Declaration: I/We confirm that this submission is my own work.

In it, I give references and citations whenever I refer to or use the published, or unpublished, work of others. I am aware that this course is bound by penalties as set out in the College examination offenses policy.

Signed:_____

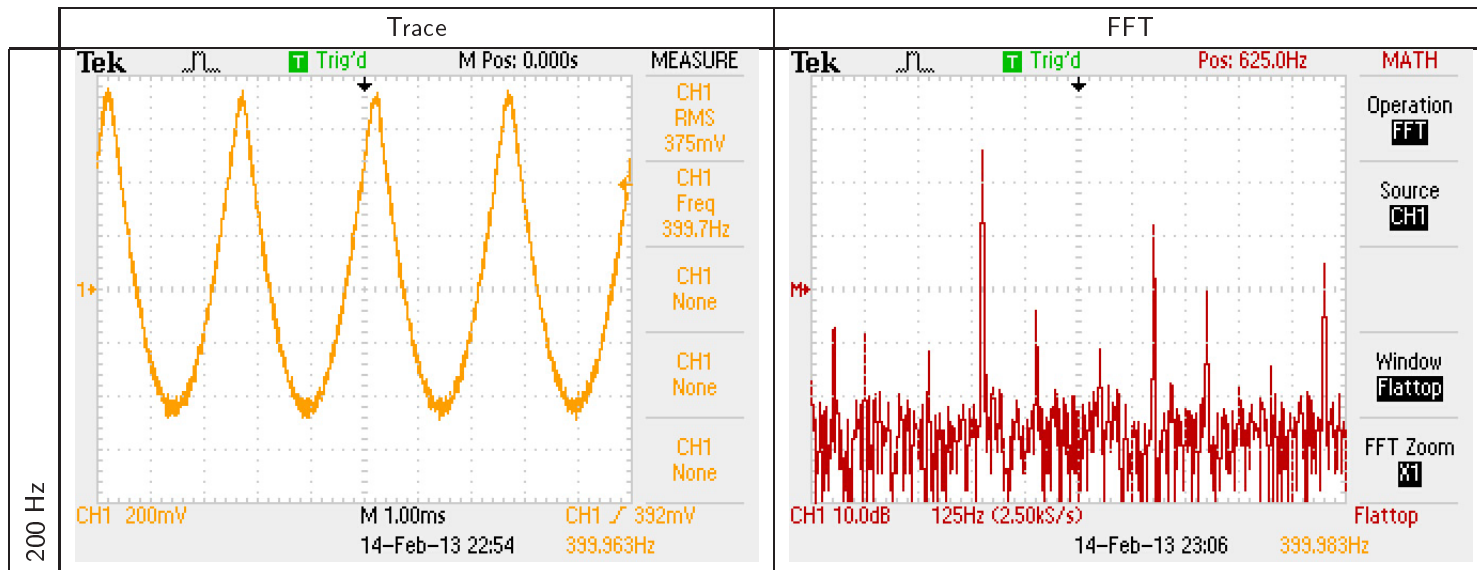# RTDSP Lab 3

Yong Wen Chua (ywc110)

