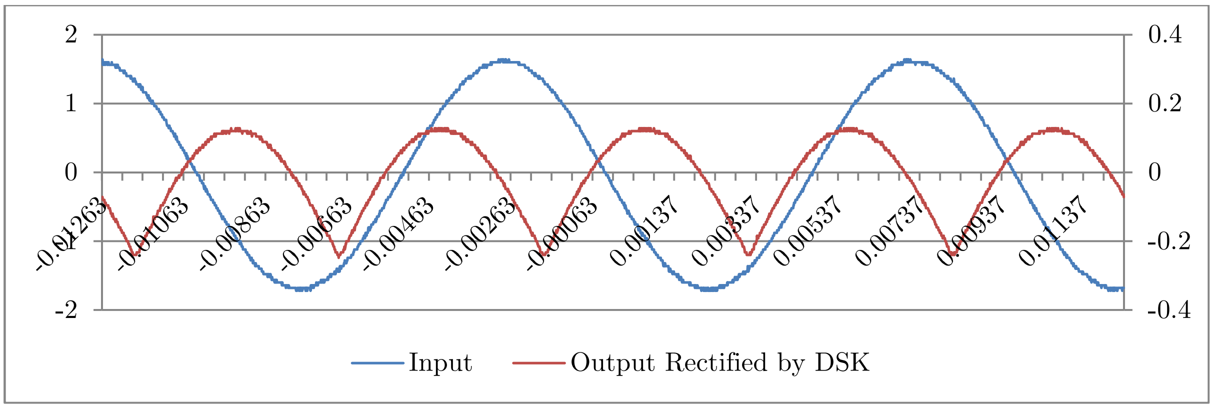
Questions

Note: data collection in the oscilloscope was all done with the invert option selected for both channels.

In this part we look at various effects which occur when full wave rectifying a signal.

Question 1

From Graph 1, we notice that the full rectified sinewave output is centered around 0 and as such also includes some negative values. The reason for this is that at the output of the DSP we have a high pass filter which removes any DC bias. Another to look at it is that the capacitor at the output (which forms part of the filter) "differentiates" the signal which removes all fixed terms (though this is not exactly what happens as the output is not, as implied, differentiated). The reason for this DC bias is that it avoids a speaker diaphragm to be flexed in one direction all the time, which would damage it.

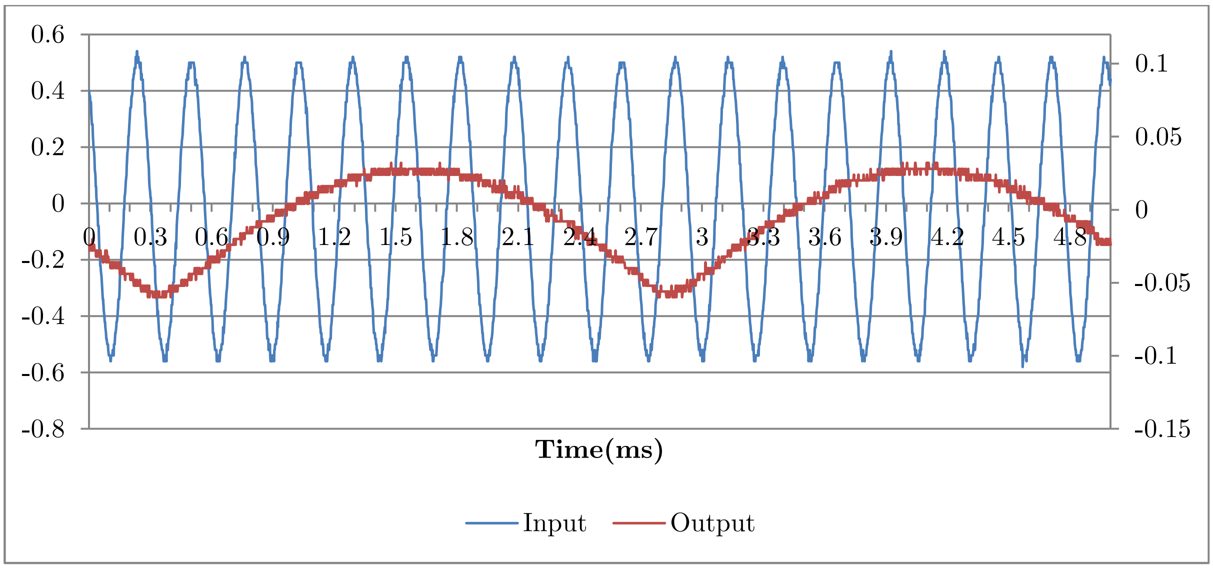


Graph 1. Plot of collected data of a 100Hz sinewave with a rectified output. We notice that there is a lag between the rectified signal and the input – this is simply due to the processing time

Question 2

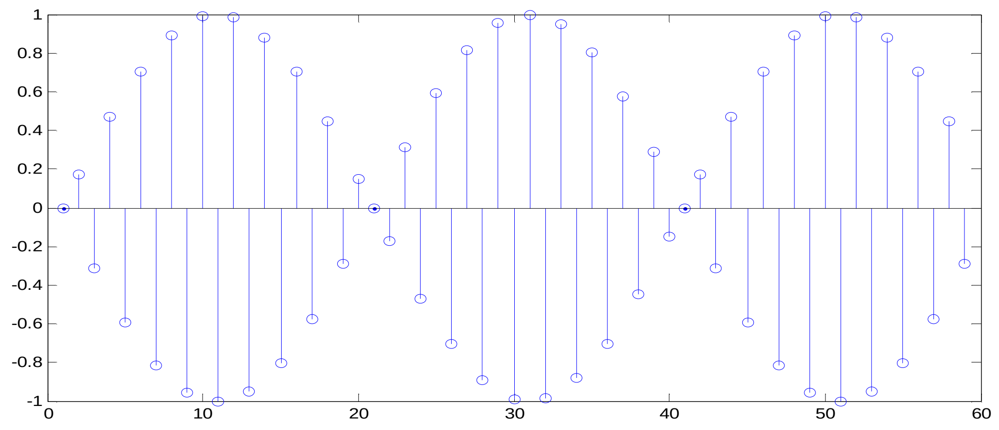
4000Hz to 3000Hz

At high frequencies, for example around 3800Hz (as shown in Graph 2) the rectified sinewave output is at a much lower frequency. In fact the frequency of this sinewave can be determined by the formula $2 \times (\frac{f_{samp}}{2} - f_{gen})$ which works for this case (this is valid for all frequencies above 2000Hz).



Graph 2. Plot of data collected for an input signal of 3800Hz. The data collected at the output is a rectified wave of frequency 400Hz.

The reason for this is due to effects observed in the previous lab wherein we observed that when sampling a signal of 3800Hz at 8000Hz an amplitude modulation occurred. Graph 3 helps explain this.

