RTDSP Lab 3

Yong Wen Chua (ywc110)

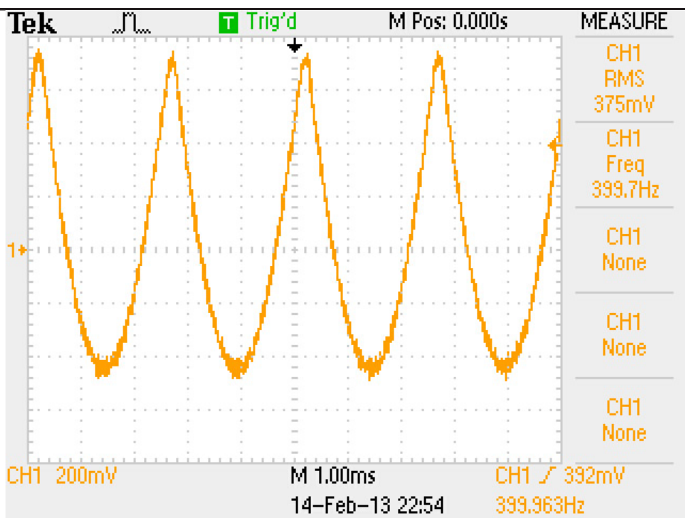
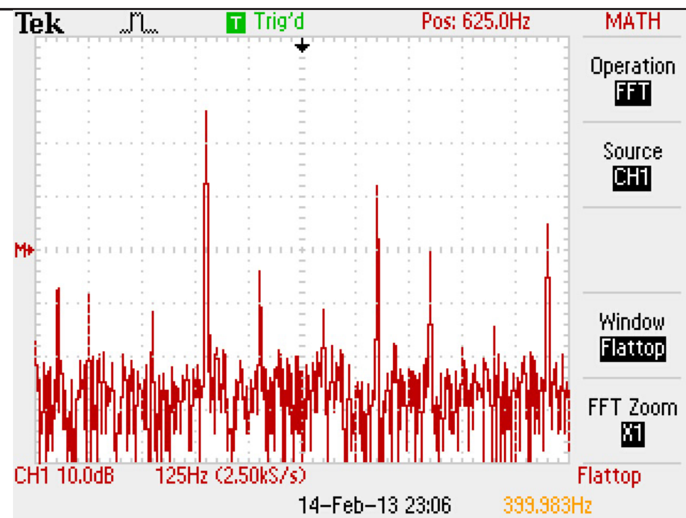
1 Exercise 1 - Input Sine Wave Rectification

1.1 Reason for 0 V Centre

The capacitor at the line out port blocks the DC component of the output signal. This is because the impedance of the capacitor is given by $\frac{1}{j\omega C}$. When the frequency $\omega = 0$, the capacitor has infinite impedance. Thus, it acts to block the DC component of the output signal.

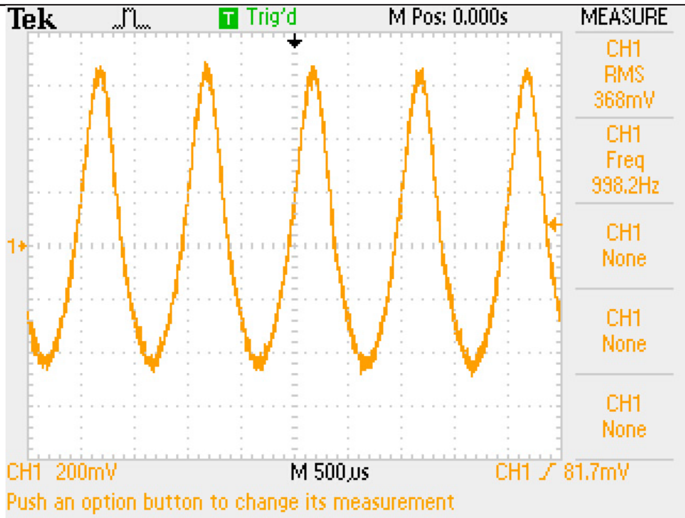
1.2 Oscilloscope Traces

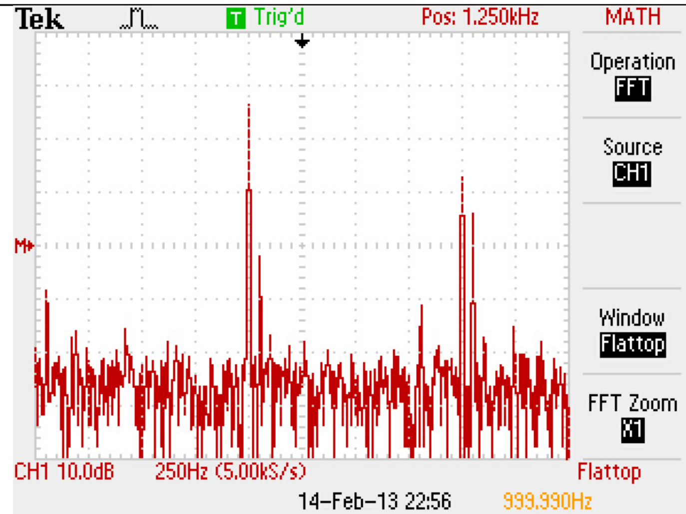
The traces for the various input frequencies and their corresponding FFT is given in the table below. This table will be used to answer the second question regarding the output frequency.