1 Exercise 1 - Input Sine Wave Rectification
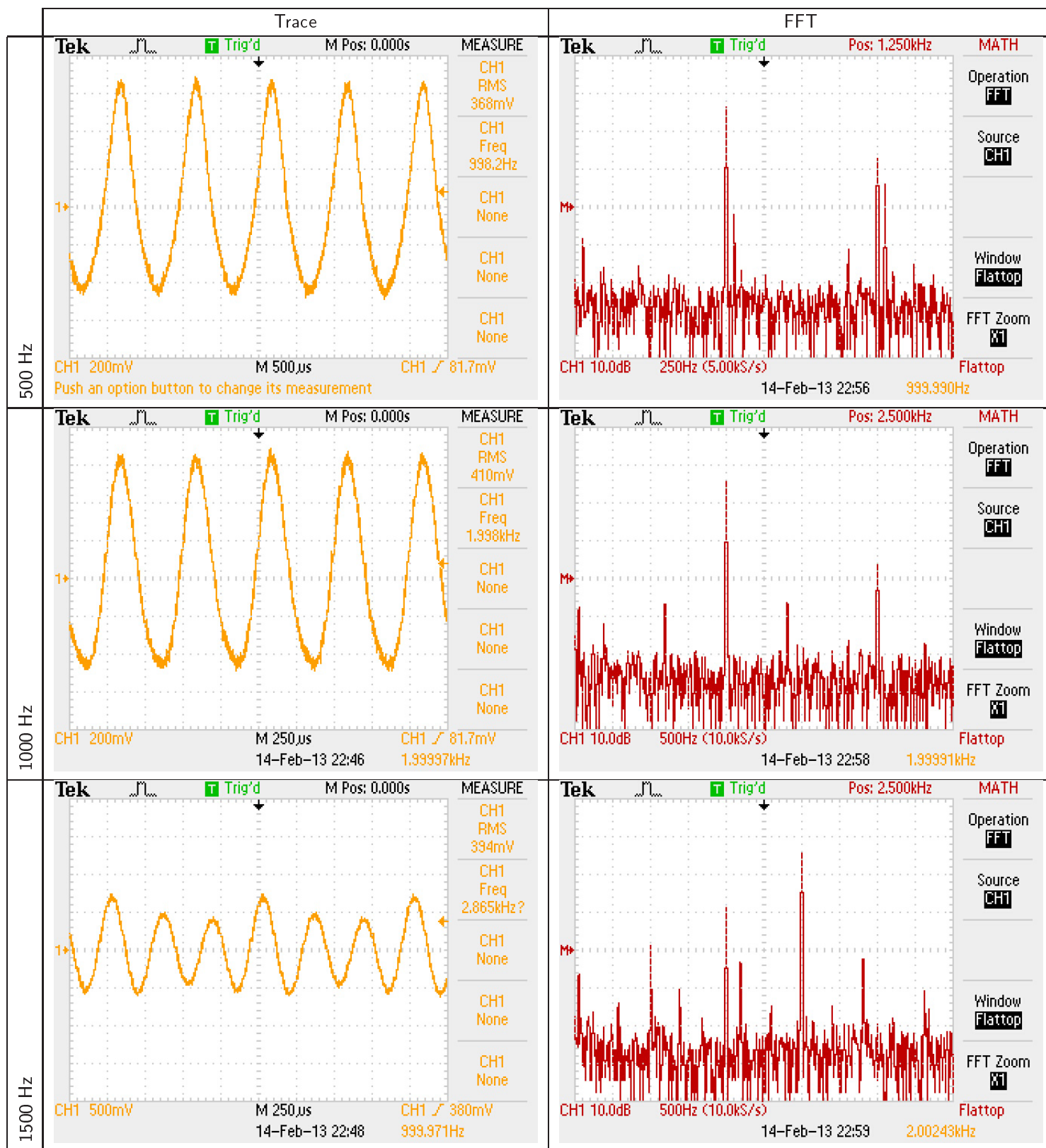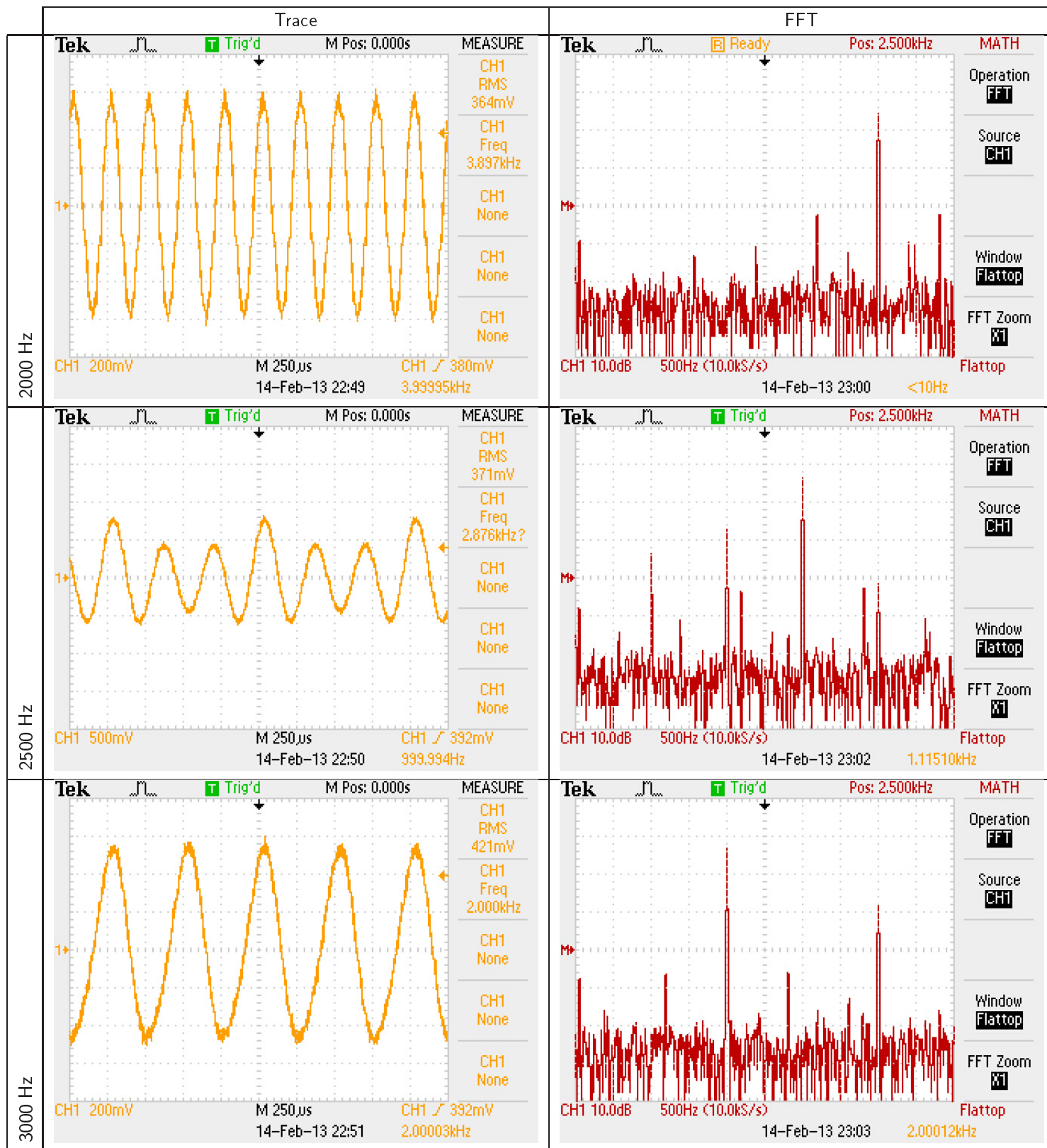
1.1 Reason for 0 V Centre