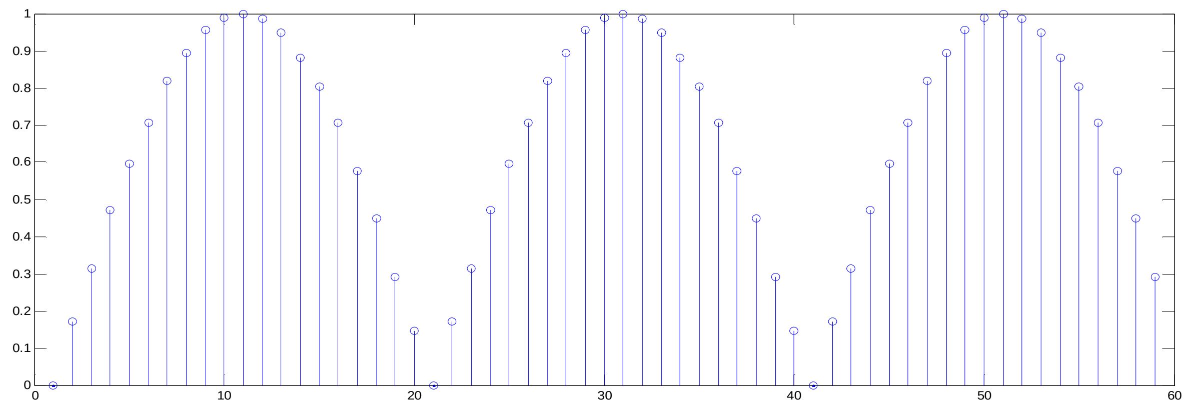


Graph 3. Plot of samples collected for a 3800Hz signal sampled at 8000Hz. i.e. each sample is separated by 125ms.

The amplitude modulation effect which arises from 3800Hz also appearing as a 4200Hz signal if Nyquist frequency is 4000Hz we get:

$$\cos(3800 \times 2\pi \times t) + \cos(4200 \times 2\pi \times t) = 2\cos(4000 \times 2\pi \times t)\cos(200 \times 2\,\pi \times t)$$

Thus taking the absolute value we get Graph 4 as $abs\left(\cos\left(4000 \times 2\pi \times \frac{t}{80000}\right)\right) = abs(\cos(\pi t)) = 1, \forall t \in \mathbb{Z}$ which leads to $abs(2\cos(4000 \times 2\pi \times t)\cos(200 \times 2\,\pi \times t)) = abs(2\cos(200 \times 2\pi \times t))$ which has a frequency of 400Hz, as we take the absolute value.

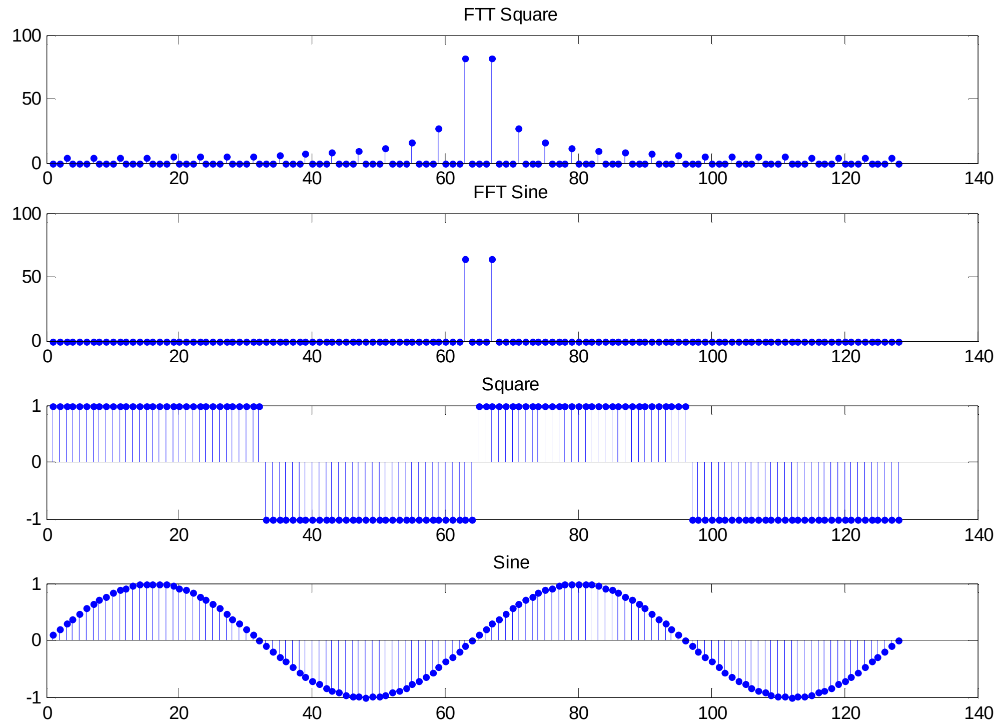


Graph 4. Plot of the absolute value of samples collected for a 3800Hz signal sampled at 8000Hz. i.e. each sample is separated by 125ms. (note that the amplitude is normalized.)

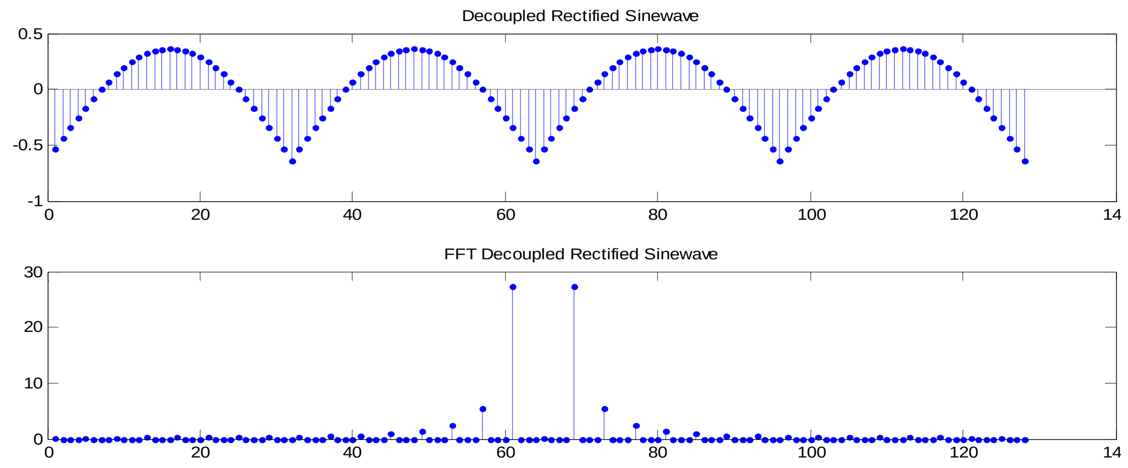
We can also look at this problem in frequency domain where Graphs 5 and 6 allow us to view what a rectified sinewave looks like in frequency domain and how it arises. Graph 5 shows us how it arises by multiplying a sinewave with a square with, which results in convolution in frequency domain. It follows from this that the FT of a decoupled rectified sinewave is (we ignore the coefficients as they are not relevant):

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

This is important to know as we can notice that a harmonic is present at every multiple of the double of the original frequency of the sinewave.