	Trace	FFT
200 Hz		

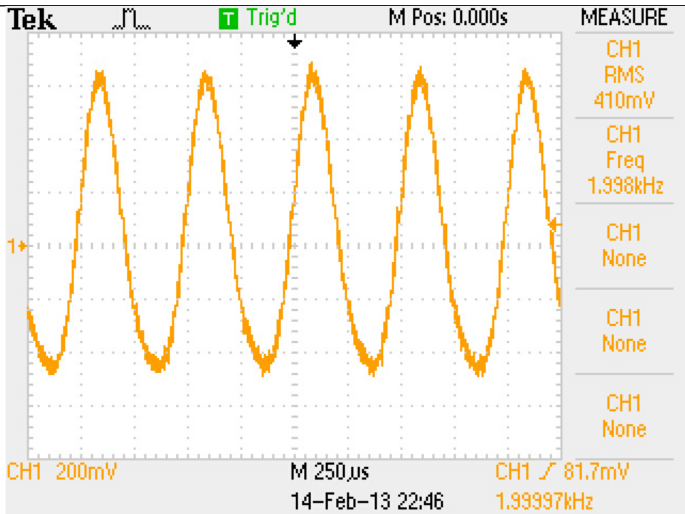
	Trace	FFT

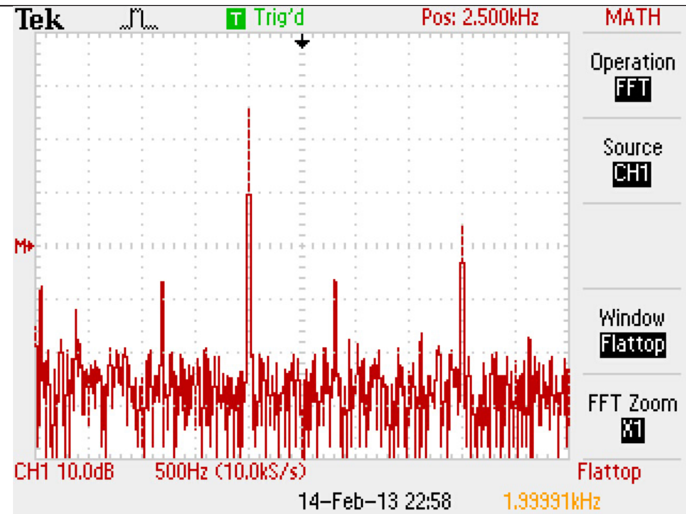
500 Hz



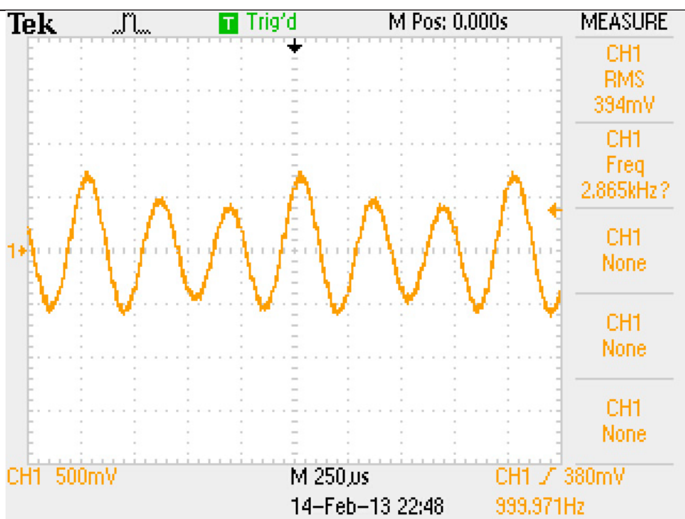


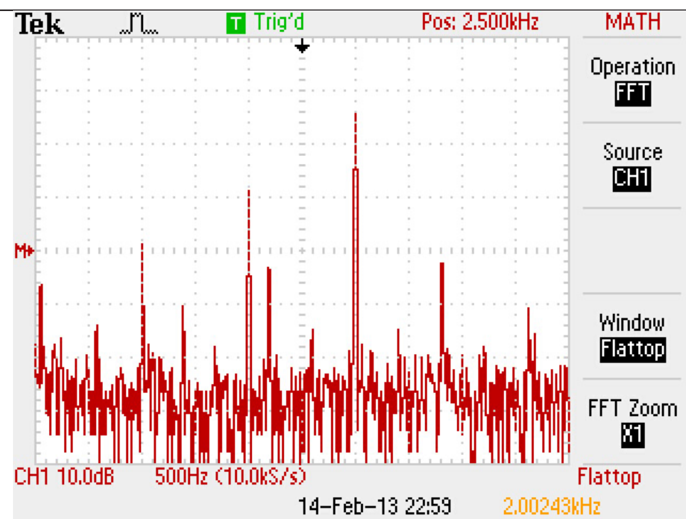
1000 Hz