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.
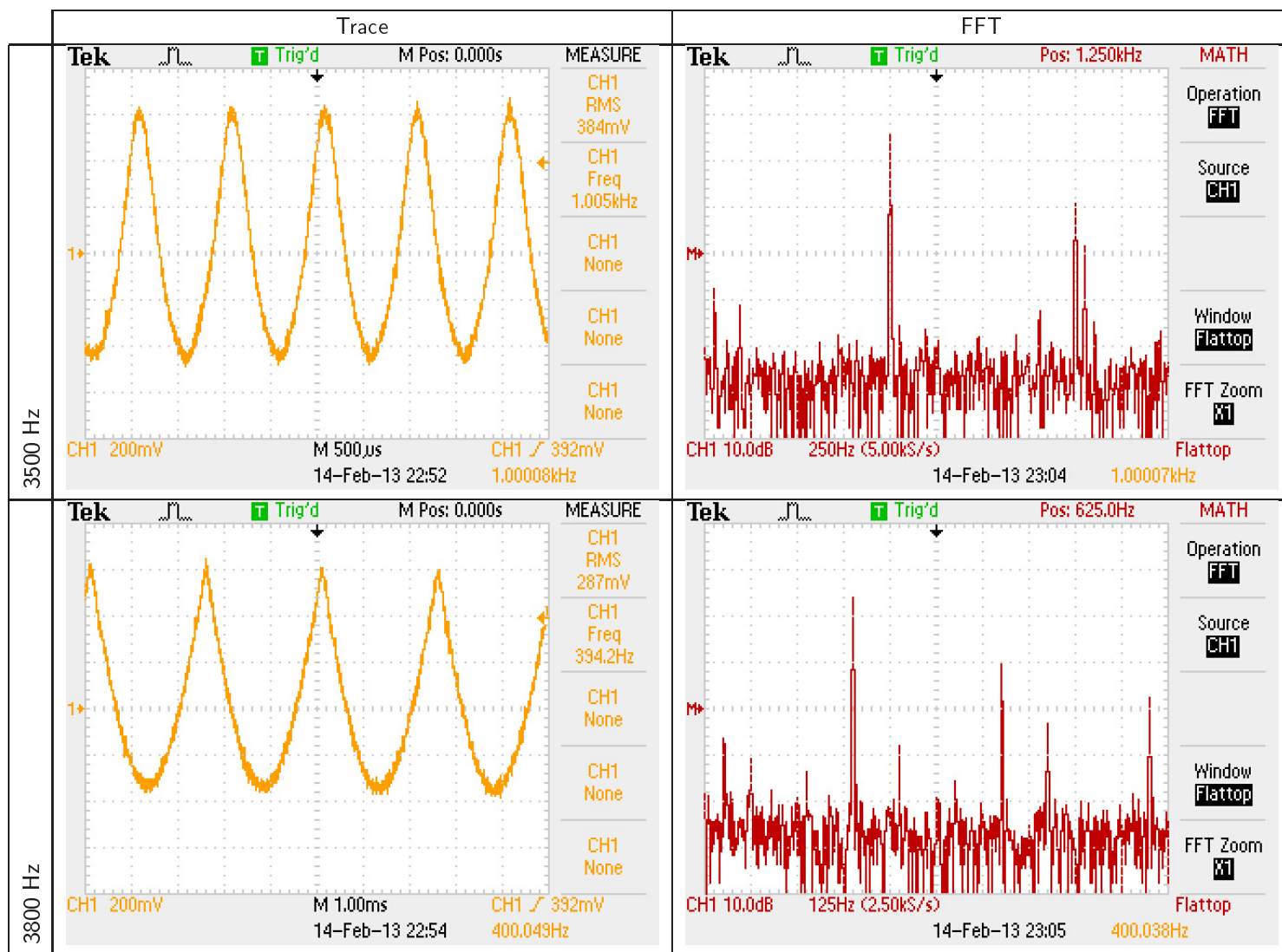
1.2 Oscilloscope Traces

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.

