Real-Time Digital Signal Processing : Lab 3 - Interrupt I/O

Jiyu Fang CID: 01054797 Deland Liu CID: 01066080

February 2018

1 Objectives

- Understand the concept of interrupts
- Use interrupt driven code in c to process audio signal samples sampled from software signal generator or from a look-up table containing signal data

2 Equipment

- Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator(DSK6713)
- Oscilloscope
- Software signal generator

3 Exercise 1: Interrupts service routine

Code description (the complete program listings can be found in **Appendix**):

- **main()** initializes hardware settings by calling functions init_hardware() and init_HWI(), then goes into an infinite loop waiting for interrupts to take place.
- **init_hardware** initializes the board support library by calling function DSK6713_init(), sets the bit resolution and the sampling frequency of read to 16 bit and 8kHz respectively. It also ensures that an interrupt is generated on each available 32-data packet from audio port and unpack this 32 bit sample into 2 16-bit samples from left and right channels respectively. When writing to audio ports, the processed 16-bit sample is copied twice to form a 32-bit sample so that both left and right get a copy of the processed 16-bit sample
- **init_HWI()** maps the interrupt event initialized above to a specific interrupt priority (in this case, interrupt priority is 4). We have already configured this physical event under interrupt HWI_INT4, which is of the highest priority among those available for audio ports, and enables it. We also configured this interrupt in the configuration tool such that whenever the interrupt takes place the function ISR_AIC() will be executed.
- **ISR_AIC()** contains some code that we want to execute when the data is available from the audio port. In this case, the function reads in samples at a rate of 8kHz by calling the function mono_read_16Bit(), which sums up the left and right channel samples and divides the sum by 2. In this exercise, the absolute value of this sum will be passed to each channel, by calling the function mono_write_16Bit(). By doing so on each sample, the output waveform will be the rectified version of the input waveform. Note that since the bit resolution is configured as 16 bit, we define *sampin* as of data type *short* while *samp* is of type *unsigned short* since it must be positive:
- **Interrupt Programming** Declaring and initializing handle enables us to write and read from hardware device. The traditional polling method requires program polling in a while loop and waiting for sample to arrive and read. Instead program can be interrupt driven, whereas ISR is loaded and executed when interrupt occurs. In this exercise, interrupt source is MCSP_1_Receive and ISR is mapped to this source; hence current program (looping within infinite while loop)

suspended as a sample ready to be read and branched to ISR_AIC function. By writing other codes under the main function, the program could do some other useful tasks if no signal is present. The MCBSP_FSETS() functions would have to configure interrupts for both data receive and transfer to audio port, making sure an receive interrupt is raised when McBSP contains a new sample and a transmit interrupt is raised when McBSP is free to receive a processed sample from the program.

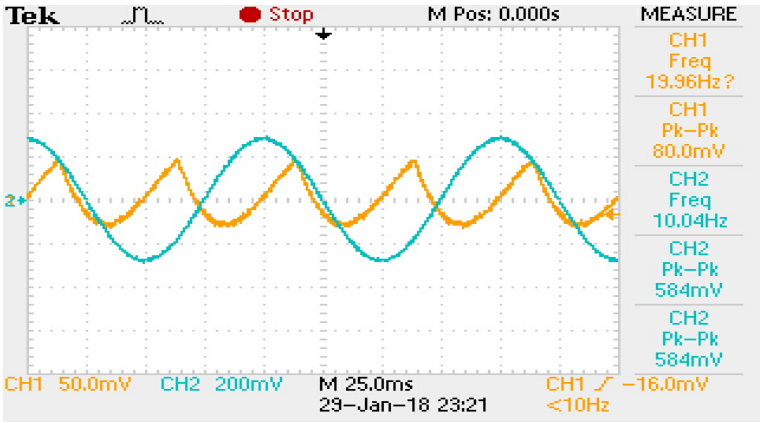
```
141 /********************* WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
142
143 void ISR_AIC (void)
144 {
145
146         sampin = mono_read_16Bit();
147         samp = abs(sampin);
148         mono_write_16Bit(samp);
149
150
151 }
```

After recompiling the program, use the software signal generator to source the audio port as the maximum peak-to-peak voltage provided by it is about 1.8V under 0 attenuation while the audio port can take a maximum $2V_{RMS}$. Generate a sine wave of frequency between 10Hz and 3.8kHz and measure the signal on left channel output connector on the oscilloscope.

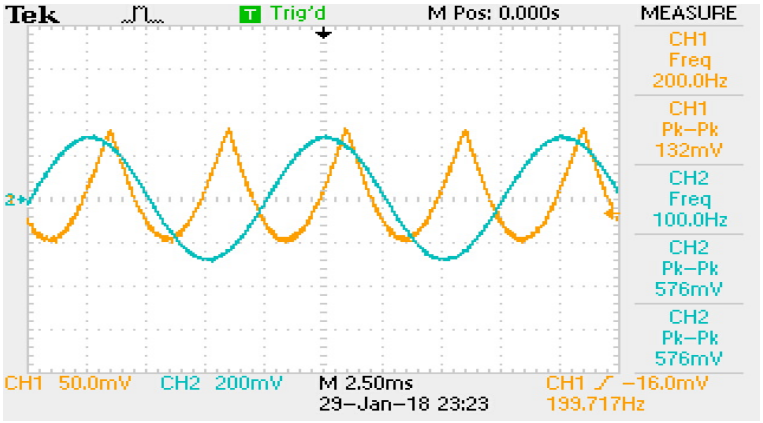
Observations:

Left channel line in Left channel line out

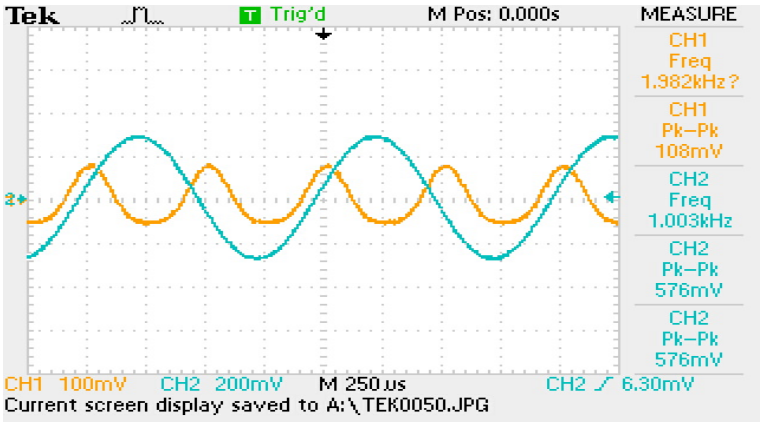
- $f_0 = 10Hz$



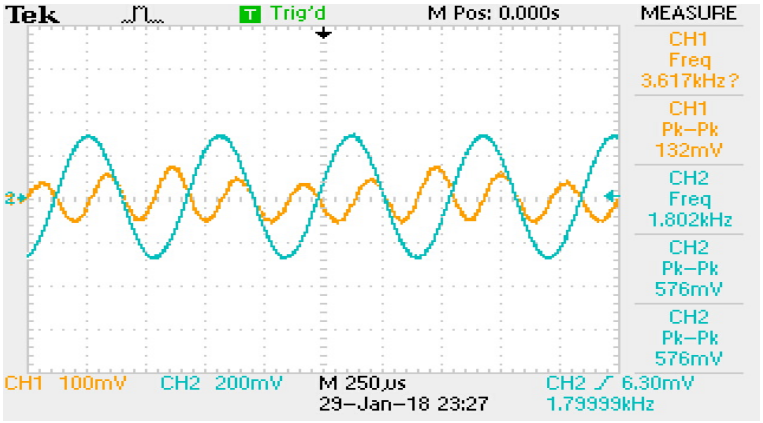
- $f_0 = 100Hz$

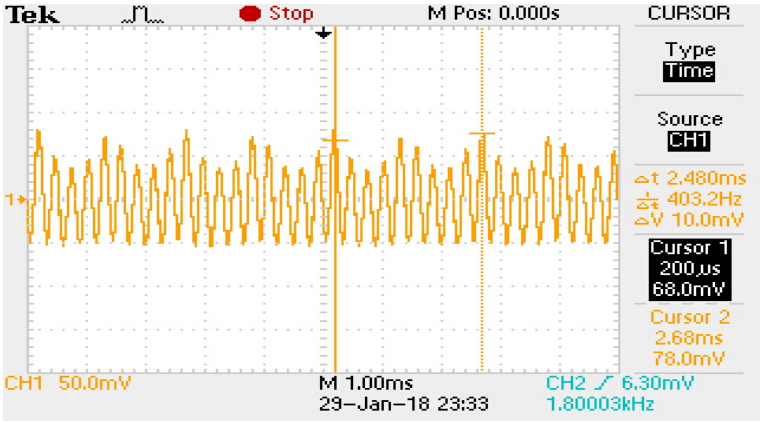


- $f_0 = 1kHz$

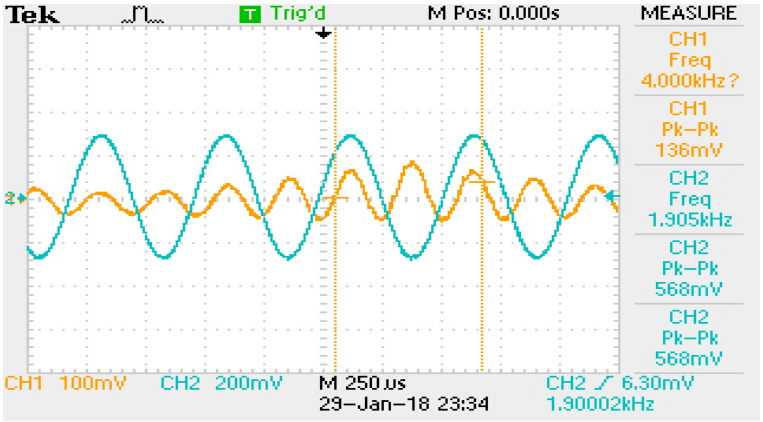


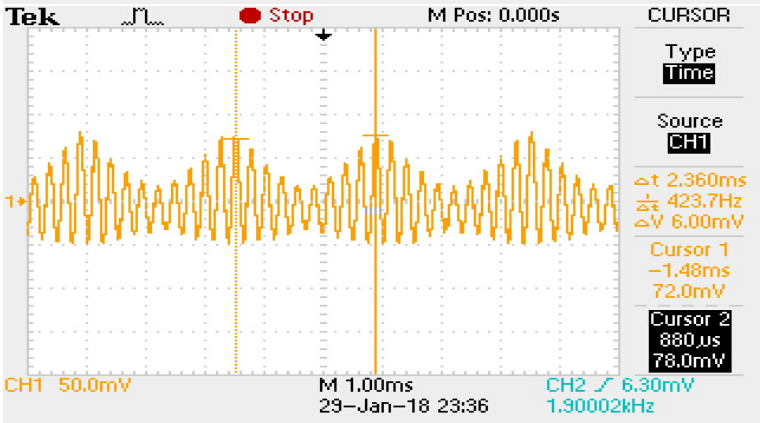
- $f_0 = 1.8kHz$ ($\frac{1}{2T_{beat}} \approx 400Hz$)



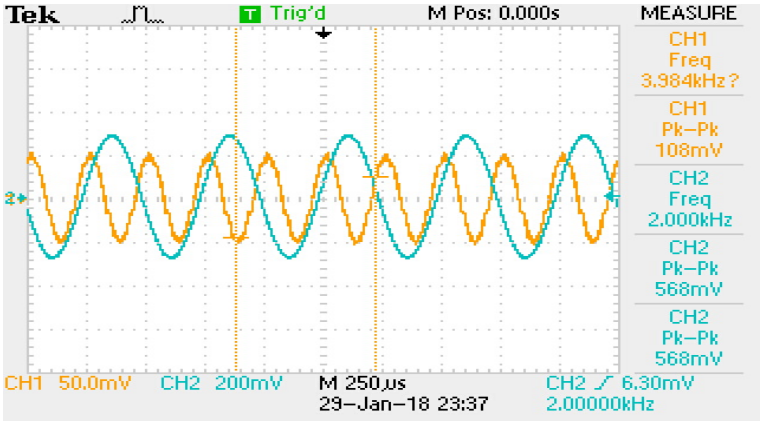


- $f_0 = 1.9kHz$ ($\frac{1}{T_{beat}} \approx 400Hz$)

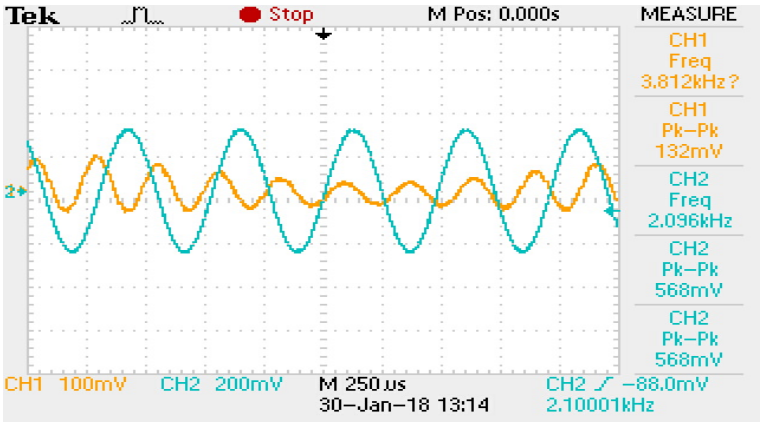


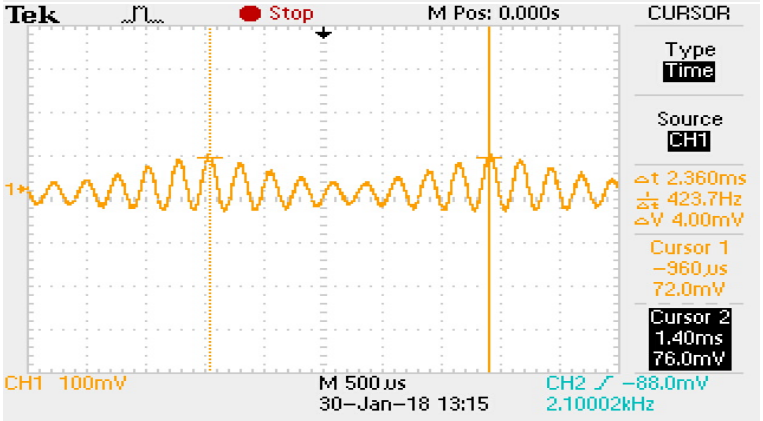


- $f_0 = 2kHz$

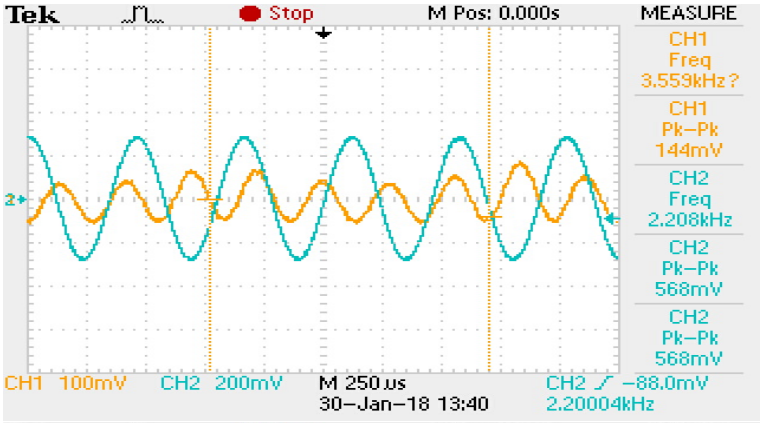


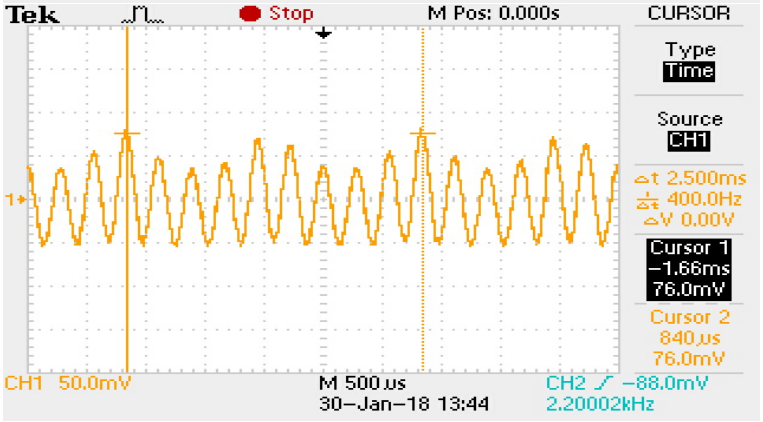
- $f_0 = 2.1kHz$ ($\frac{1}{T_{beat}} \approx 400Hz$)