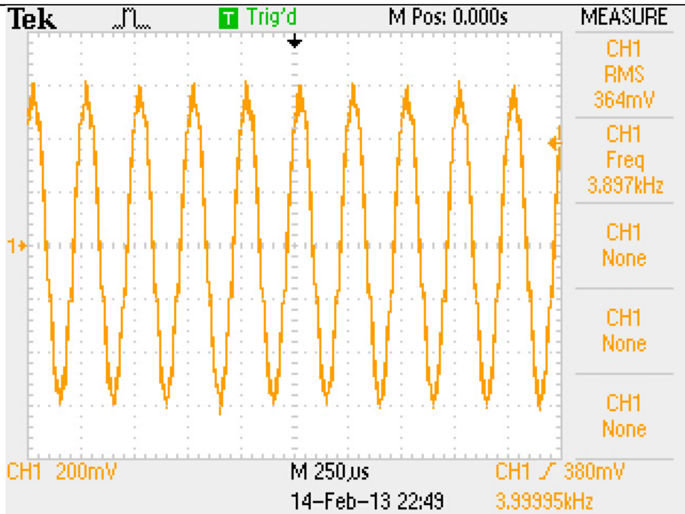
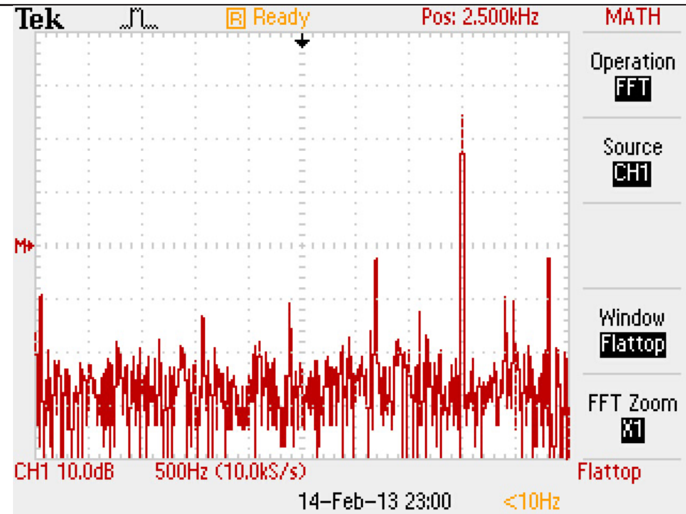
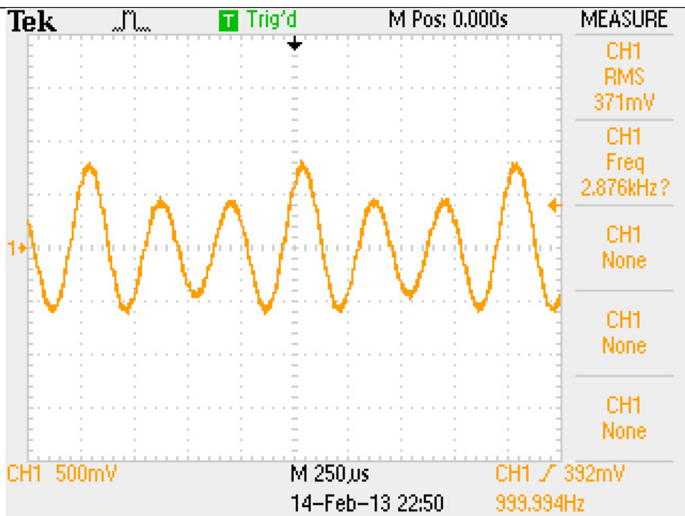
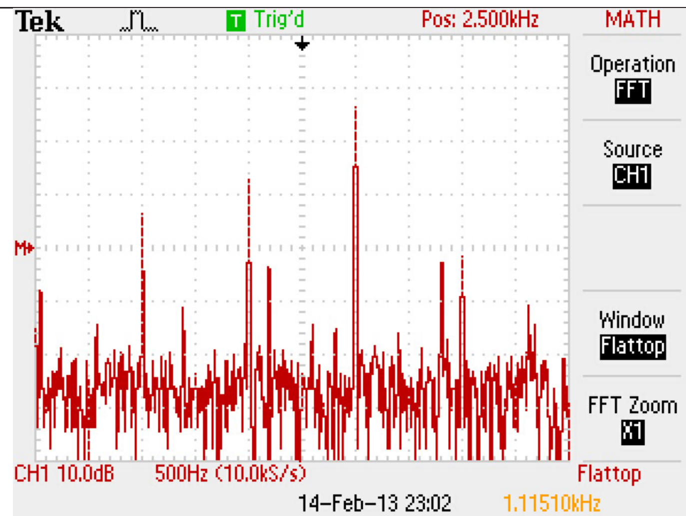
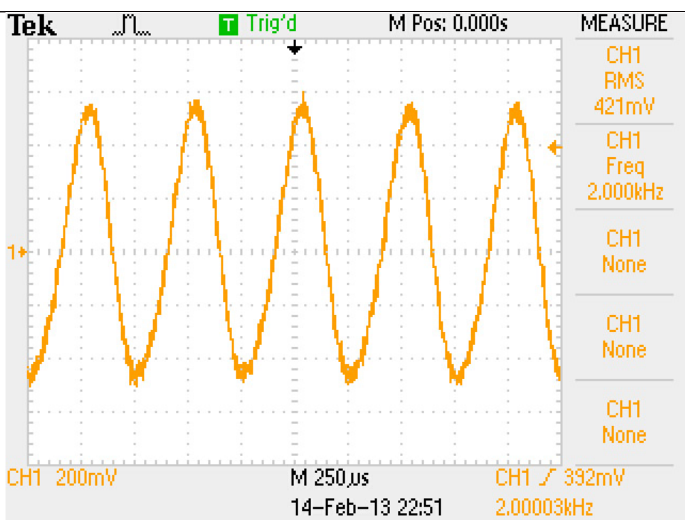
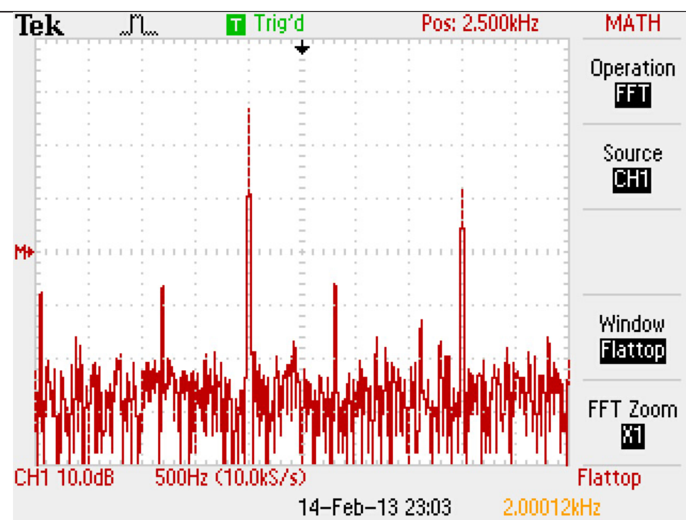