500 Hz

1000 Hz

1500 Hz

Trace	FFT

2000 Hz

Tek   T Trig'd   M Pos: 0.000s   MEASURE

CH1 RMS 364mV
CH1 Freq 3.897kHz
CH1 None
CH1 None
CH1 None

CH1 200mV   M 250μs   CH1 ∕ 380mV
14-Feb-13 22:49   3.99995kHz

Tek   R Ready   Pos: 2.500kHz   MATH

Operation FFT
Source CH1
Window Flattop
FFT Zoom X1

CH1 10.0dB   500Hz (10.0kS/s)   Flattop
14-Feb-13 23:00   <10Hz

2500 Hz

Tek   T Trig'd   M Pos: 0.000s   MEASURE

CH1 RMS 371mV
CH1 Freq 2.876kHz ?
CH1 None
CH1 None
CH1 None

CH1 500mV   M 250μs   CH1 ∕ 392mV
14-Feb-13 22:50   999.994Hz

Tek   T Trig'd   Pos: 2.500kHz   MATH

Operation FFT
Source CH1
Window Flattop
FFT Zoom X1

CH1 10.0dB   500Hz (10.0kS/s)   Flattop
14-Feb-13 23:02   1.11510kHz

3000 Hz

Tek   T Trig'd   M Pos: 0.000s   MEASURE

CH1 RMS 421mV
CH1 Freq 2.000kHz
CH1 None
CH1 None
CH1 None

CH1 200mV   M 250μs   CH1 ∕ 392mV
14-Feb-13 22:51   2.00003kHz

Tek   T Trig'd   Pos: 2.500kHz   MATH

Operation FFT
Source CH1
Window Flattop
FFT Zoom X1

CH1 10.0dB   500Hz (10.0kS/s)   Flattop
14-Feb-13 23:03   2.00012kHz

3

Trace	FFT

3500 Hz

Tek ⎍⎍ T Trig'd M Pos: 0.000s MEASURE
CH1 RMS 384mV
CH1 Freq 1.005kHz
CH1 None
CH1 None
CH1 None
CH1 200mV M 500μs CH1 ⌐ 392mV
14-Feb-13 22:52 1.00008kHz

Tek ⎍⎍ T Trig'd Pos: 1.250kHz MATH
Operation FFT
Source CH1
Window Flattop
FFT Zoom X1
CH1 10.0dB 250Hz (5.00kS/s) Flattop
14-Feb-13 23:04 1.00007kHz

3800 Hz

Tek ⎍⎍ T Trig'd M Pos: 0.000s MEASURE
CH1 RMS 287mV
CH1 Freq 394.2Hz
CH1 None
CH1 None
CH1 None
CH1 200mV M 1.00ms CH1 ⌐ 392mV
14-Feb-13 22:54 400.049Hz

Tek ⎍⎍ T Trig'd Pos: 625.0Hz MATH
Operation FFT
Source CH1
Window Flattop
FFT Zoom X1
CH1 10.0dB 125Hz (2.50kS/s) Flattop
14-Feb-13 23:05 400.038Hz

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. This is because when a sine wave is rectified, its orignal frequency component is removed, and a DC component along with decreasing frequency components that are even multiples (i.e. two times, four times, etc.) of the original frequency component (see figure 1). This is observable from the FFT traces in in section 1.2.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
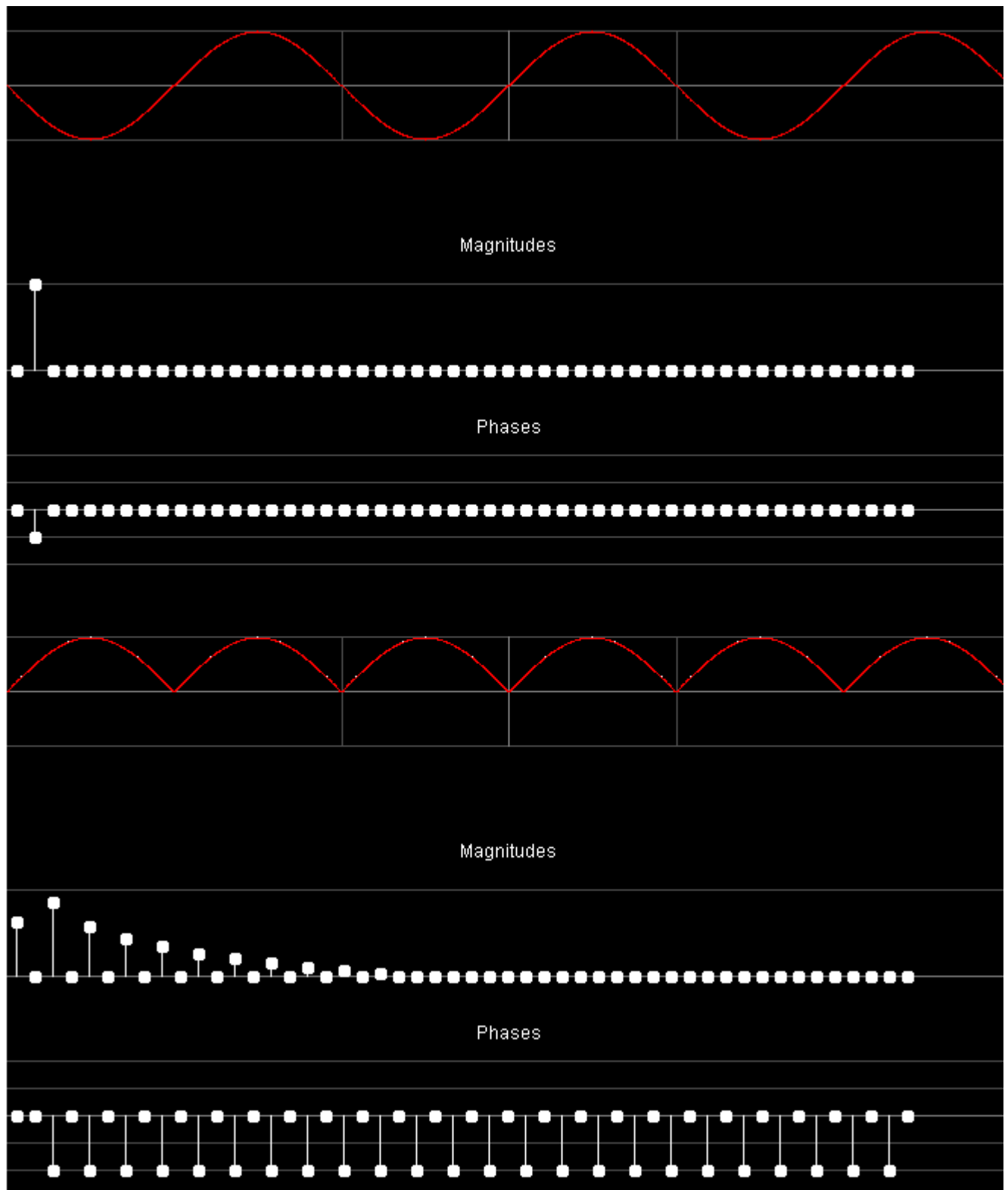
Figure 1: The Fourier transform of a sine wave (above) and its fully rectified version (below). Taken from an applet at http://www.falstad.com/fourier/e-fullrect.html