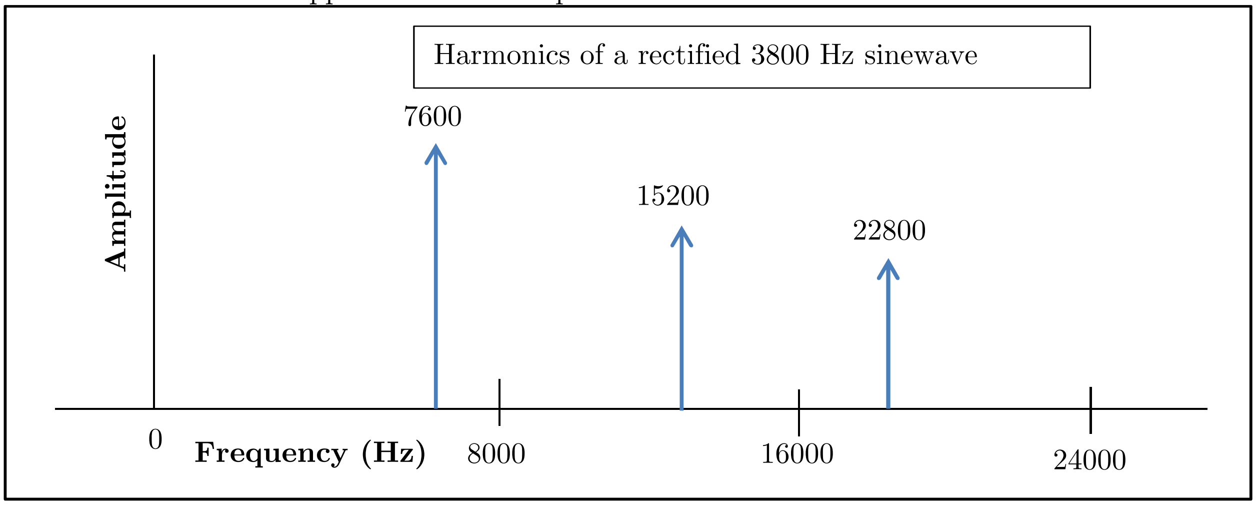


Graph 5. Plot of the (absolute) Fourier transform of a sinewave with its matching frequency sinewave as well as the time domain plots to demonstrate that a rectified sinewave is equivalent to $SquareWave(t) \times Sin(t)$, so long as the frequencies and phases match.



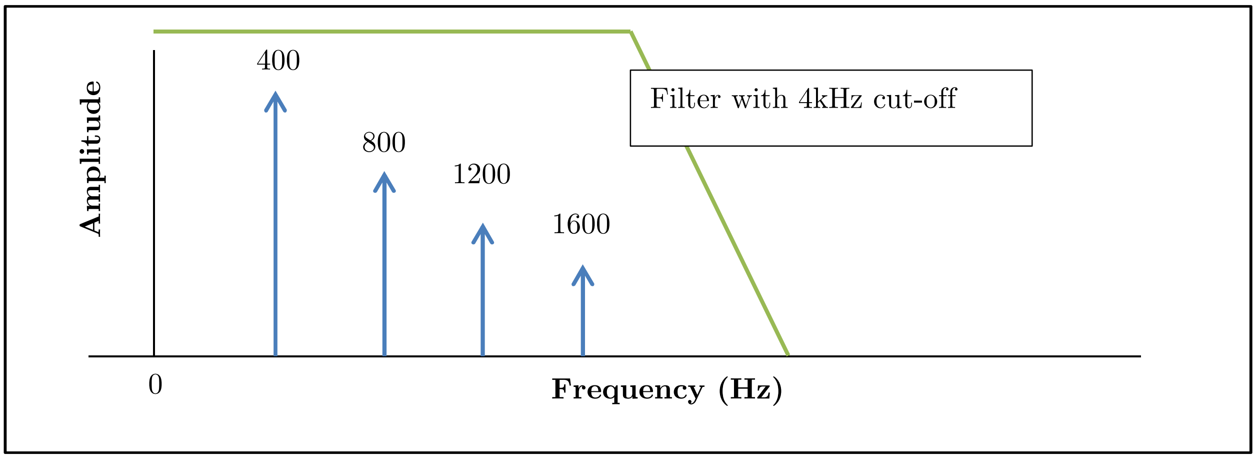
Graph 6. Plot of a decoupled rectified sinewave in time and frequency domain.

Let us now see what happens when we output a rectified sinewave at 3800Hz.



Graph 7. Plot of a decoupled rectified sinewave at 3800, with the sampling frequency and its harmonics displayed.

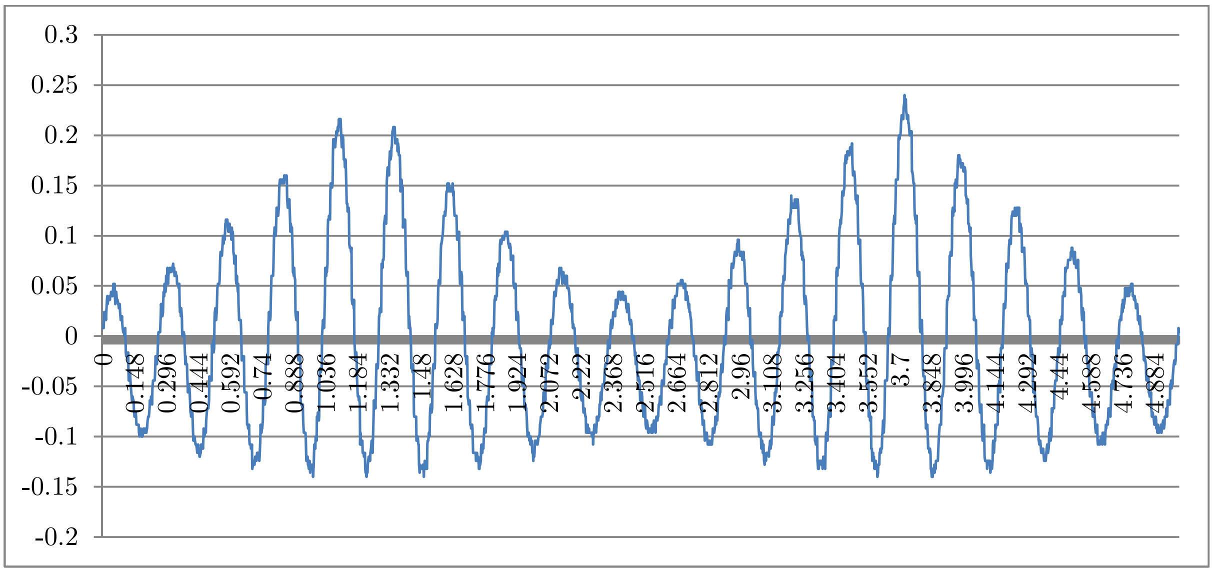
From Graph 7 we also know that the harmonics will be convoluted with the sampling frequency and all its multiples in frequency domain, thus resulting in what we see in Graph 8 and ultimately explaining why the output is a rectified sinewave of 400Hz.