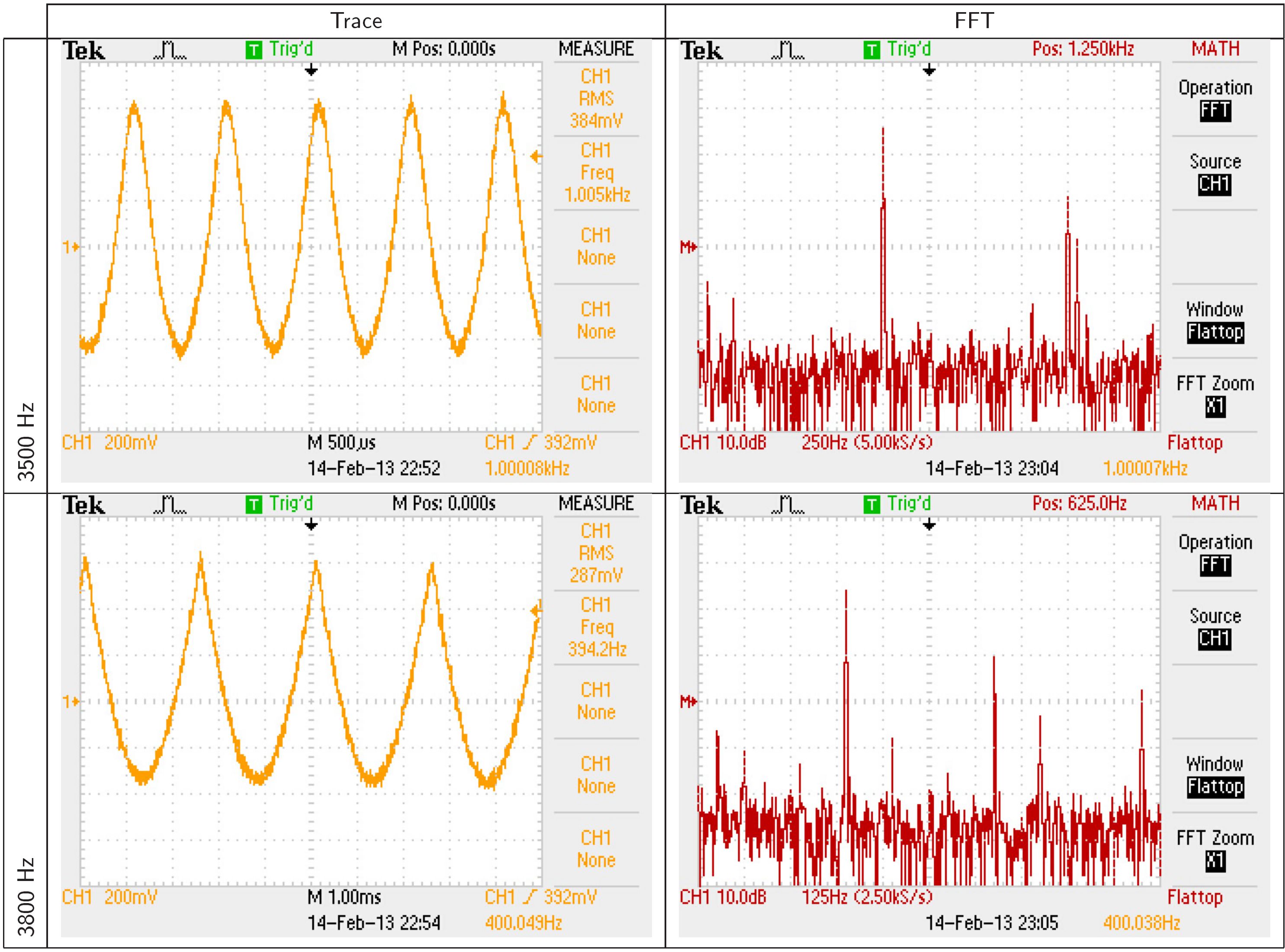


1500 Hz





	Trace	FFT
2000 Hz		
2500 Hz		
3000 Hz		



1.3 On Frequency

When a signal is input into the system, the output effectively has twice the frequency of the input due to the rectification done on the input signal. Thus, subject to the Nyquist Sampling Frequency limitation, the output cannot be more than 4 KHz (for a sampling frequency of 8KHz), and this translates to having an input frequency that, theoretically, can have a maximum of 2 KHz (which will give an output of 4 KHz) before aliasing kicks in.

When the input frequency goes beyond 2 KHz, aliasing takes place and folding can be observed in the output. This is evident if one compares the trace and FFT output (in section 1.2) of input frequencies of 200 Hz and 3800 Hz. It can be seen that these two input frequencies both give an output frequency of 400 Hz and have the same peak at 400 Hz in their FFT. This is due to folding being observed as a result in aliasing. Folding is the "wrapping" around of the frequency output due to aliasing. When a signal is sampled, copies of its frequency spectrum is made in the frequency domain. If the signal has frequency content higher than half that of the sampling frequency, these copies of the frequency content overlaps. The overlapped frequency content will cause an apparently lower frequency signal to be produced. This is the folding effect of aliasing. Let f_a be the apparent frequency of the sampled frequency, f be the (supposed) frequency of the sinusoid, and f_s be the sampling frequency. Then the folding effect is given by

$$|f_a| = |f - nf_s| \tag{1}$$

where $|f_a| \leqslant \frac{f_s}{2}$ and $n \in \mathbb{Z}$.