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, as described in the first paragraph. Take, for example, the FFT of the input frequency of 1.5 KHz. There are peaks at 1 KHz, due to the wrap around effect of the duplicated even harmonics that is supposed to be at 6 KHz (as described by equation (1)), and another at 2 KHz due to "a copy" made by the sampling process. Also, from the FFT traces, it can be seen that some of frequency components above 4 KHz are not properly filtered out, due to the imperfect anti-aliasing filter. This also contributes to the distortion of the output signal.

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:

```
1  void ISR_AIC(void){
2      int sample = mono_read_16Bit(); // read
3      sample = abs(sample); // rectify
4      mono_write_16Bit((Int16) sample); // write
5  }
```

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):

```
1  void sine_init(void){
2    int i;
3    for (i = 0; i < SINE_TABLE_SIZE; ++i)
4         table[i] = sin(i * ( (2*PI) /(RES_MULTIPLIER*SINE_TABLE_SIZE)));
5  }
```

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;
```
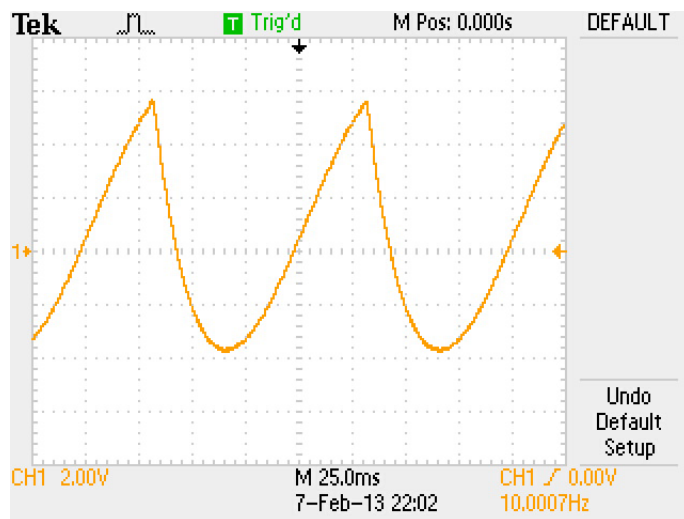
2.2 Oscilloscope Traces

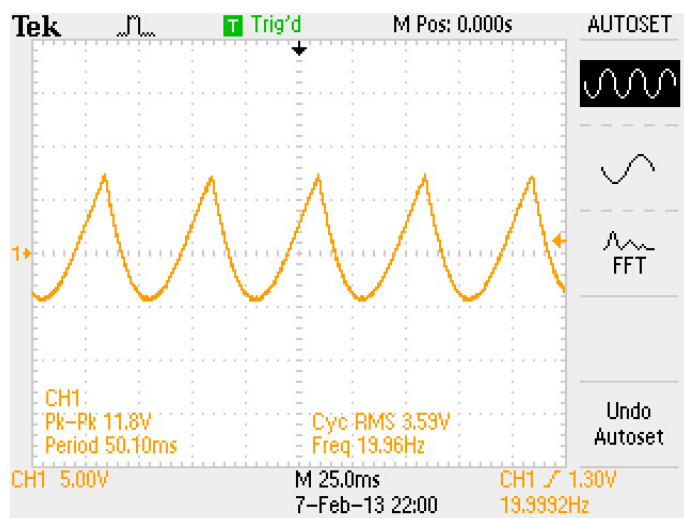Figure 2: 5 Hz input sine wave rectified to 10 Hz

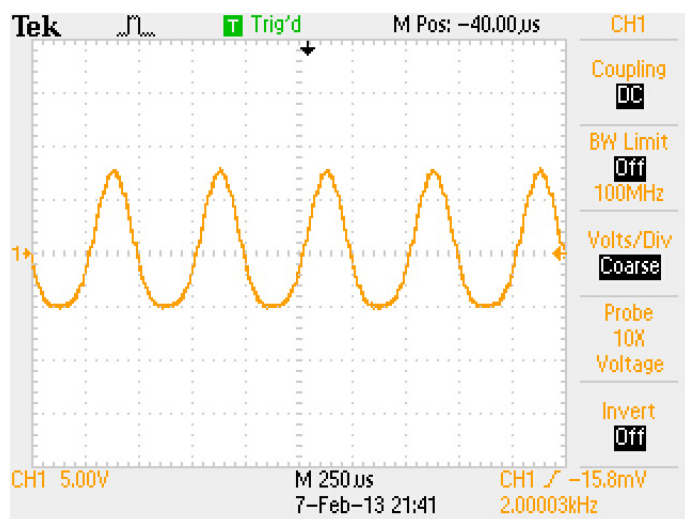Figure 3: 10 Hz input sine wave rectified to 20 Hz.

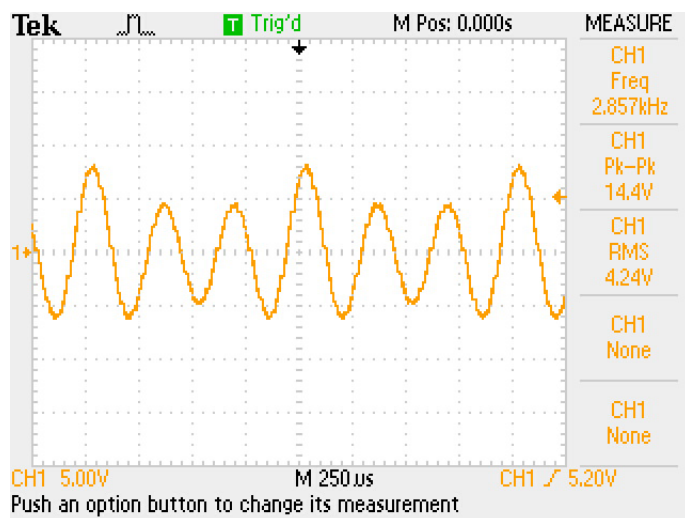Figure 4: 1 KHz sine wave rectified to 2 KHz

Figure 5: 1.5 KHz input rectified to 3 KHZ. Notice the varying amplitude due to amplitude modulating effects
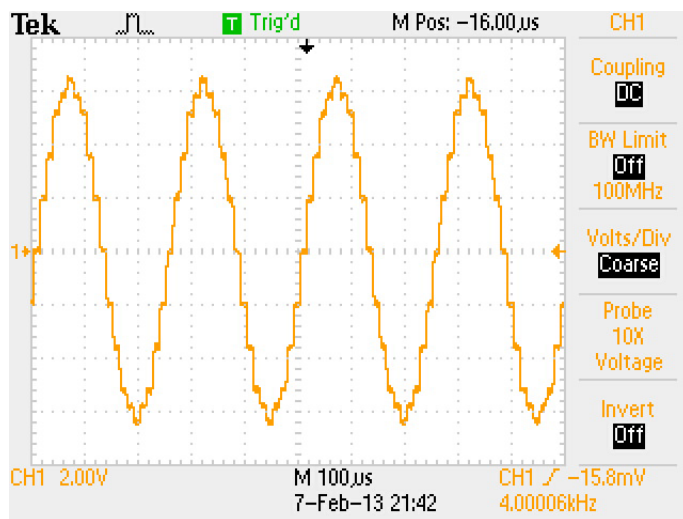
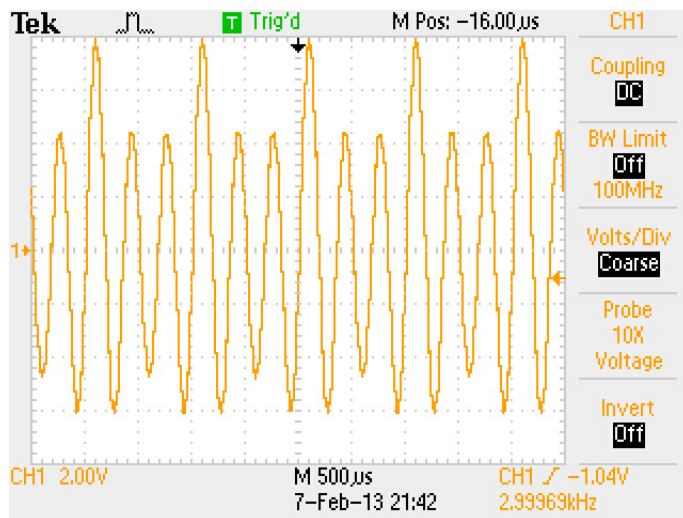Figure 6: 2 KHz input rectified to 4 KHz. The Nyquist Frequency

Figure 7: 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

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