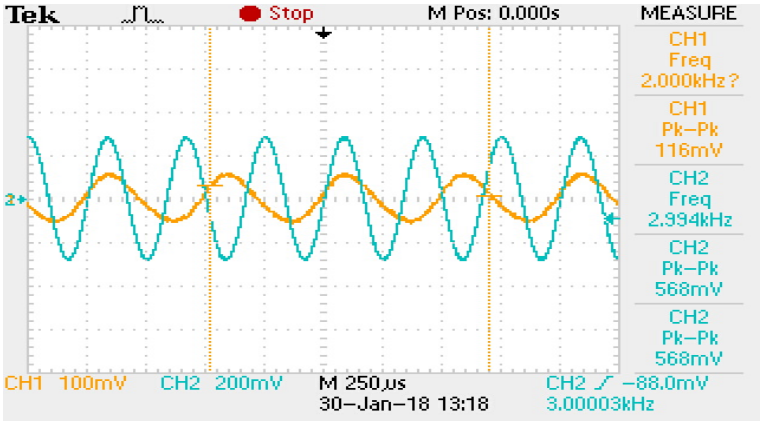


- $f_0 = 2.2kHz$ ($\frac{1}{2T_{beat}} \approx 400Hz$)

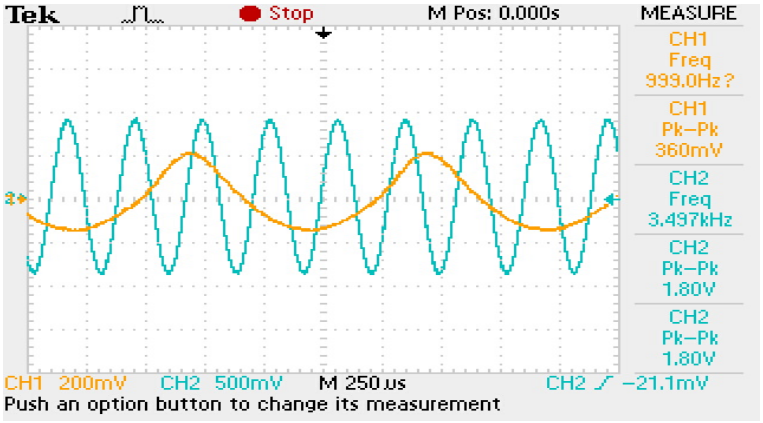




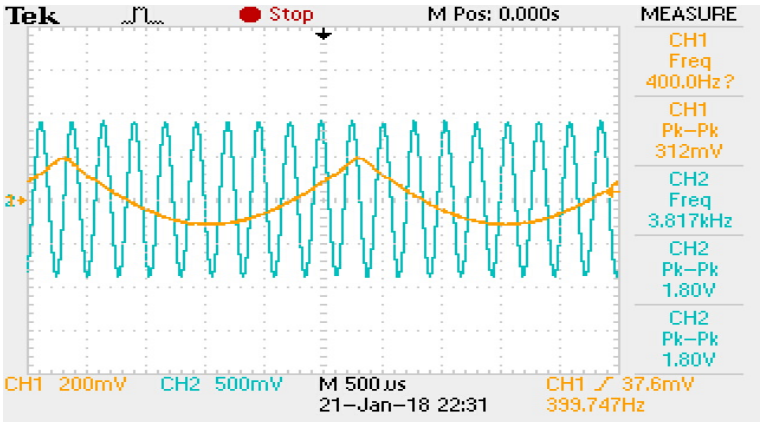
- $f = 3kHz$



- $f = 3.5Hz$

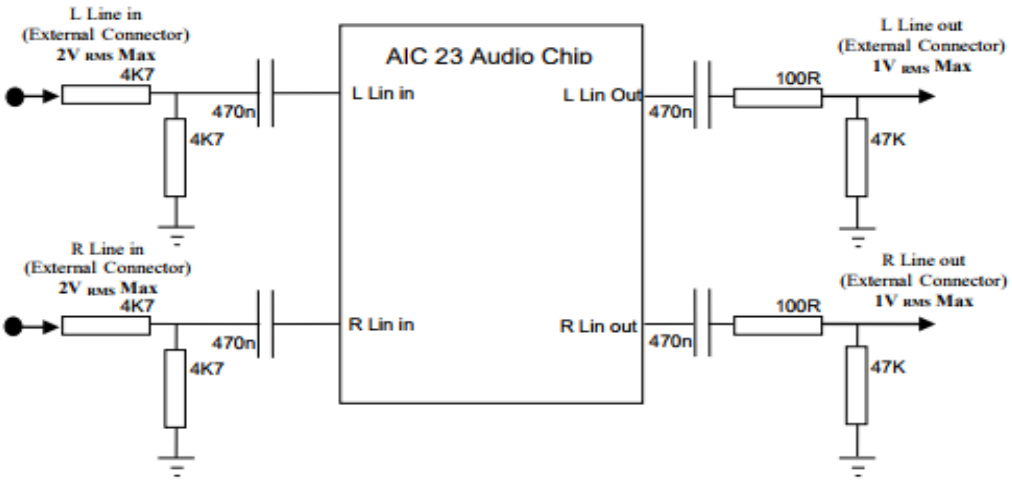


- $f = 3.8Hz$

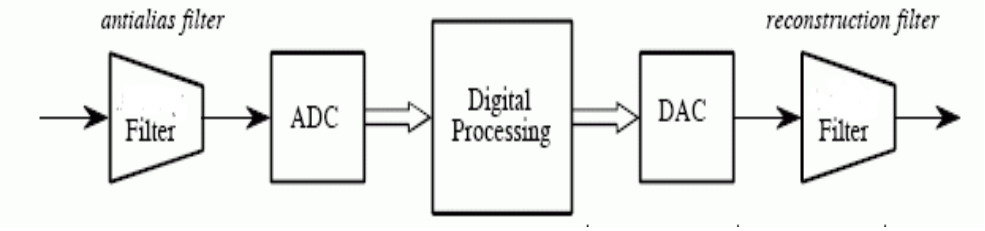


Insights:

As we can see from the lab results above the output waveform is sort of inverted and centered around $0V$ instead of always being positive. This is due to the inverting amplifier of Digital to Analog Converter inside AIC 23 Audio Chip which inverts the waveform and since the Lin Out of both channel are AC coupled with a capacitor of $479nF$, the dc component is filtered out and the waveform is centered around $0V$ so that it has zero time average.



Before explaining other phenomenon observe, we need to be familiar what is at the input and output ports of the AIC23 chip



Both filters are low pass filters with cut-off frequency approximately at half of the sampling frequency, and the sampling frequencies for both ADC and DAC are $8kHz$ in this case, hence Nyquist Frequency $f_N = 4kHz$.

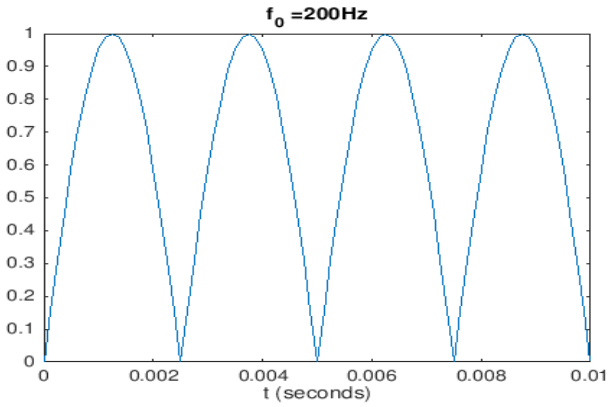
The output waveform will only have the correct frequency(approximately twice the frequency of the input waveform) approximately below $2kHz$. The explanation for this comes with the DTFT of the sampled sine wave:

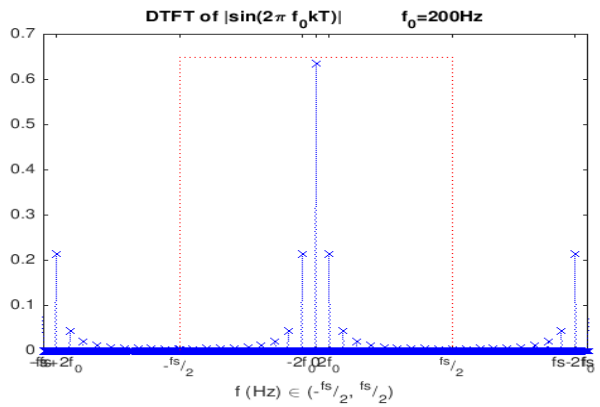
$$\mathcal{F}_d(w) = \frac{1}{T}\sum_{k=-\infty}^{\infty}\mathcal{F}(w - kw_s)$$

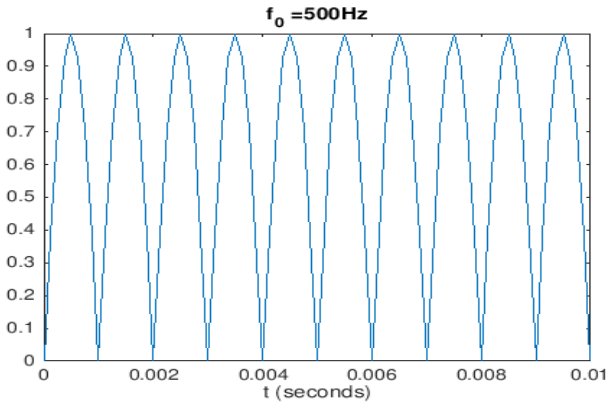
Where $\mathcal{F}(w) = j\pi(\delta(w + w_0) - \delta(w - w_0)), w_0 = 2\pi f_0, w_s = 2\pi f_s = \frac{2\pi}{T}$, and k is an integer. The frequency spectrum of a sampled signal is equal to $\frac{1}{T}$ multiplied by the sum of infinitely many copies of the frequency spectrum of the original analogue signal, with each copy shifted by an integer multiple of the sampling frequency.
Since the wave is then rectified by using $abs()$, the rectified samples have doubled the frequency, the DTFT of sampled rectified sine wave at the input to the DAC of the audio chip could be approximated as:

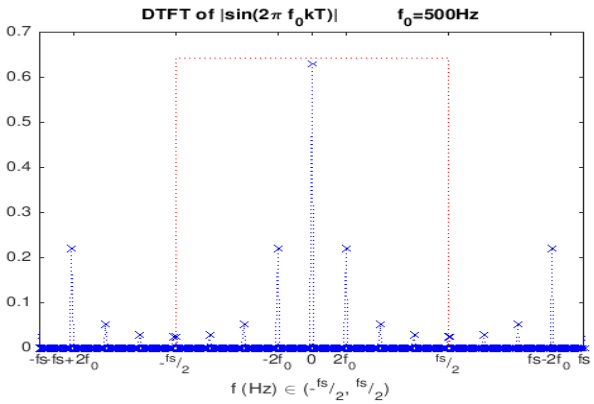
$$\mathcal{F}_{dr} = \frac{1}{T}\sum_{k=\infty}^{\infty} j\pi(\delta(w - kw_s + 2w_0) - \delta(w - kw_s - 2w_0))$$

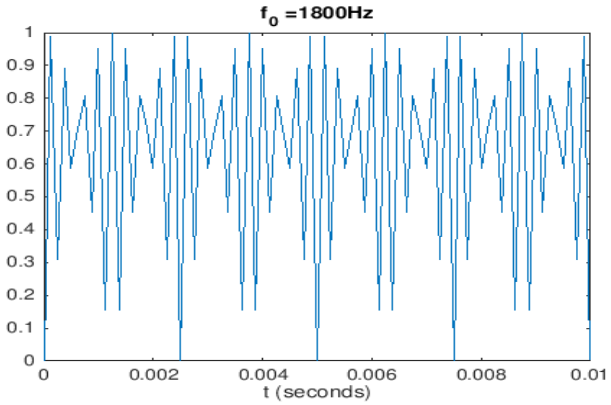
However, it is not a sinusoidal due to the sharp peak resulted from the rectifying process.
By using Matlab we could plot out the expected behaviour of $|sin(2\pi f_0 kT)|$ in both time and frequency domains, the complete Matlab code could be found in the **Appendix**:

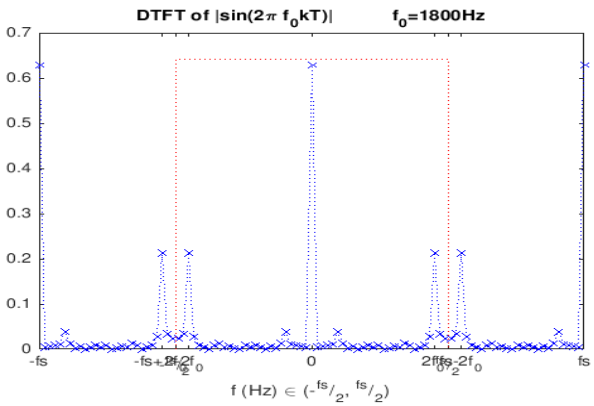


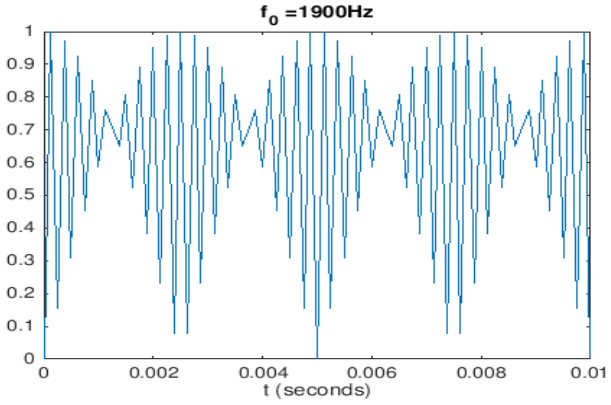


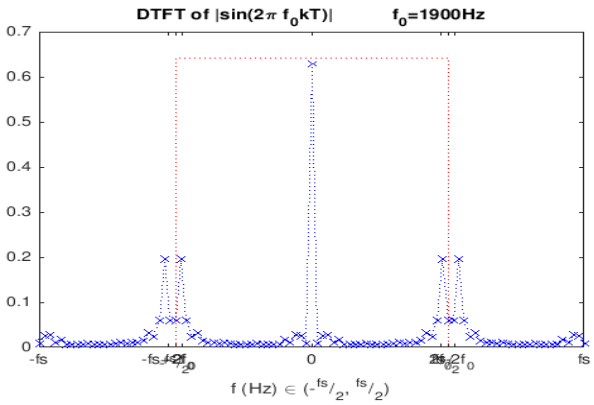

f_0 =500Hz
t (seconds)

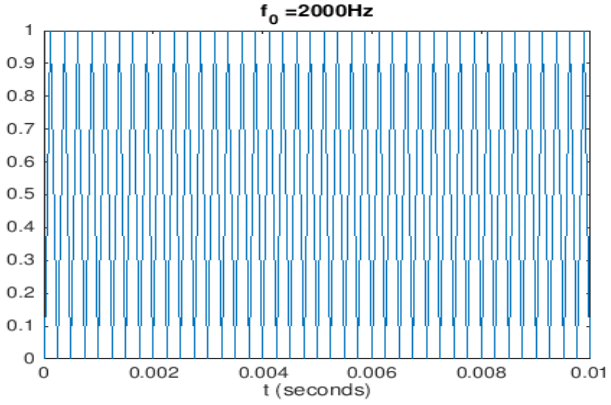

DTFT of |sin(2π f_0kT)|
f_0=500Hz
f (Hz) ∈ (-fs/2, fs/2)

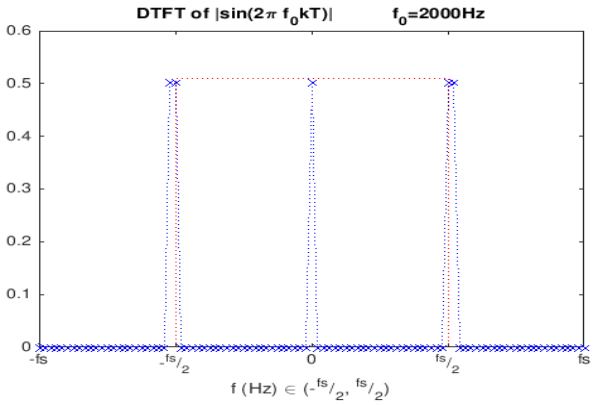

f_0 =1800Hz
t (seconds)

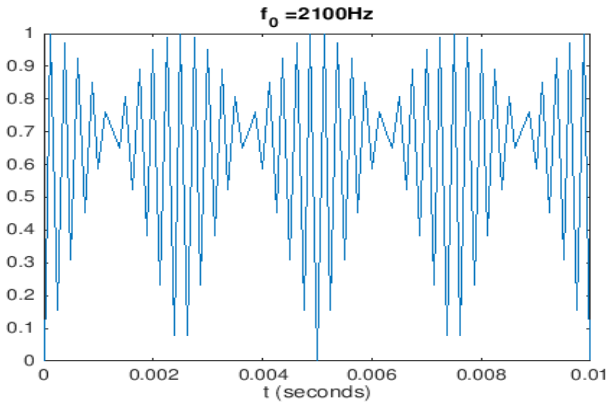

DTFT of |sin(2π f_0kT)|
f_0=1800Hz
f (Hz) ∈ (-fs/2, fs/2)

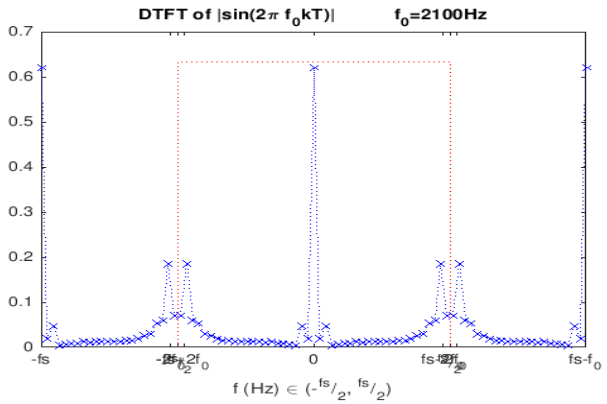

f_0 =1900Hz
t (seconds)