Graph 8. Plot of a decoupled rectified sinewave at 3800, with the sampling frequency and its harmonics displayed sampled at 8kHz. Each term appears due to the repetitions of the sinewave. We notice that this is effectively the Fourier transform of 400Hz wave.

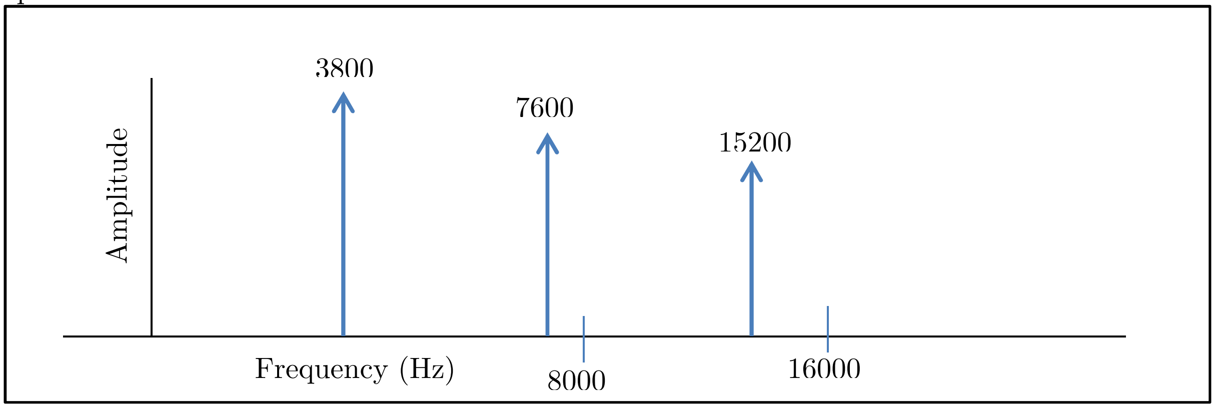
2000Hz to 1000Hz

At 1900Hz we observe another unusual effect which is amplitude modulation of a signal expected to be simply a 3900Hz rectified sinewave, as can be seen in Graph 9.



Graph 9. Plot of output data collected for an input sinewave of 1900Hz. An amplitude modulation of 400Hz with a base frequency of 3800Hz is observed.

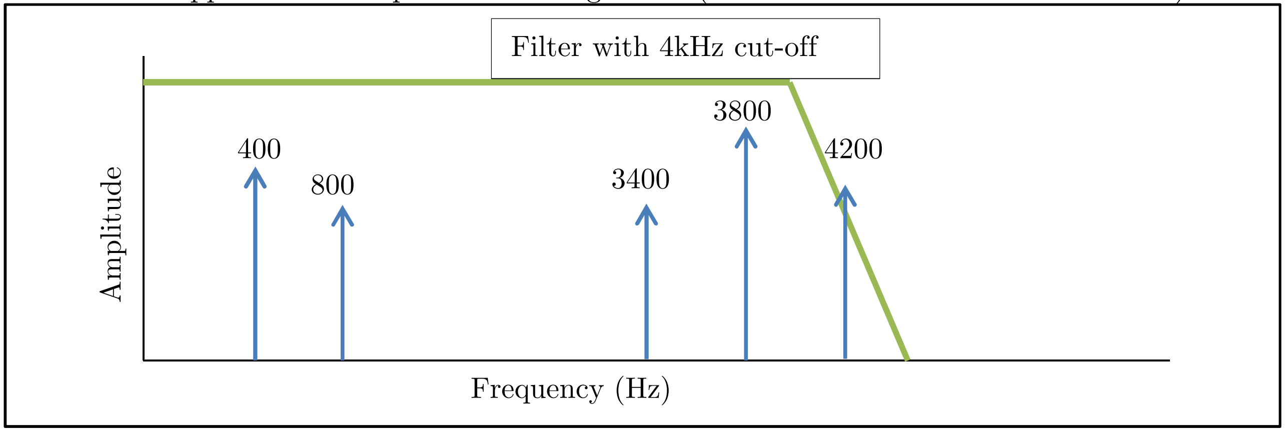
The reason for this happening is that a 1900Hz rectified sinewave would have the following spectra:



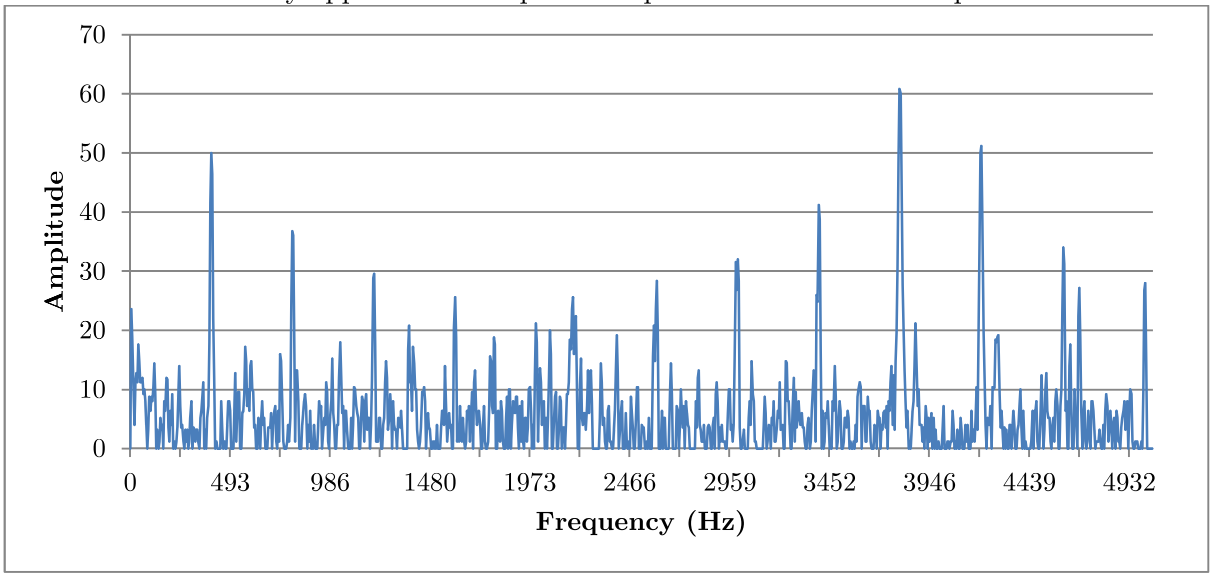
Which when sampled at 8 kHz, i.e. convoluted with $\delta(f) + \delta(f - 8000) + \delta(f + 8000) + \delta(f + 16000) + \delta(f - 16000)$ etc, gives us (we will only consider dominant frequencies and ignore the different amplitudes):