```
53  DSK6713_AIC23_CodecHandle H_Codec;
54
55   /****************************** Function prototypes *******************************/
56  void init_hardware(void);
57  void init_HWI(void);
58  void ISR_AIC(void);
59  /******************************** Main routine ************************************/
60  void main(){
61
62
63    // initialize board and the audio port
64    init_hardware();
65
66    /* initialize hardware interrupts */
67    init_HWI();
68
69    /* loop indefinitely, waiting for interrupts */
70    while(1)
71    {};
72
73  }
74
75  /******************************** init_hardware() ********************************/
76  void init_hardware()
77  {
78      // Initialize the board support library, must be called first
79      DSK6713_init();
80
81      // Start the AIC23 codec using the settings defined above in config
82      H_Codec = DSK6713_AIC23_openCodec(0, &Config);
83
84    /* Function below sets the number of bits in word used by MSBSP (serial port) for
85     receives from AIC23 (audio port). We are using a 32 bit packet containing two
86     16 bit numbers hence 32BIT is set for  receive */
87    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
88
89    /* Configures interrupt to activate on each consecutive available 32 bits
90     from Audio port hence an interrupt is generated for each L & R sample pair */
91    MCBSP_FSETS(SPCR1, RINTM, FRM);
92
93    /* These commands do the same thing as above but applied to data transfers to
94     the audio port */
95    MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
96    MCBSP_FSETS(SPCR1, XINTM, FRM);
97
98
99  }
100
101  /******************************** init_HWI() ************************************/
102  void init_HWI(void)
103  {
104    IRQ_globalDisable();      // Globally disables interrupts
105    IRQ_nmiEnable();          // Enables the NMI interrupt (used by the debugger)
106    IRQ_map(IRQ_EVT_RINT1,4);   // Maps an event to a physical interrupt
107    IRQ_enable(IRQ_EVT_RINT1);    // Enables the event
108    IRQ_globalEnable();        // Globally enables interrupts
109
```

```
110  }
111
112  /******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE*********************/
113
114  void ISR_AIC(void){
115      int sample = mono_read_16Bit(); // read
116      sample = abs(sample); // rectify
117      mono_write_16Bit((Int16) sample); // write
118  }
```

3.2   Exercise 2

```
1   /**************************************************************************************
2                   DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3                             IMPERIAL COLLEGE LONDON
4
5                     EE 3.19: Real Time Digital Signal Processing
6                         Dr Paul Mitcheson and Daniel Harvey
7
8                         LAB 3: Interrupt I/O Exercise 2
9    *************************************************************************************/
10  /*************************** Pre-processor statements ******************************/
11
12  #include <stdlib.h>
13  #include <stdio.h>
14  // Included so program can make use of DSP/BIOS configuration tool.
15  #include "dsp_bios_cfg.h"
16
17  /* The file dsk6713.h must be included in every program that uses the BSL. This
18      example also includes dsk6713_aic23.h because it uses the
19      AIC23 codec module (audio interface). */
20  #include "dsk6713.h"
21  #include "dsk6713_aic23.h"
22
23  // math library (trig functions)
24  #include <math.h>
25
26  // Some functions to help with writing/reading the audio ports when using interrupts.
27  #include <helper_functions_ISR.h>
28
29  // PI defined here for use in your code
30  #define PI 3.141592653589793
31
32  // The number of entries in the sine lookup table
33  #define SINE_TABLE_SIZE 256
34
35  // Multiplier depending on whether INCREASE_RES is set
36  #define RES_MULTIPLIER 4
37
38  /****************************** Global declarations *******************************/
39
40  /* Audio port configuration settings: these values set registers in the AIC23 audio
41      interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
42  DSK6713_AIC23_Config Config = { \
43          /*******************************************************************\
44          /*  REGISTER              FUNCTION                 SETTINGS         */
45          /*******************************************************************/\
```