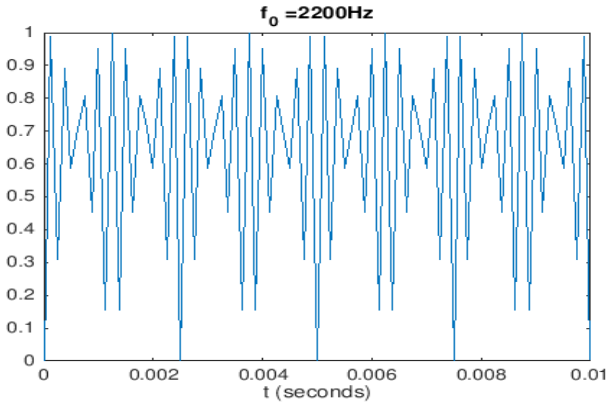

DTFT of |sin(2π f_0kT)|
f_0=1900Hz
f (Hz) ∈ (-fs/2, fs/2)

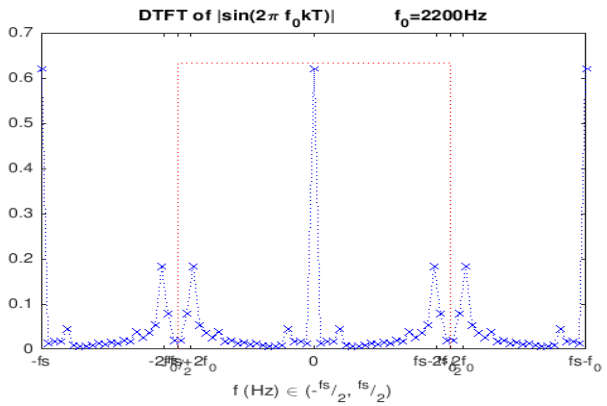

f_0 =2000Hz
t (seconds)

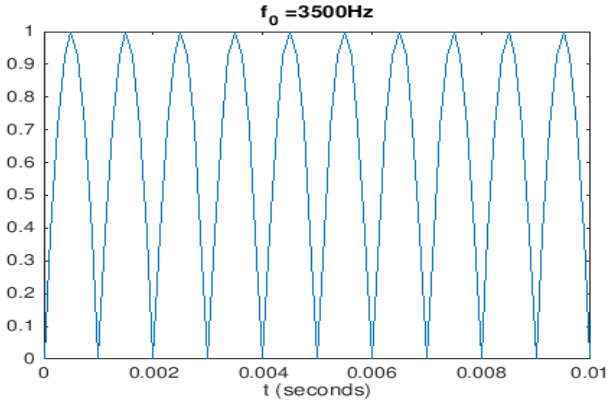

DTFT of |sin(2π f_0kT)|
f_0=2000Hz
f (Hz) ∈ (-fs/2, fs/2)

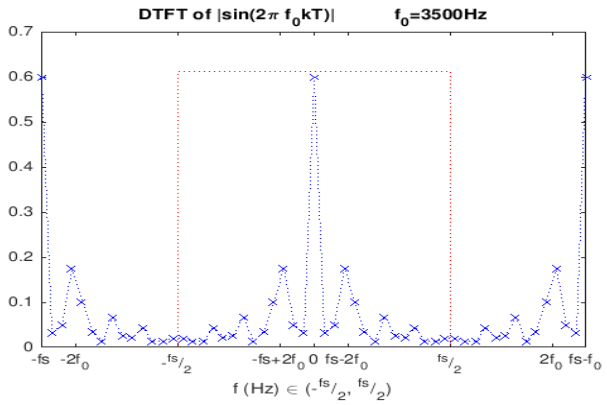


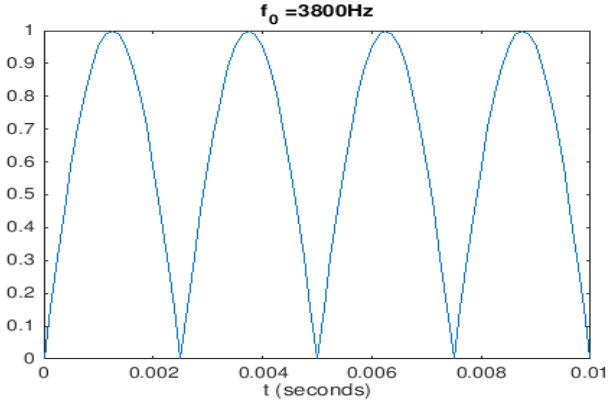


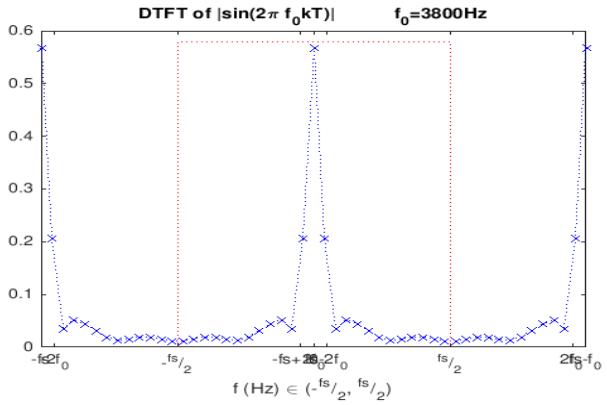












One could notice that the output will only have right frequency only if $2f_0 <= f_N$, hence f_0 needs to be below $2kHz$, according to Nyquist-Shannon Sampling Theorem. Note that the spectrum of a digital signal above Nyquist Frequency is not zero.

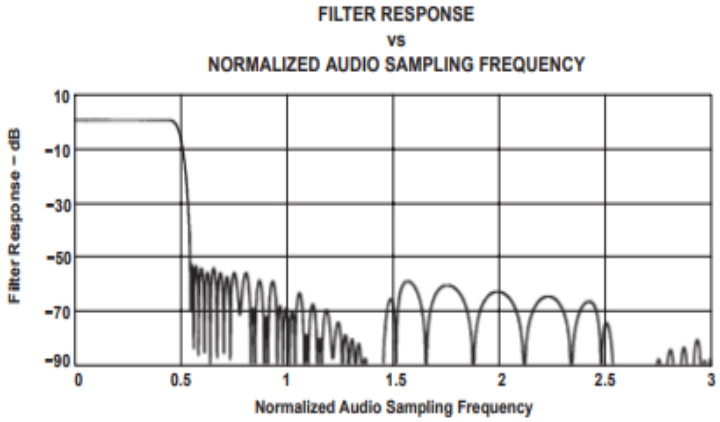
Given f_0 is below the $2kHz$, when f_0 gets closer to this frequency, a behaviour shown as *beatwave* could be observed. It seems like the output has a low frequency behaviour, even though by looking at the frequency diagram it doesn't seem to have that frequency component. This might be due to the following reasons :

- Defining $\Delta = f_N - 2f_0$, the sampled rectified sine wave could be written as:

$$\begin{aligned} sin(4\pi f_0 kT) &= sin(2\pi(f_N - \Delta)kT) \\ &= sin(2\pi f_N kT)cos(2\pi\Delta kT) - cos(2\pi f_N kT)sin(2\pi\Delta kT) \\ &= -cos(2\pi f_N kT)sin(2\pi\Delta kT) \\ &= (-1)^{k+1}sin(2\pi\Delta kT) \end{aligned} \tag{1}$$

  As Δ gets smaller, the output will behave like a sinusoidal with frequency f_N and the amplitude modulated by another sinusoidal with frequency Δ, hence $f_{beat} = 2\Delta$, which could be shown by the frequency measurements above when f_0 equals $1.8kHz$, $1.9kHz$,$2.1kHz$ and $2.2kHz$ respectively.

- The reconstruction filter at the DAC output is not ideal, in other terms, both the filter response and the signal spectrum above f_N is non-zero:



  The resulting wave will look like a sum of two sinusoidals with frequencies f_0 and $f_s - f_0$, if they are close enough. Note that:

$$cos(2\pi f_1 t) + cos(2\pi f_2 t) = 2cos(2\pi(\frac{f_1 + f_2}{2})t)cos(2\pi(\frac{f_1 - f_2}{2})t)$$

  when $f_1 = f_s - f_0$, $f_2 = f_0$:

$$cos(2\pi f_1 t) + cos(2\pi f_2 t) = 2cos(2\pi f_N t)cos(2\pi(f_N - 2f_0)t)$$

  which implies similar beating behavior as the first reason does.

When $f_0 > 2kHz$, the aliasing frequency $f_a = f_s - 2f_0$ is extracted by the reconstruction filter, since the image copies of the analogue signal spectrum starts to overlap. Similar beating behaviour might still occur if $2f_0$ is close to f_N or due to non-ideal reconstruction filter, as shown by the lab results.