It can be observed that when the input frequency is at 3800 Hz, the output frequency, f, should be at 7600 Hz. Then according to the equation (1), $n = 1$ and $f_a = 400$, which is what can be observed.

A similar observation can be seen in the output traces for inputs of 2.5 KHz, 3 KHz, and 3.5 KHz. They give rise to apparent frequencies of 3 KHz, 2 KHz, and 1 KHz respectively. According to equation (1), they would all have $n = 1$.

Amplitude modulation can also be observed. This is due to the rectification of the sine wave, giving rise to additional harmonics. Take, for example, the FFT of the input frequency of 1.5 KHz. Around the main peak at 3 KHz, there are peaks at 2.25 KHz and 3.75 KHz, both of which are equidistant to the main peak at 3 KHz. This gives rise the the amplitude modulation, as can be observed from the trace. The imperfect anti-aliasing filter also limits the output frequency to around 3.6 KHz to 3.8 KHz. because the "copies" of the frequency content due to sampling above 4 KHz will not be filtered out properly, resulting in amplitude modulation. This can be seen in the FFT for the input frequency of 2 KHz, where a peak at 4.75 KHz was not filtered, resulting in amplitude modulation being observed. (The amplitude modulation was not captured in the trace due to the high frequency modulation.)

1.4 Code Operation

Rectification is done every time a sample is sent to the codec. The function to service the interrupt is given below:

```
void ISR_AIC(void){
    int sample = mono_read_16Bit(); // read
    sample = abs(sample); // rectify
    mono_write_16Bit((Int16) sample); // write
}
```

The code first reads the sample from the input, calculates its absolute value and then write it out to the output port. Note the cast to `Int16` as the output port is operating with 16 bits of data as per the configuration.