```
46      0x0017,   /* 0 LEFTINVOL    Left line input channel volume  0dB              */\
47      0x0017,   /* 1 RIGHTINVOL Right line input channel volume 0dB                */\
48      0x01f9,   /* 2 LEFTHPVOL    Left channel headphone volume   0dB              */\
49      0x01f9,   /* 3 RIGHTHPVOL Right channel headphone volume  0dB                */\
50      0x0011,   /* 4 ANAPATH      Analog audio path control        DAC on, Mic boost 20dB*/\
51      0x0000,   /* 5 DIGPATH      Digital audio path control       All Filters off  */\
52      0x0000,   /* 6 DPOWERDOWN Power down control                 All Hardware on   */\
53      0x0043,   /* 7 DIGIF        Digital audio interface format  16 bit            */\
54      0x008d,   /* 8 SAMPLERATE Sample rate control                8 KHZ            */\
55      0x0001    /* 9 DIGACT       Digital interface activation     On              */\
56          /*********************************************************************/
57  };
58
59
60  // Codec handle:- a variable used to identify audio interface
61  DSK6713_AIC23_CodecHandle H_Codec;
62
63  /* Sampling frequency in HZ. Must only be set to 8000, 16000, 24000
64  32000, 44100 (CD standard), 48000 or 96000  */
65  int sampling_freq = 8000;
66
67  // Gain - Less than 16 bit
68  Int16 gain = 30000;
69
70  /* Use this variable in your code to set the frequency of your sine wave
71      be carefull that you do not set it above the current nyquist frequency! */
72  float sine_freq = 1000.0;
73
74  /* The array to hold the values of the sine lookup table */
75  float table[SINE_TABLE_SIZE] = {0};
76
77   /****************************** Function prototypes ******************************/
78  void init_hardware(void);
79  void init_HWI(void);
80  void ISR_AIC(void);
81  void sine_init(void);
82  #ifdef INCREASE_RES
83    float sine_value(int);
84  #endif
85  /****************************** Main routine ******************************/
86  void main(){
87
88    // initialise sine table
89    sine_init();
90
91    // initialize board and the audio port
92    init_hardware();
93
94    /* initialize hardware interrupts */
95    init_HWI();
96
97    /* loop indefinitely, waiting for interrupts */
98    while(1)
99    {};
100
101  }
102
```

```
103   /******************************** init_hardware() *********************************/
104   void init_hardware()
105   {
106       // Initialize the board support library, must be called first
107       DSK6713_init();
108
109       // Start the AIC23 codec using the settings defined above in config
110       H_Codec = DSK6713_AIC23_openCodec(0, &Config);
111
112     /* Function below sets the number of bits in word used by MSBSP (serial port) for
113      receives from AIC23 (audio port). We are using a 32 bit packet containing two
114      16 bit numbers hence 32BIT is set for receive */
115     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
116
117     /* Configures interrupt to activate on each consecutive available 32 bits
118      from Audio port hence an interrupt is generated for each L & R sample pair */
119     MCBSP_FSETS(SPCR1, RINTM, FRM);
120
121     /* These commands do the same thing as above but applied to data transfers to
122      the audio port */
123     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
124     MCBSP_FSETS(SPCR1, XINTM, FRM);
125
126
127   }
128
129   /******************************** init_HWI() *************************************/
130   void init_HWI(void)
131   {
132     IRQ_globalDisable();      // Globally disables interrupts
133     IRQ_nmiEnable();          // Enables the NMI interrupt (used by the debugger)
134     IRQ_map(IRQ_EVT_XINT1,4);   // Maps an event to a physical interrupt
135     IRQ_enable(IRQ_EVT_XINT1);    // Enables the event
136     IRQ_globalEnable();        // Globally enables interrupts
137
138   }
139
140   /******************* Interrupt Service Routine - Generate Sine*********************/
141
142   void ISR_AIC(void){ // code is generally similar to Lab 2
143     float sample; // the sample value
144     Int16 output; // output value
145       static double index = 0;  // Store the "progress" of the sine wave generation
146       static double prev_freq = 0;   // Previous frequency
147       static double prev_sample = 0;   // Previous sampling frequency
148       // Based on sampling frequency and sine frequency, determine number of samples per cycle
149       static double cycleSampleCount;
150       // Determine the number of intervals to "skip" each time we proceed to the next stage of the
151            sine wave generation
152       static double step;
153
154       if (prev_freq != sine_freq || prev_sample != sampling_freq){     // If frequency has changed,
155            we should take note.
156            index = 0;
157            prev_freq = sine_freq;
158            prev_sample = sampling_freq;
159            cycleSampleCount =  (double) sampling_freq / (double) sine_freq;
```

15

```c
158            step = (double) (SINE_TABLE_SIZE*RES_MULTIPLIER)/(double) cycleSampleCount;
159        }
160      index += step; // "advance" to the next step
161
162      // Check that index is in range
163      if ((int) index >=  SINE_TABLE_SIZE*RES_MULTIPLIER)
164              index -= (SINE_TABLE_SIZE*RES_MULTIPLIER);    // Reset with an offset so that we can
                       try and generate more frequencies
165
166      sample = sine_value((int) round(index)); // get value from table
167      output = (Int16) fabs(sample*gain); // ouput saved to a variable for ease of debugging
168      mono_write_16Bit(output); // write to port
169 }
170 /******************************** sine_init() *************************************/
171 void sine_init(void){
172   int i;
173
174   for (i = 0; i < SINE_TABLE_SIZE; ++i)
175          table[i] = sin(i * ( (2*PI) /(RES_MULTIPLIER*SINE_TABLE_SIZE)));
176 }
177
178 float sine_value(int index){
179      int quadrant = index/SINE_TABLE_SIZE;    // the quadrant in which the cycle is in
180      int modulo = index % SINE_TABLE_SIZE;    // the modulo
181      float value;
182      if (quadrant == 0)
183          value = table[index];
184      else if (quadrant == 1)
185          value =  table[SINE_TABLE_SIZE-modulo-1];
186      else if (quadrant == 2)
187          value =  table[modulo]*-1;
188      else if (quadrant == 3)
189          value =  table[SINE_TABLE_SIZE-modulo-1]*-1;
190      else
191          value =   0;
192
193      return value;
194 }
```