$$\begin{aligned}&(\delta(f) + \delta(f - 8000) + \delta(f + 8000) + \delta(f - 16000) + \delta(f + 16000))\\ &\quad * (\delta(f - 3800) + \delta(f + 3800) + \delta(f - 7600) + \delta(f + 7600) + \delta(f - 11400)\\ &\quad + \delta(f + 11400) + \delta(f - 15200) + \delta(f + 15200)) =\\ &\delta(y - 31200) + \delta(y - 27400) + \delta(y - 23600) + \delta(y - 23200) + \delta(y - 19800) + \delta(y\\ &\quad - 19400) + \delta(y - 15600) + \delta(y - 15200) + \delta(y - 12200) + \delta(y - 11800)\\ &\quad + \delta(y - 11400) + \delta(y - 8400) + \delta(y - 7600) + \delta(y - 7200) + \delta(y - 4600)\\ &\quad + \boldsymbol{\delta(y - 4200)} + \boldsymbol{\delta(y - 3800)} + \boldsymbol{\delta(y - 3400)} + \boldsymbol{\delta(y - 800)} + \boldsymbol{\delta(y - 400)}\\ &\quad + \boldsymbol{\delta(y + 400)} + \boldsymbol{\delta(y + 800)} + \boldsymbol{\delta(y + 3400)} + \boldsymbol{\delta(y + 3800)} + \boldsymbol{\delta(y + 4200)}\\ &\quad + \delta(y + 4600) + \delta(y + 7200) + \delta(y + 7600) + \delta(y + 8400) + \delta(y + 11400)\\ &\quad + \delta(y + 11800) + \delta(y + 12200) + \delta(y + 15200) + \delta(y + 15600) + \delta(y\\ &\quad + 19400) + \delta(y + 19800) + \delta(y + 23200) + \delta(y + 23600) + \delta(y + 27400)\\ &\quad + \delta(y + 31200)\end{aligned}$$

Which when applied to a low pass 4Hz filter gives us (in bold are the concerned harmonics):



As we have ignored some higher harmonics for the signal we expect the real spectrum to include a few more, smaller harmonics. However we can see that the main components, 400Hz and 3800Hz, are present. The components at 1200Hz, 3000Hz etc. will appear for the same reasons that they appeared in Graph 8. Graph 10 shows all the components.



Graph 10. Plot of collected data for the FFT of the signal seen in Graph 9. The main harmonics, 400 and 3800Hz are easily noticeable as well as the smaller harmonics.

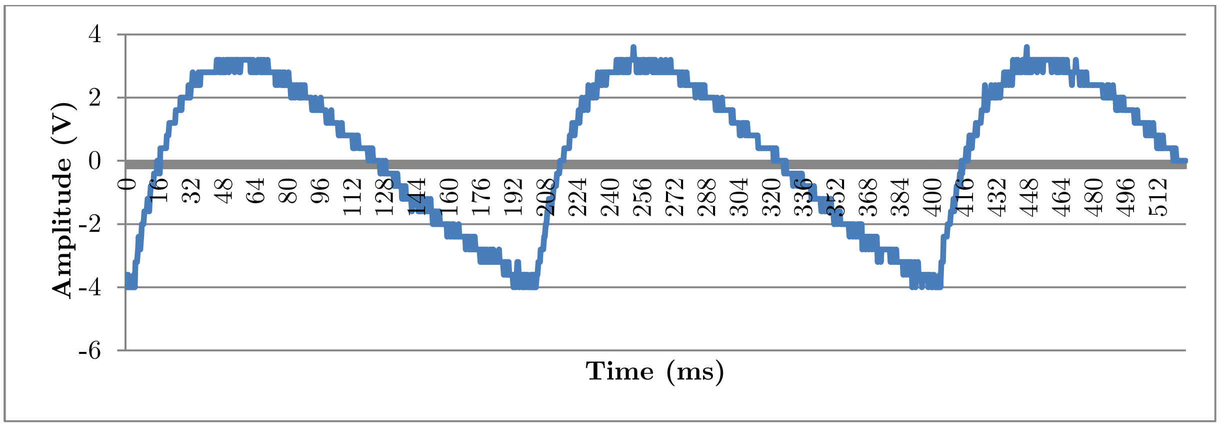
To summarise we can conclude that only less than 2000Hz does a valid (i.e. at matching frequency) full-wave rectified sinewave appear. However we notice amplitude modulation effects from 1000Hz upwards to 3000Hz.

Very low frequencies (<10Hz)

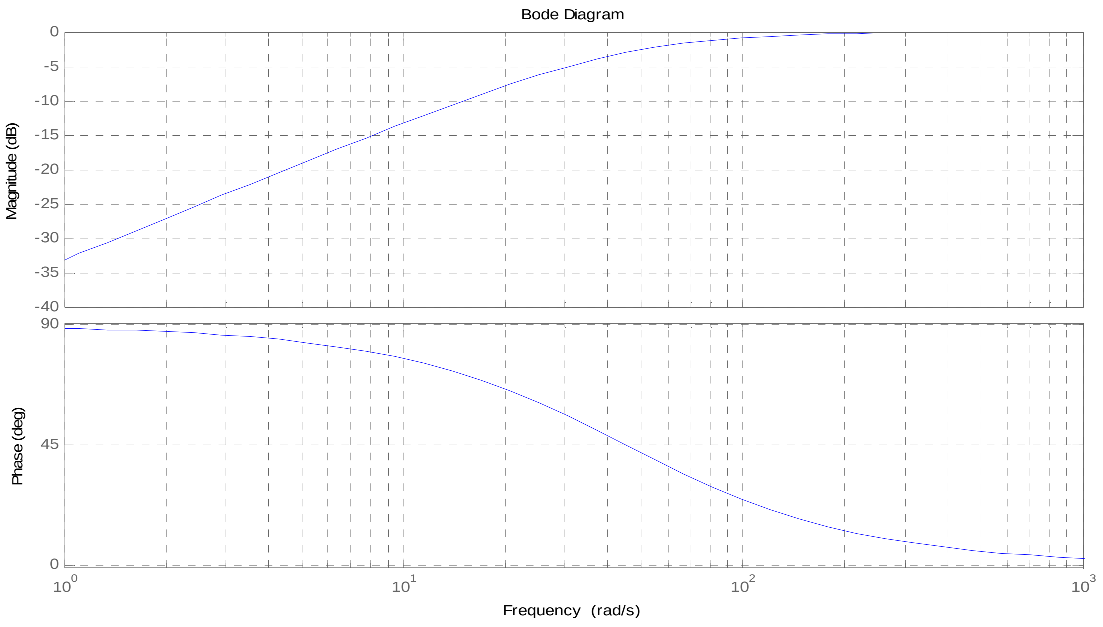
At very low frequencies, below 10 Hz or above 3990Hz (which creates a low frequency output), we observe a slight distortion in our signal, as seen in Graph 11.