2 Exercise 2 - Interrupt generated sine wave

2.1 Code Operation

The code to generate the sine wave is similar to the one used in Lab 2. Firstly, a lookup table is prepared during the initialisation stages to prepare 256 values for one quarter of a sine wave. The table is prepared using the following function (which is the same as Lab 2):

```
void sine_init(void){
  int i;
  for (i = 0; i < SINE_TABLE_SIZE; ++i)
        table[i] = sin(i * ( (2*PI) /(RES_MULTIPLIER*SINE_TABLE_SIZE)));
}
```

The macro `RES_MULTIPLIER` is set to a value of '4'. The service interrupt routine as described in section 1.4 is then replaced with essentially the same function as in Lab 2.

The routine, by keeping track of the "phase" of the current wave, will output the value to allow for the generation of the sine wave in the appropriate frequency. It keeps track of the "phase" of the wave using a `static` variable `index`. The static variables `prev_freq` and `prev_sample` are used top check for changes in those settings. If either are changed, the `index` is reset to zero, and the values described below will be recalculated.

The first value determines the number of samples necessary to generate the entire wave using the following line:

```
double cycleSampleCount = (double) sampling_freq / (double) sine_freq;
```

The number of entries in the sine table to skip each time a sample is required is then calculated using the line, along with an increment of the `index`:

```
step = (double) (SINE_TABLE_SIZE*RES_MULTIPLIER)/(double) cycleSampleCount;
//... other code ...
index += step;
```

To ensure that we do not exceed the number of entries in the table and cause a segmentation fault, the following line will reset the index with the necessary offset for the next cycle to ensure a smoother wave and allows the generation of "odd" frequencies:

```
// Check that index is in range
if ((int) index >= SINE_TABLE_SIZE*RES_MULTIPLIER)
  index -= (SINE_TABLE_SIZE*RES_MULTIPLIER);
```

The table is then retrieved from the table, mulitplied by an appropriate gain and sent to the output.

```
sample = sine_value((int) round(index)); // get value from table
output = (Int16) fabs(sample*gain); // ouput saved to a variable for ease of debugging
mono_write_16Bit(output); // write to port
```

The `sine_value` function works by first determing the quadrant of a sine wave in which the index we want to retrieve is at, and also calculates the "progress" in that specific quadrant:

```
int quadrant = index/SINE_TABLE_SIZE;  // the quadrant in which the cycle is in
int modulo = index % SINE_TABLE_SIZE;   // the modulo
```

Then, according to the quadrant the index is in, the appropriate value is read from the table array and adjusted accordingly.

```
if (quadrant == 0)
  value = table[index];
else if (quadrant == 1)
  value =  table[SINE_TABLE_SIZE-modulo-1];
else if (quadrant == 2)
  value =  table[modulo]*-1;
else if (quadrant == 3)
  value =  table[SINE_TABLE_SIZE-modulo-1]*-1;
else
  value =  0;
```