At some larger range of frequencies around $2kHz$, the power of frequency components at $2f_0$ or $fs{-}2f_0$ is much larger compared to other frequencies', the output waveform is smoother, like a sinusoidal.

When $f_0 > 4kHz$, the output barely shows anything due to the anti-aliasing filter at the input of ADC.

Hence, we could conclude that the rectifier would work fine up to $f_0 = 1.8kHz$. However, f_0 should not be too small due to the high-pass filter at the channel line out, which has a cut-off frequency of about $7Hz$.

4 Exercise 2: Interrupt-driven sine wave

Code description (the complete program listings can be found in **Appendix**):

- All other parts of the code are the same as in Exercise 1 except for some adjustments as below:
- Modify in the configuration tool the interrupt source of HWI_INT5 to MCSP_1_Transmit, so that the ISR will be executed when MCBSP is ready to write a sample to the audio port.
- Change the event symbol arguments inside IRQ_map() and IRQ_enable to IRQ_EVT_XINT4, which corresponds to a transmit interrupt.
- The sine wave is generated by reading from the *table*, which is initialized before reading by calling the function sine_init(), by calling the function sin_gen(), the sampling_freq is set to 8kHz. Since the direct output has very small amplitudes, a *Gain* of 10000 is multiplied to the reading value while making sure the output waveform does not exceed $1V_{RMS}$. By modifying the value of sine_freq one could change the input frequency f.
- In this exercise, interrupt source is a transmit interrupt instead of a receive interrupt. The DAC is clocked at 8kHz and every time the DAC is clocked, a sample from McBSP buffer is stored to DAC internal buffer and a sample from internal buffer transmitted to output port. At this instance, a transmit interrupt is sent to the program, informing program that a write operation for a new sample to the McBSP can now occur. As a result, program loads and executes an ISR, in which new sample from SineWave generated, rectified and written to buffer on McBSP serial port.

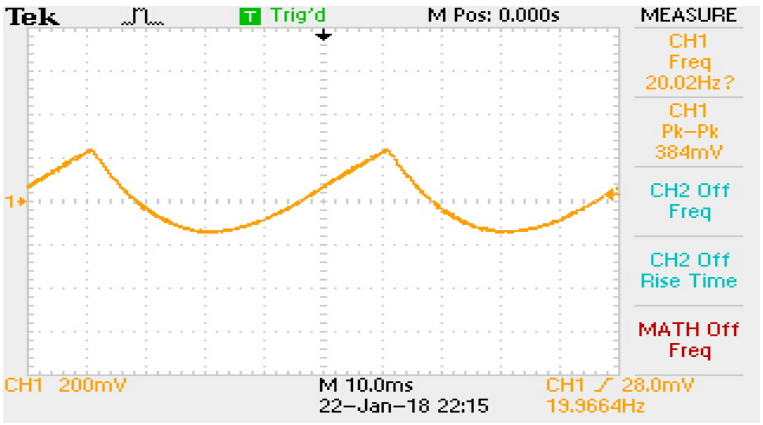
the new Interrupt Service Routine is shown below:

```
141 /******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
142
143 void ISR_AIC (void)
144 {
145
146         sampin = sinegen()*Gain;
147         samp = abs(sampin);
148         mono_write_16Bit(samp);
149
```

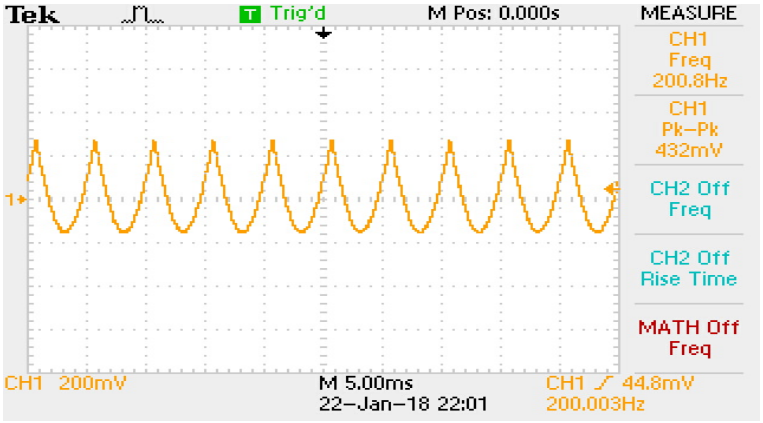
Observations:

left channel line out

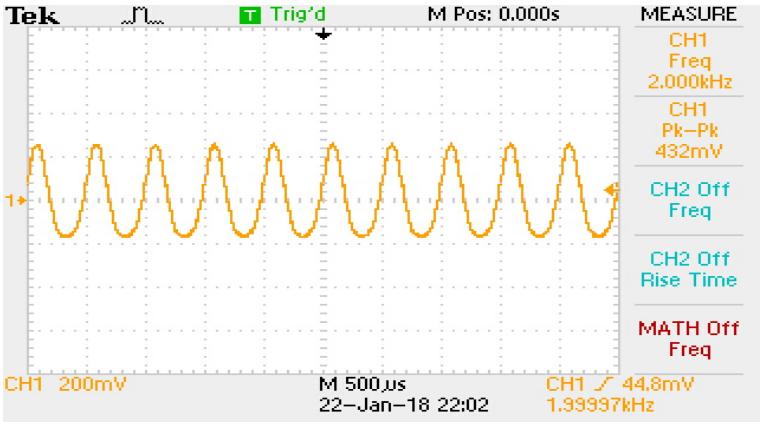
- $f_0 = 10Hz$



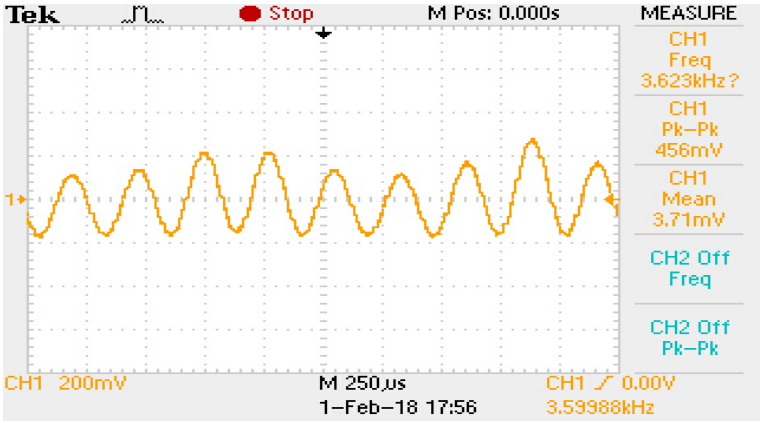
- $f_0 = 100Hz$

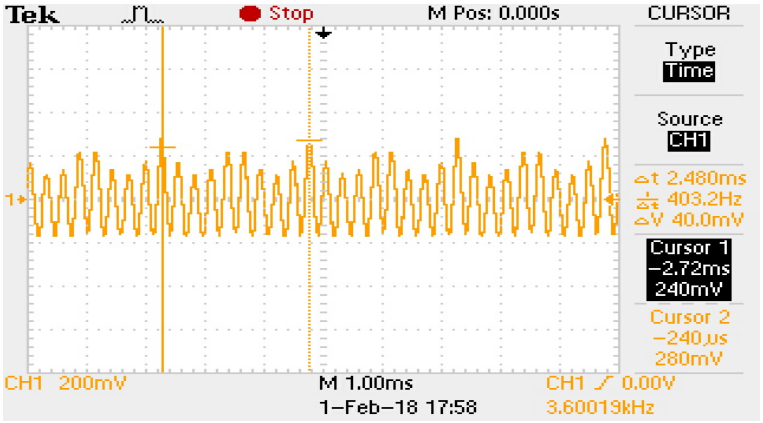


- $f_0 = 1kHz$

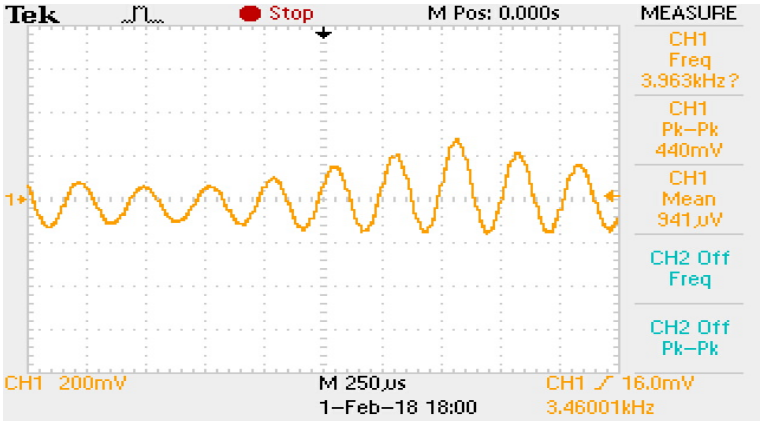


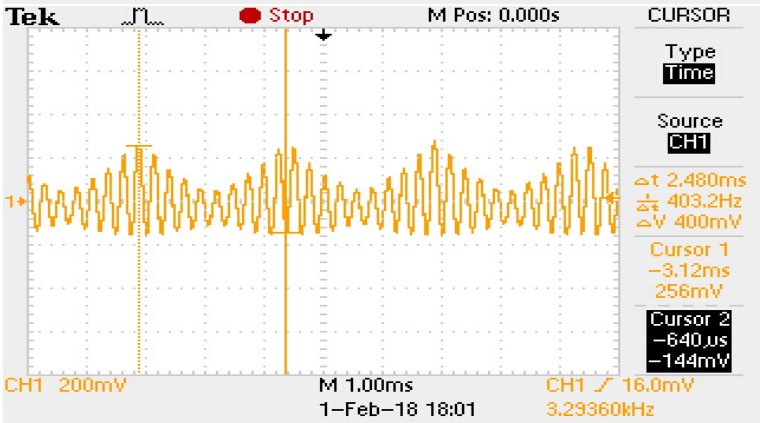
- $f_0 = 1.8kHz$



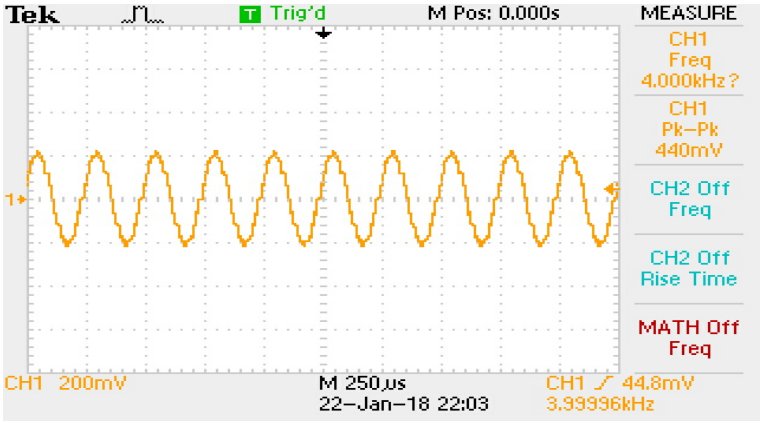

Tek
Stop
M Pos: 0.000s
CURSOR
Type
Time
Source
CH1
Δt 2.480ms
1/Δt 403.2Hz
ΔV 40.0mV
Cursor 1
−2.72ms
240mV
Cursor 2
−240µs
280mV
CH1 200mV
M 1.00ms
1−Feb−18 17:58
CH1 ⁄ 0.00V
3.60019kHz

- $f_0 = 1.9kHz$

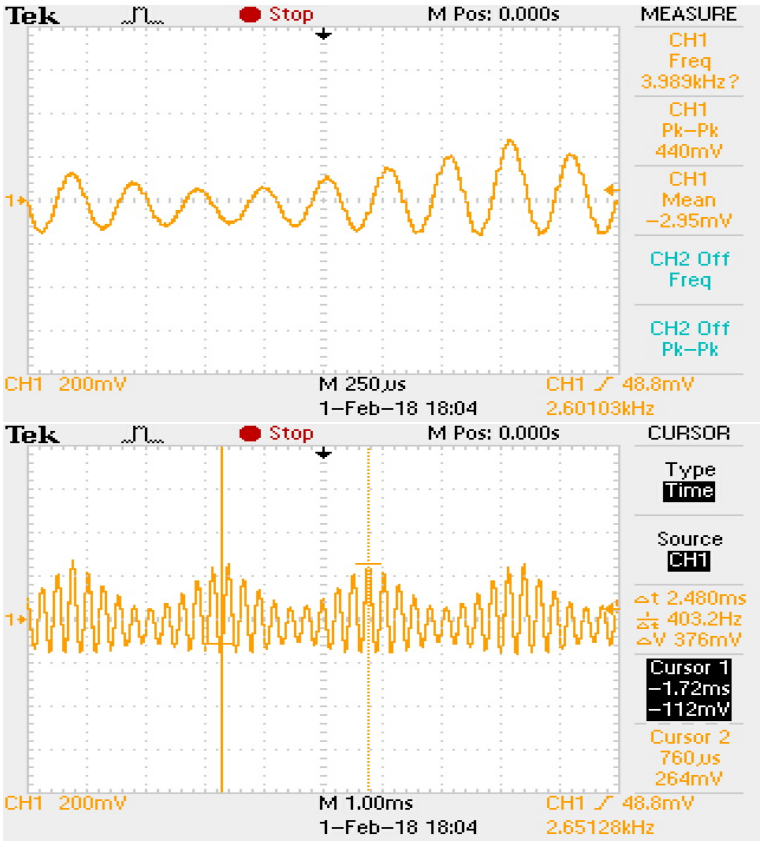

Tek
Stop
M Pos: 0.000s
MEASURE
CH1
Freq
3.963kHz?
CH1
Pk−Pk
440mV
CH1
Mean
941µV
CH2 Off
Freq
CH2 Off
Pk−Pk
CH1 200mV
M 250µs
1−Feb−18 18:00
CH1 ⁄ 16.0mV
3.46001kHz


Tek
Stop
M Pos: 0.000s
CURSOR
Type
Time
Source
CH1
Δt 2.480ms
1/Δt 403.2Hz
ΔV 400mV
Cursor 1
−3.12ms
256mV
Cursor 2
−640µs
−144mV
CH1 200mV
M 1.00ms
1−Feb−18 18:01
CH1 ⁄ 16.0mV
3.23360kHz