The reason for this is the high pass filter that is present at the output of the DAC. Indeed it introduces an amplitude decrease but more important a phase shift, as can be seen from Graph 12.

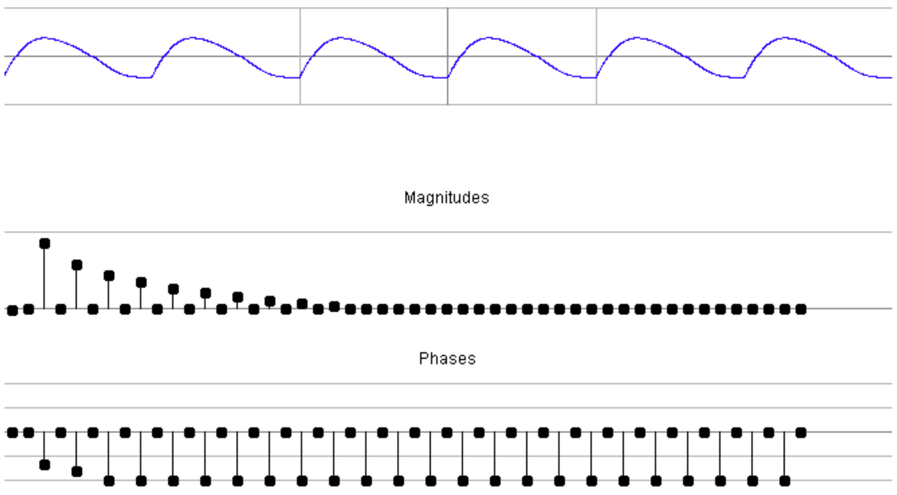
Paul Falstad's website (Falstad, 2008) provides us with the means of verifying this claim by allowing us to change the phase shift of the first few harmonics of the rectified sinewave (Graph 13). From this we can conclude that this is the result of a phase (and also amplitude) change due to the high-pass filter.



Graph 11. Plot of output data collected for an input sinewave of 3997.5Hz.



Graph 12. Bode plot of the output high-pass filter, using values provided in the datasheet.



Graph 13. Plot of a Fourier transform of a distorted rectified sinewave with the sinewave itself. The phases for the first few signals were manually decreased to match the output in Graph 11.

Above 4000Hz

At 4000Hz only a DC value can appear for a rectified version of the signal as the sampled function is $\sin\left(4000 \times 2\pi \times \frac{t}{8000} + \theta\right)_{\substack{t=0,1,2\dots \\ \theta \text{ represents any phase shift}}}$, which when we take the absolute value gives us a constant dependant on θ. This constant is treated as DC bias and thus removed by the output filter.

Past 4000Hz, harmonics of the rectified sinewave also get "reflected" back down to frequencies less than 4000Hz due to 8 kHz sampling thus getting resulting in another set of rectified sine waves with frequencies ranging from 4kHz to 0kHz. The system works such that:

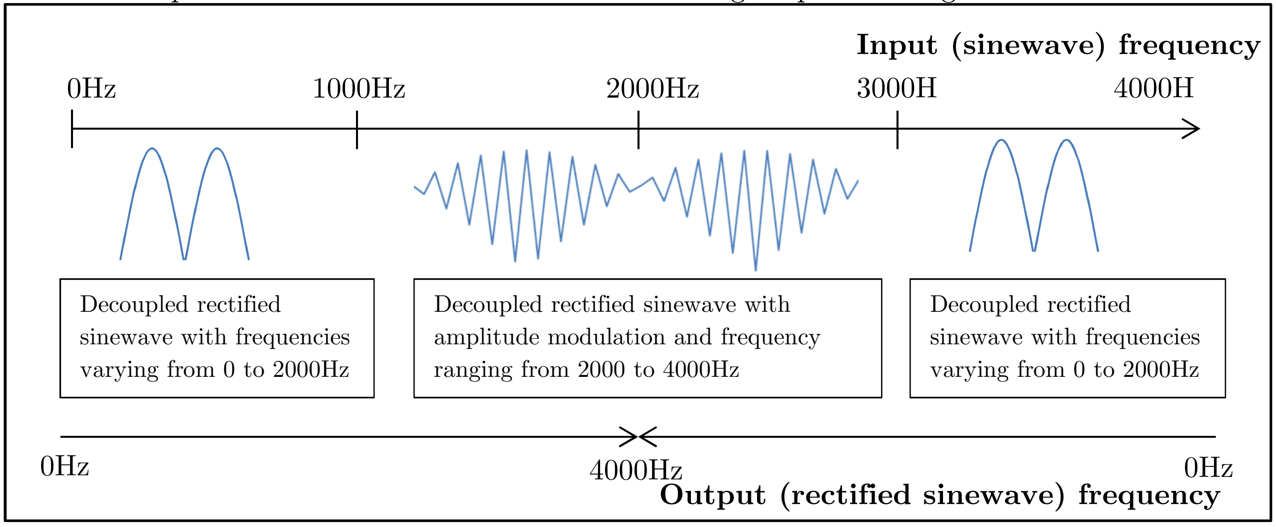
$$output(f_{input}) = output(f_{input} + 4000), \forall\, 0Hz < f_{input} \leq 4000Hz$$

Summary

We have noticed that for each frequency range various effects occur with the following table summarising the findings.
Notation: f_{gen} is the generated sinewave frequency and f_{samp} is the sampling frequency i.e. 8000Hz