2.2 Oscilloscope Traces

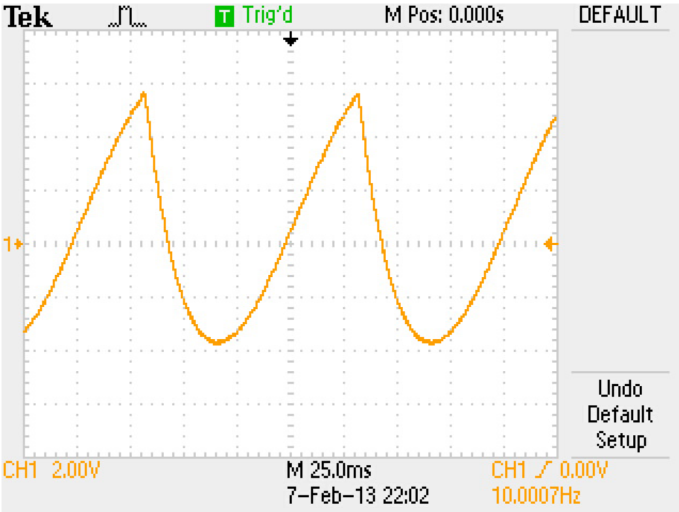


Figure 1: 5 Hz input sine wave rectified to 10 Hz

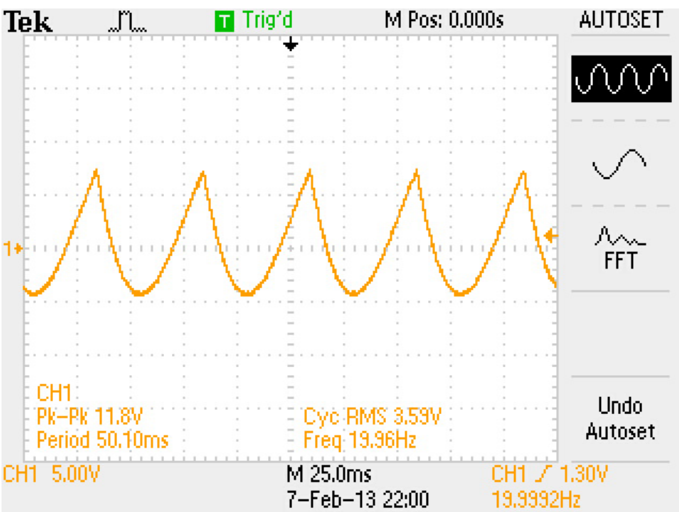


Figure 2: 10 Hz input sine wave rectified to 20 Hz.

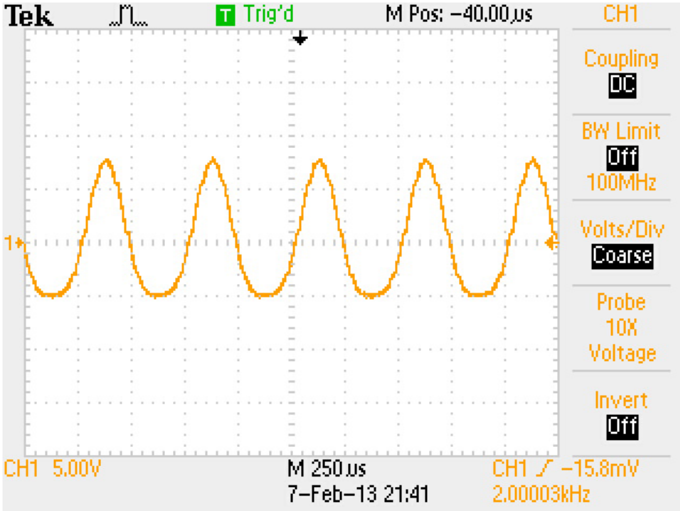


Figure 3: 1 KHz sine wave rectified to 2 KHz

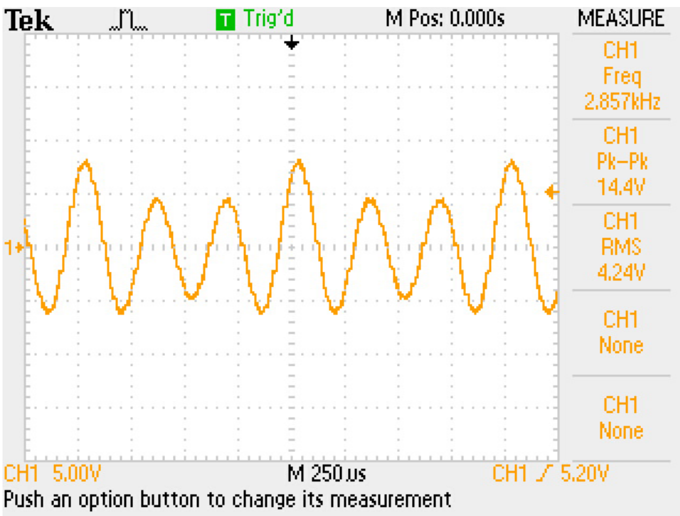


Figure 4: 1.5 KHz input rectified to 3 KHZ. Notice the varying amplitude due to amplitude modulating effects