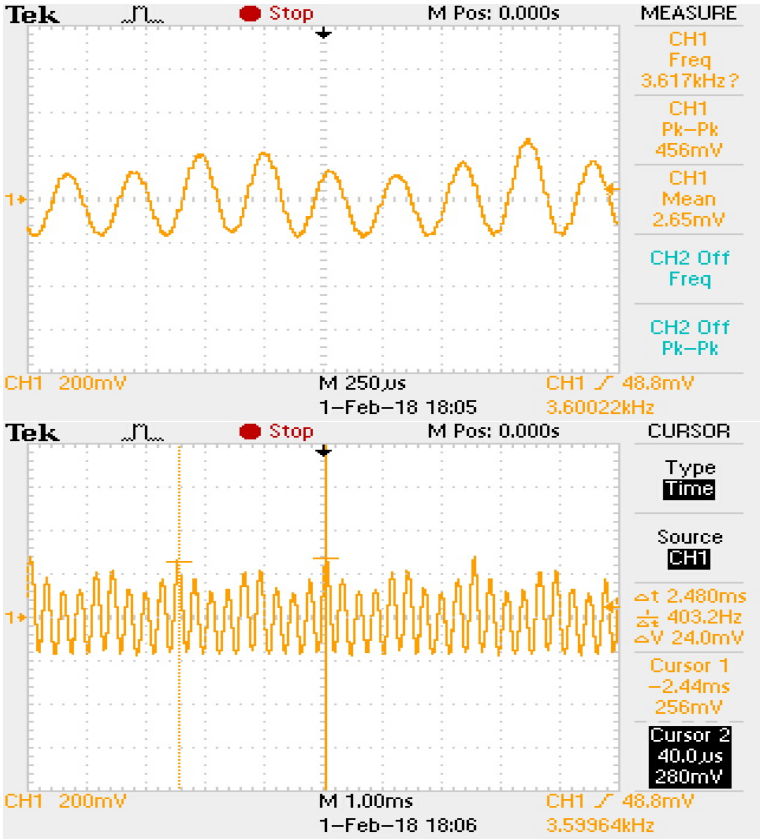
- $f_0 = 2kHz$



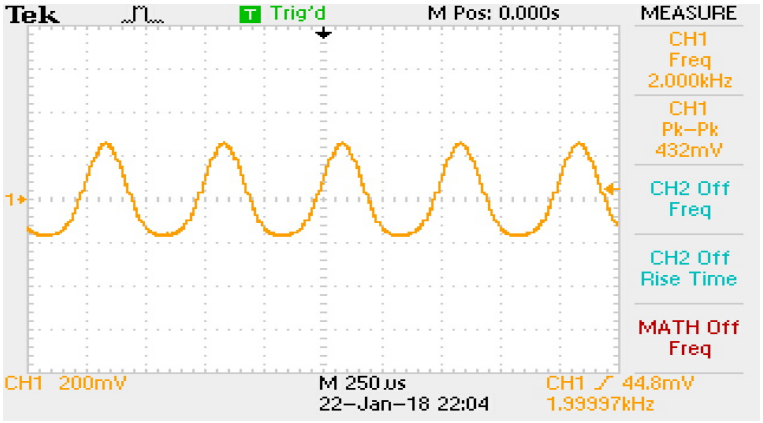
- $f_0 = 2.1kHz$



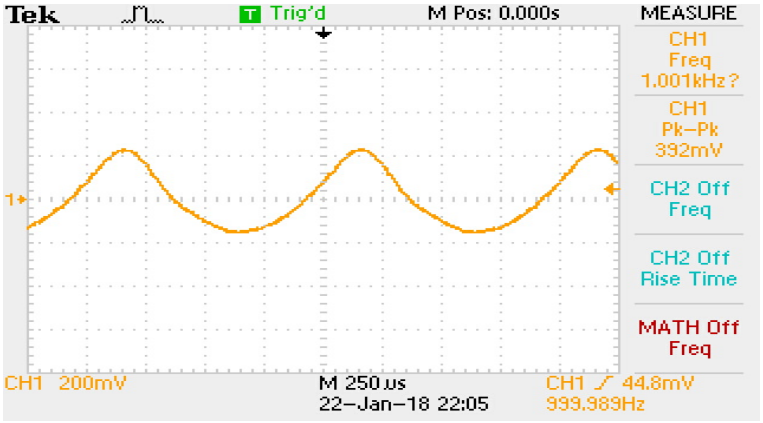
- $f_0 = 2.2kHz$



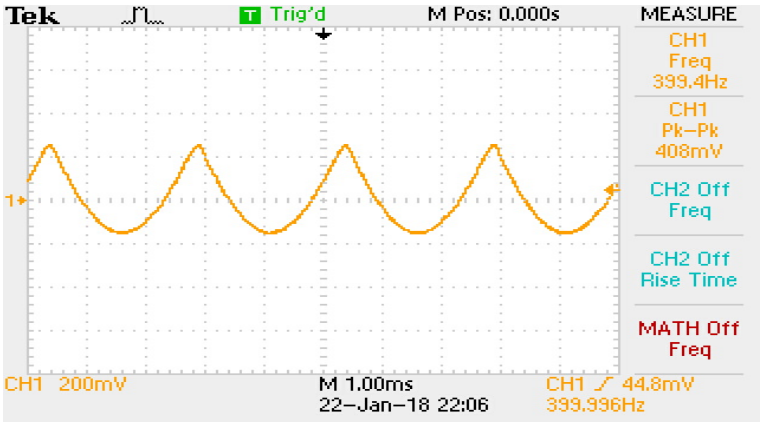
- $f_0 = 3kHz$



- $f_0 = 3.5Hz$



- $f_0 = 3.8Hz$



- As shown in results, similar phenomenons in Ex1 are observed. As sine_freq approaches close to 2kHz (the rectified sine waves approaches close to 4kHz), beat wave can be observed. If Nyquist Shannon Sampling theorem is violated, a higher frequency component can be observed as a lower frequency component mirrored about the Nyquist frequency. This is the aliasing effect.

5 Conclusion

In this session we learned how to use interrupt to implement real time tasks and analyzed the results together with the knowledge of hardware structure. In addition to providing signal below Nyquist Frequency following the Nyquist-Shannon Sampling Theorem, one should expect to see a low frequency behavior that may appear to be present in a signal shown as *beats* after a sampling mechanism when signal frequency is close enough to the Nyquist Frequency.

6 Appendix

- Exercise 1: Interrupts service routine

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

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

                                                    LAB 3: Interrupt I/O

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

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

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

#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

/******************************* Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
                        /*************************************************************************/
                        /*   REGISTER                    FUNCTION                            SETT
                        /*************************************************************************/\
    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;

/**************************** Global Variables ****************************/

short sampin;
unsigned short samp;

 /************************** 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 de
        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)
{

                sampin = mono_read_16Bit();
                samp = abs(sampin);
                mono_write_16Bit(samp);


}
```

- Exercise 2: Interrupt-driven sine wave

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

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

                                                        LAB 3: Interrupt I/O

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

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

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

#include <stdlib.h>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

/******************************* Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
                        /*************************************************************************/
                        /*   REGISTER                    FUNCTION                                SETT
                        /*************************************************************************/\
    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;

/******************************* Global Variables ********************************/
#define SINE_TABLE_SIZE 256
#define PI 3.141592653589793

int sampling_freq = 8000;
float sine_freq = 1000.0;
short sampin;
unsigned short samp;
float table_index = 0;
float table[SINE_TABLE_SIZE];

/* Audio channel gain values, calculated to be less than maximum acceptable value. */
Int32 Gain = 10000;


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

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


      // initialize board and the audio port
  init_hardware();
   // initialize the look-up table
  sine_init();
  /* 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 de
        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)
{

                sampin = sinegen()*Gain;
                samp = abs(sampin);
                mono_write_16Bit(samp);


}

float sinegen(void)
{
/*  This code produces a fixed sine of 1KHZ (if the sampling frequency is 8KHZ)
    using a digital filter.
         You will need to re-write this function to produce a sine of variable frequency
         using a look up table instead of a filter.*/

        // temporary variable used to output values from function
        float wave;
        int rounded_index;
        float factor;
        //determined how many table values needed to skip to get the next correct sample value
        factor = SINE_TABLE_SIZE * sine_freq / sampling_freq;
        if(table_index * factor >= SINE_TABLE_SIZE)
        {
                table_index  = (table_index * factor - SINE_TABLE_SIZE)/factor;
        }
        //the index of table has to be int data type, use round() to ensure accuracy
        rounded_index = round(table_index * factor);
        wave = table [rounded_index];
```

```
        table_index++;


    return(wave);

}


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

- Matlab Simulation (Time Domain)

```
fs = 8000;
fo = [200,500, 1800, 1900, 2000, 2100, 2200, 3500, 3800];
r=length(fo);
for i=1:r
t = (0:1/fs:0.02);
s = abs(sin(2*pi*fo(i)*t));
figure;
plot(t,s);
hold;
xlabel('t (seconds)')
title(strcat('f_{0} = ',num2str(fo(i)), 'Hz'));
end
```

- Matlab Simulation (Frequency Domain)

```
fs = 8000;
fsd = 1*fs;
fo = [200,500, 1800, 1900, 2000, 2100, 2200, 3500, 3800];
r=length(fo);
j = 12;
ndouble = realmin('double');
for i=1:r
    t = (0:1/fsd:(j/fo(i)-1/fsd));
    n = length(t);
    k = floor(n/2);
    f = (-fsd*k/n-3*fsd/n:fsd/n:fsd*k/n+2*fsd/n);
    fk = horzcat(f-fsd*(2*k+6)/n, f ,f+fsd*(2*k+6)/n);
    m = length(f);
    ff = horzcat(-fs/2-ndouble,(-fs/2:fs/n:fs/2),fs/2+ndouble);
    filter = horzcat(0,ones(1,length(ff)-2,'double'),0);
    %f=(0:fsd/n:fsd-fsd/n);
    s = abs(sin(2*pi*fo(i)*t));
    x = abs(fft(s))./n;
    filter = filter.*max(x)*1.02;
    %j = find(x==min(x), 1,'first');
    %y=x(1:n);
   % m = horzcat(x(n-k-2:n),x(1:k+3));

   if n>k+2 && n<k+3
        y = horzcat(x(n-k-2:n),x,x(1:k+3-n));

    elseif n>=k+3
         y = horzcat(x(n-k-2:n),x(1:k+3));

   elseif n<=k+2
         y = horzcat(x(2*n-k-2:n),x,x,x(1:k+3-n));
    end

    yk = horzcat(y(m-4:m),y(1:m-2),y(4:m-2),y(4:m),y(1:5));
    figure;
    plot(fk,yk,'b:x',ff,filter,'r:');
    if i<5
    xticks([-fs, -fs+2*fo(i), -fs/2,-2*fo(i),0, 2*fo(i),fs/2,fs-2*fo(i),  fs]);
    xticklabels({'-fs','-fs+2f_{0}','-^{fs}/_{2}','-2f_{0}', '0','2f_{0}','^{fs}/_{2}','fs-2f_{0}',
    elseif i==5
    xticks([-fs, -fs/2,0,fs/2, fs]);
    xticklabels({'-fs','-^{fs}/_{2}',  '0','^{fs}/_{2}', 'fs'});
    elseif i>5
    xticks([-fs,-2*fo(i), -fs/2,-fs+2*fo(i),0, fs-2*fo(i),fs/2,2*fo(i), fs]);
    xticklabels({'-fs','-2f_{0}','-^{fs}/_{2}','-fs+2f_{0}', '0','fs-2f_{0}','^{fs}/_{2}','2f_{0}',
    end
    xlabel('f (Hz) \in (-^{fs}/_{2}, ^{fs}/_{2})')
    xlim([-fsd, fsd]);
    %ylabel(strcat('|X(e^{jw})|'));
    title(strcat('DTFT of |sin(2\pi',' f_{0}','kT)| ',{'           '},'f_{0}=',num2str(fo(i)),'Hz'))
    hold;


end
```