Frequency	**Effect on output**
Above 4000Hz	System is basically a shifted version of the "base" system by 4000Hz
4000-3000Hz	• Output is rectified sinewave of $f_{samp} - 2f_{gen}$ Hz, i.e. output will range from a 2000Hz decoupled rectified sinewave to 0Hz. The specifics are explained earlier however the general idea is that the between 3000 Hz and 4000Hz the only harmonics that appear are due to sampling at 8kHz (indeed $3000Hz \times 2 > Nyquist$) leaving the images of the harmonics appear, which create the rectified sinewave.
3000-2000Hz	• Output is a rectified sinewave of frequency $f_{samp} - 2f_{gen}$ (i.e. it ranges from $2kHz$ to $4kHz$) and has an amplitude modulation defined by $4\,f_{gen} - f_{samp}$(i.e. ranges from $0Hz$(for 2000Hz) to $4000Hz$(for 3000Hz)) The idea here is that all frequencies that are still too high to appear as their rectified sinwave is still above Nyquist range($2000Hz \times 2 > Nyquist$) thus the main components of the signal are made from the "trailing" harmonics which are reflected due to sampling at 8000Hz (i.e. from the other components of $\mathcal{F}\left(\lvert\sin(\omega_o t)\rvert_{decoupled}\right) \propto \sum_{n=1,2\ldots}^{\infty} \frac{\delta\left(2\omega_o - \frac{\omega}{n}\right) + \delta\left(\frac{\omega}{n} - 2\omega_o\right)}{n^2-1}$)
2000-1000Hz	• Output is a rectified sinewave of frequency $2f_{gen}$ (i.e. it ranges from $2kHz$ to $4kHz$) and has an amplitude modulation of $f_{samp} - 4f_{gen}$(i.e. ranges from $0Hz$(for 2000Hz) to $4000Hz$(for 1000Hz)) The reason why this happens is explained earlier and is best understood if looking at this problem in the frequency domain.
1000-0Hz	• At these frequencies output is simple a $2 \times f_{gen}$ decoupled rectified sinewave At these frequencies the harmonics of the rectified sinewave are mainly under 4000Hz so do not suffer from any sampling effects as most harmonic components did not reappear at a lower frequency due to sampling. This is because each subsequent component after the first one has its amplitude decrease relative to $\frac{1}{n^2-1}$.
Low frequencies	Distortion occurs due to amplitude and phase shift from high pass filter.

We can also place these observations on a line of working frequencies to give:


Input (sinewave) frequency
0Hz
1000Hz
2000Hz
3000H
4000H
Decoupled rectified sinewave with frequencies varying from 0 to 2000Hz
Decoupled rectified sinewave with amplitude modulation and frequency ranging from 2000 to 4000Hz
Decoupled rectified sinewave with frequencies varying from 0 to 2000Hz
0Hz
4000Hz
0Hz
Output (rectified sinewave) frequency

We can notice from the above figure that this is a many to one system, meaning that from the output we cannot predict what the input sinewave was. Indeed for inputs from 0-8000Hz we have 4 input sinewave frequencies which will lead to the same output.

Code Explanation

Exercise 1

This code worked on the basic principle that the input can be read from the mono_read_16Bit(); function. From that the code checked whether or not the sample was negative. In the case that it is the code inverted the value by multiplying it by -1. A variable called rectify was added to enable or disable this inverting process during runtime for debugging and research purposes.

Finally the rectified sample was written to the output using the `mono_write_16Bit(var);` function.

```
/********************************** ISR_AIC ******************************/
void ISR_AIC(){
        var = mono_read_16Bit(); // read input using read function
        if (rectify&&(var < 0)){var = -1*var;} // rectify the input and obey rectify
        mono_write_16Bit(var); // write to output
}
```

The function ISR_AIC() gets called 8000 times a second by the setting
0x008d, /* 8 SAMPLERATE Sample rate control 8 KHZ */\
The program Signal Generator was used to generate an input sinewave to then go on and test the code.

Exercise 2

The second exercise consisted of recycling the code from the previous lab.

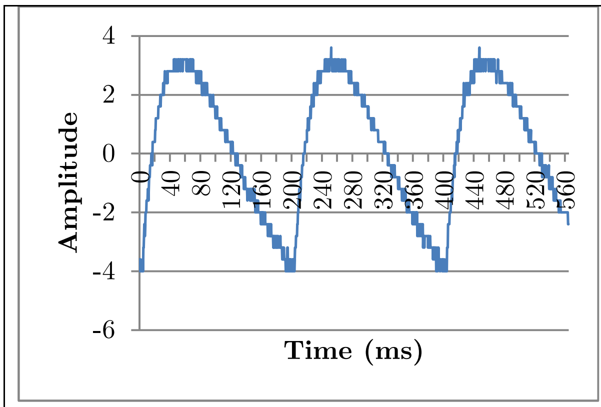
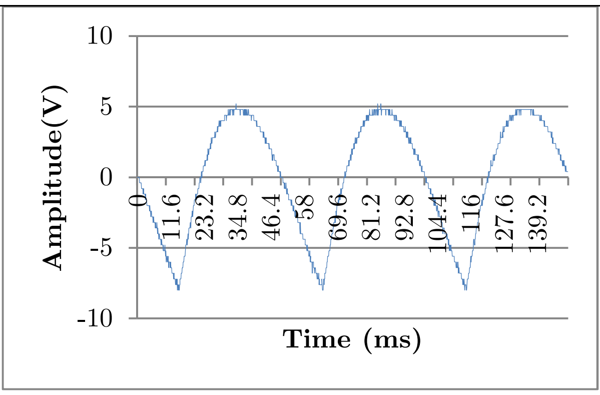
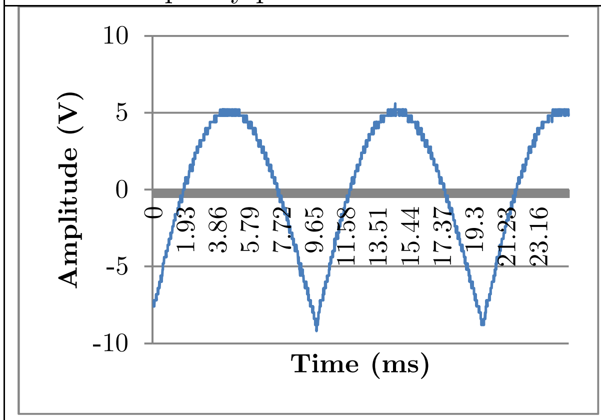
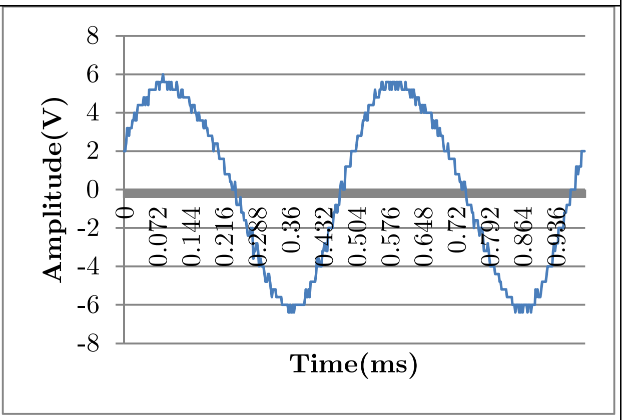
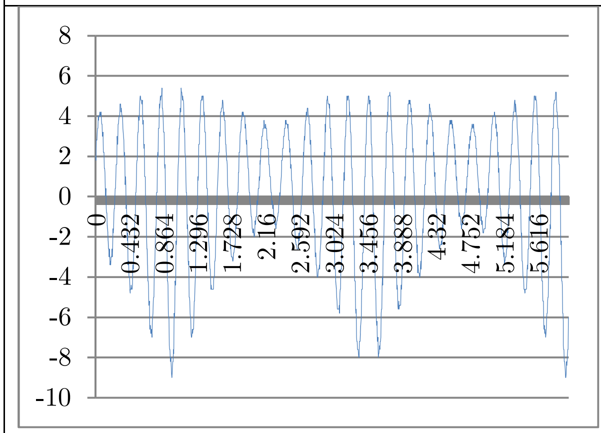
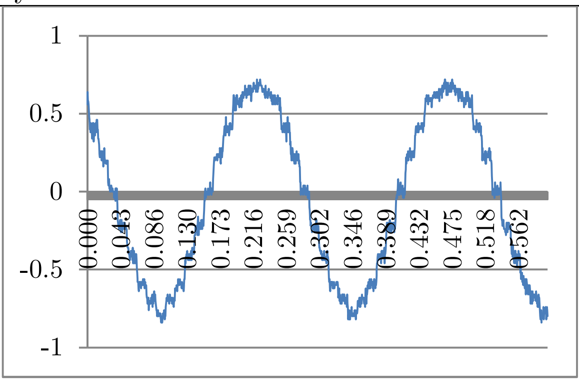
Firstly sinewave values were stored into a table of length 256 by using the following formula: $cell[i] = \sin\left(2\pi \times \frac{i}{256}\right)$. As in the previous lab the indexing variable was stored as a float to avoid any loss of data from storing as an integer.