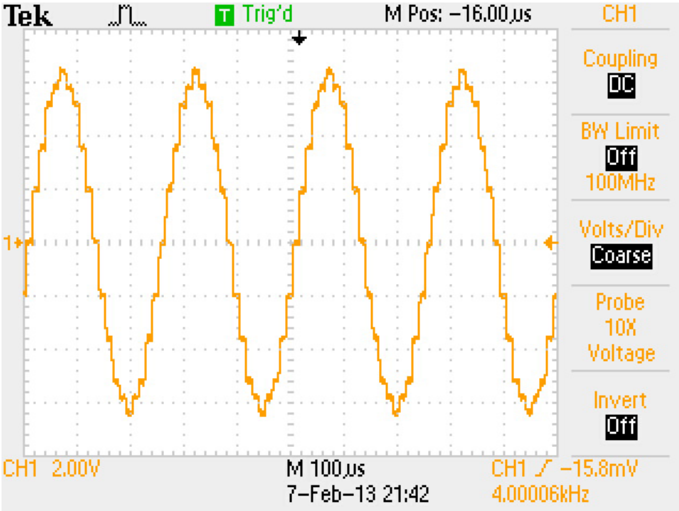


Figure 5: 2 KHz input rectified to 4 KHz. The Nyquist Frequency

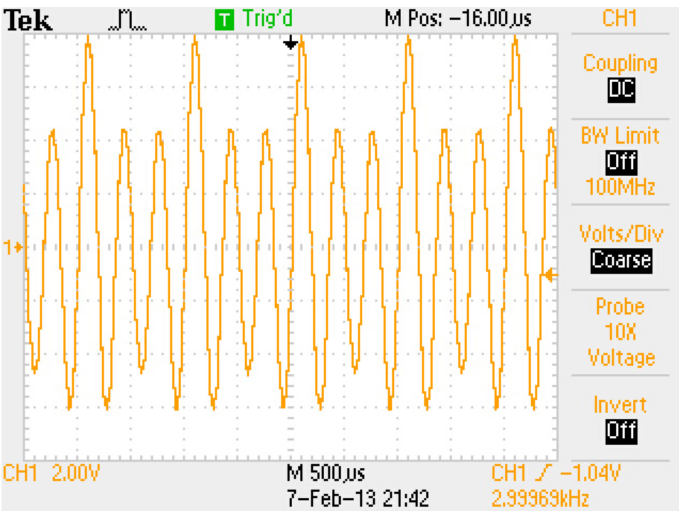


Figure 6: 2.5 KHz input that is supposed to give a 5 KHz. Due to aliasing giving rise to folding, the output is at 3 KHz.

3 Code Listing

3.1 Exercise 1

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

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

                        LAB 3: Interrupt I/O Exercise 1
 ************************************************************************************/

/***************************** Pre-processor statements ******************************/

#include <stdlib.h>
#include <stdio.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
```

```c
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){
    int sample = mono_read_16Bit(); // read
    sample = abs(sample); // rectify
    mono_write_16Bit((Int16) sample); // write
}
```

3.2 Exercise 2

```
/*************************************************************************************
             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 Exercise 2
 ************************************************************************************/
/**************************** Pre-processor statements ******************************/

#include <stdlib.h>
#include <stdio.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

// The number of entries in the sine lookup table
#define SINE_TABLE_SIZE 256

// Multiplier depending on whether INCREASE_RES is set
#define RES_MULTIPLIER 4

/******************************* 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         */
       /**********************************************************************/\
```

```c
    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;

// Gain - Less than 16 bit
Int16 gain = 30000;

/* 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;

/* The array to hold the values of the sine lookup table */
float table[SINE_TABLE_SIZE] = {0};

 /****************************** Function prototypes ******************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
void sine_init(void);
#ifdef INCREASE_RES
  float sine_value(int);
#endif
/********************************** Main routine **********************************/
void main(){

  // initialise sine table
  sine_init();

  // 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_XINT1,4);    // Maps an event to a physical interrupt
  IRQ_enable(IRQ_EVT_XINT1);     // Enables the event
  IRQ_globalEnable();       // Globally enables interrupts

}

/******************** Interrupt Service Routine - Generate Sine **********************/

void ISR_AIC(void){ // code is generally similar to Lab 2
  float sample; // the sample value
  Int16 output; // output value
    static double index = 0;  // Store the "progress" of the sine wave generation
    static double prev_freq = 0;  // Previous frequency
    static double prev_sample = 0;  // Previous sampling frequency
    // Based on sampling frequency and sine frequency, determine number of samples per cycle
    static double cycleSampleCount;
    // Determine the number of intervals to "skip" each time we proceed to the next stage of the sine wave generation
    static double step;

    if (prev_freq != sine_freq || prev_sample != sampling_freq){    // If frequency has changed, we should take note.
        index = 0;
        prev_freq = sine_freq;
        prev_sample = sampling_freq;
        cycleSampleCount = (double) sampling_freq / (double) sine_freq;
```

```c
        step = (double) (SINE_TABLE_SIZE*RES_MULTIPLIER)/(double) cycleSampleCount;
    }
    index += step; // "advance" to the next step

    // Check that index is in range
    if ((int) index >=  SINE_TABLE_SIZE*RES_MULTIPLIER)
            index -= (SINE_TABLE_SIZE*RES_MULTIPLIER);     // Reset with an offset so that we can try and generate more frequencies

    sample = sine_value((int) round(index)); // get value from table
    output = (Int16) fabs(sample*gain); // ouput saved to a variable for ease of debugging
    mono_write_16Bit(output); // write to port
}
/******************************** sine_init() ************************************/
void sine_init(void){
  int i;

  for (i = 0; i < SINE_TABLE_SIZE; ++i)
        table[i] = sin(i * ( (2*PI) /(RES_MULTIPLIER*SINE_TABLE_SIZE)));
}

float sine_value(int index){
    int quadrant = index/SINE_TABLE_SIZE;    // the quadrant in which the cycle is in
    int modulo = index % SINE_TABLE_SIZE;    // the modulo
    float value;
    if (quadrant == 0)
        value = table[index];
    else if (quadrant == 1)
        value =  table[SINE_TABLE_SIZE-modulo-1];
    else if (quadrant == 2)
        value =  table[modulo]*-1;
    else if (quadrant == 3)
        value =  table[SINE_TABLE_SIZE-modulo-1]*-1;
    else
        value =  0;

    return value;
}
```