Then we have to access each sample in the table which is done in the same way as the previous lab and we also increase the index in a similar way that avoids loss of data by treating everything as float. We also ensure that loop handling is done correctly.

The sample is then rectified by multiplying by -1 if it is originally negative and the penultimate step is to multiply this by the gain which is set to $2^{15} - 1 = \frac{2^{16}}{2} - 1 = 32767$as this takes full advantage of the 16bit allocated to the sample variable (our samples vary between 0 and 1, when rectified).

```
void ISR_AIC(){
        sample = table[(int)index]; // access correct sample by converting index to int
        index += (SINE_TABLE_SIZE)/(8000/sine_freq);//increase index by appropriate value for next
access - sampling freq is hard encoded for 8000
        //In the following case we are attempting to effectively do the following operation:
        //index %= SINE_TABLE_SIZE
        //however this is not possible with a float therefore we simply
        //substract the SINE_TABLE_SIZE when the index is over this value.
        if (index >= SINE_TABLE_SIZE)
        {
                index -= SINE_TABLE_SIZE;
        }
        if (sample <= 0){sample = -sample;} // rectify wave, could also have been done during
sinetable generation but the instructions implied rectification had to be done in the interrupt
function
        sample *=gain; //multiply by gain to produce visible output
        mono_write_16Bit(sample); //write to output
}
```

Scope traces

 Data collected for a 2.5Hz sinewave showing the low frequency phase distortion.	 Data collected for a 20Hz sinewave, showing that all is working as expected.
 Data collected for a 100Hz sinewave and 3900, which are both equivalent.	 Data collected for a 1000Hz sinewave, which could also have also been collected for a 300Hz sinewave from the reflection properties of this system.
 Data collected for a 1300Hz sinewave and 2700Hz. Here we can see the amplitude modulation in full effect.	 Data collected for a 2000Hz sinewave, which is essentially a 4000Hz sinewave. This is because the only frequencies to be conserved are at 4000Hz, as the next harmonic is simply reflected at 0Hz (due to 8k sampling) which is removed from the DC filter.

Source code

Exercise 1 code

```
/*************************************************************************************
                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>
int var; // Variable read
int rectify = 1; // used to activate and wave rectification on the fly
/******************************** 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***********************/
/********************************** ISR_AIC *******************************/
void ISR_AIC(){
        var = mono_read_16Bit(); // read input using read function
        if (rectify&&(var < 0)){var = -1*var;} // rectify the input and obey rectify
        mono_write_16Bit(var); // write to output
}
/***************************************************************************/
```

Exercise 2 code

```
/*************************************************************************************
                               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>
#define PI 3.141592653 // define PI for sine table generation
#define SINE_TABLE_SIZE 256 // define size

/******************************* 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;
// Holds the value of the current sample
float sample; // sample to output
float gain = -32767.0; // gain negative to invert signal and set to 2^16/2-1=2^15-1 (since a signed 16
bit output and we want to make the most out of it)
float sine_freq = 1000.0; // frequency of sinewave to generate
//define sinetable
float table[SINE_TABLE_SIZE];
//index
float index = 0; // index defined as float to avoid quantisation error
 /******************************* Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
void sine_init();
void ISR_AIC(void);
/********************************** Main routine ************************************/
```

```
void main(){

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

  /* initialize hardware interrupts */
  init_HWI();

   // initialize sine table
  sine_init();
  /* loop indefinitely, waiting for interrupts */
  while(1)
  {};

}

/***************************************************************************/
// sine init function
void sine_init(void)
{
        int i; // index variable

       // loops through table values setting each cell to $cell[i] = \sin\left(2\pi \times \frac{i}{256}\right)$
        for (i = 0; i < SINE_TABLE_SIZE; i++)
        {
                table[i] = sin(2*i*PI/(SINE_TABLE_SIZE));
        }
        return;
}
/*********************************** 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

}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
/*********************************** ISR_AIC ********************************/
void ISR_AIC(){
        sample = table[(int)index]; // access correct sample by converting index to int
        index += (SINE_TABLE_SIZE)/(8000/sine_freq);//increase index by appropriate value for next
access
        //In the following case we are attempting to effectively do the following operation:
        //index %= SINE_TABLE_SIZE
        //however this is not possible with a float therefore we simply
        //substract the SINE_TABLE_SIZE when the index is over this value.
        if (index >= SINE_TABLE_SIZE)
        {
                index -= SINE_TABLE_SIZE;
        }
        if (sample <= 0){sample = -sample;} // rectify wave, could also have been done during
sinetable generation but the instructions implied rectification had to be done in the ISR
        sample *=gain; //multiply by gain to produce visible output
        mono_write_16Bit(sample); //write to output
}
```

Works Cited

Falstad, P. (2008). *Fourier Series Examples: Full-Wave Rectification*. Retrieved 2013, from Paul Falstad: http://www.falstad.com/fourier/e-fullrect.html