Jurgen Shiqerukaj - CID 00938154 Jason Yuan – CID 00954645 Due 10th February 2017

Lab 3 – Interrupt I/O

Contents

Overview

In this lab session we move our attention to Interrupt service routines with the main focus using a ISR to rectify a input signal using the RT DSP and show this at the output.

Question 1

From Trace 1 (below) we can see that we do indeed get a rectified signal. It is however important to note that due to the audio chip having an inverter at the output, we have inverted output channel (orange) to see this rectified effect clearly. We can also see that the rectified output is centered on 0. This is due to there being a High pass filter at the output of the DSP which removes DC offsets. This can also be explained mathematically, the Fourier series of a full wave rectified sine wave is as shown:

$$a_0 = \frac{2}{\pi}\int_0^{\pi} |\sin x|\,dx = \frac{2}{\pi}\int_0^{\pi} \sin x\,dx = \frac{4}{\pi}$$

$$where \cos n\pi = (-1)^n$$

$$a_n = \frac{2}{\pi}\int_0^{\pi} \sin x \cos nx\,dx$$

$$2\sin x\,\cos nx = sin[(n+1)x] - sin[(n-1)x]$$

$$a_n = \frac{1}{\pi}\int_0^{\pi} sin[(n+1)x]\,dx - \frac{1}{\pi}\int_0^{\pi} sin[(n-1)x]\,dx$$

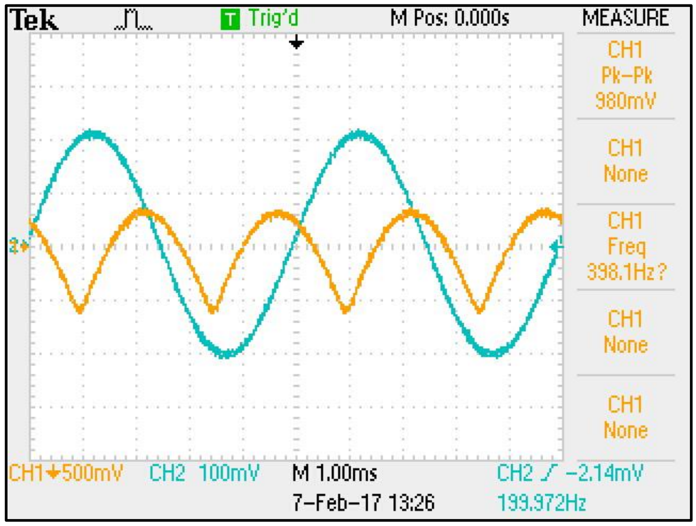
$$a_n = \frac{-1}{\pi}\left[\frac{\cos[(n+1)x]}{n+1}\right] + \frac{1}{\pi}\left[\frac{\cos[(n-1)x]}{n-1}\right]\ (Both\ limits\ between\ \pi\ \&\ 0)$$

$$a_n = -\frac{1}{\pi}[(-1)^{n+1}-1]\left[\frac{1}{n+1} - \frac{1}{n-1}\right] = \frac{2[(-1)^{n+1}-1]}{\pi(n^2-1)}$$

$$|\sin x| = \frac{2}{\pi} - \frac{4}{\pi}\sum_{m=1}^{\infty}\frac{\cos 2mx}{(4m^2-1)}$$

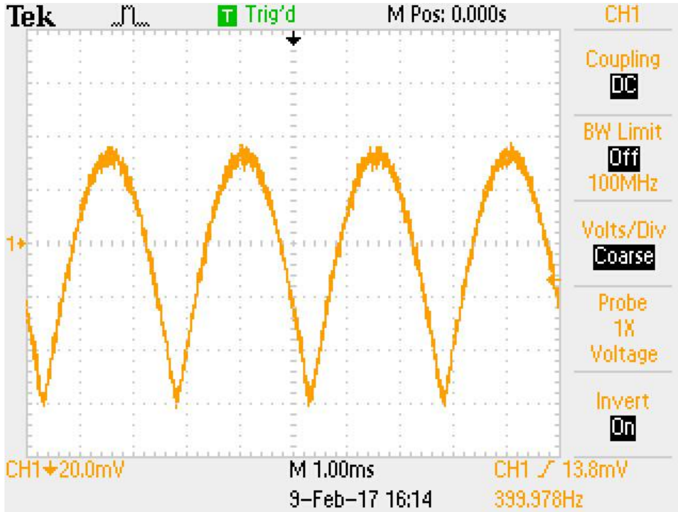
When these harmonics pass through the high pass filter at the output we remove the $\frac{2}{\pi}$ term once again indicating the rectified output must be centered at 0.

We can also see due to processing time that the output is delayed slightly from the input, output frequency has doubled as this wave has been rectified.

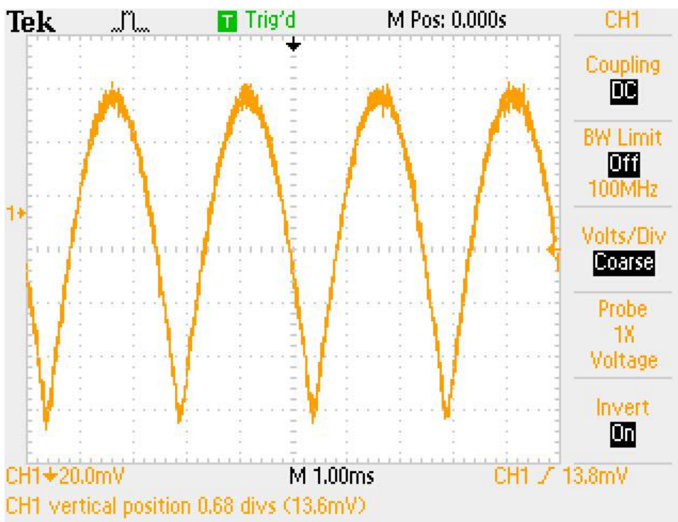


Trace 1: Traces shows a 200 Hz input (blue) alongside the 400 Hz output (orange), At 8 KHz sampling frequency.

Question 2



Trace 2: Shows the output of a 3.8 kHz sine wave, which shows a 400 Hz rectified sine wave. At 8 KHz sampling frequency.



Trace 3: Shows the output of a 200 Hz sine wave, which shows a 400 Hz rectified sine wave. At 8 KHz sampling frequency.

From Trace two and three we can observe that the two outputs have the same frequency. Additionally in order to find the frequency at the output for frequencies above 2 kHz we must use the equation. (The origins of the equation will be explained later on when comparing 200 Hz and 3.8 KHz)

$$2\left(\frac{F_{Samp}}{2} - Input\ Frequency\right).$$

To fully understand why the output waves have a similar frequency and shape it is important to observe these waves in the frequency domain.

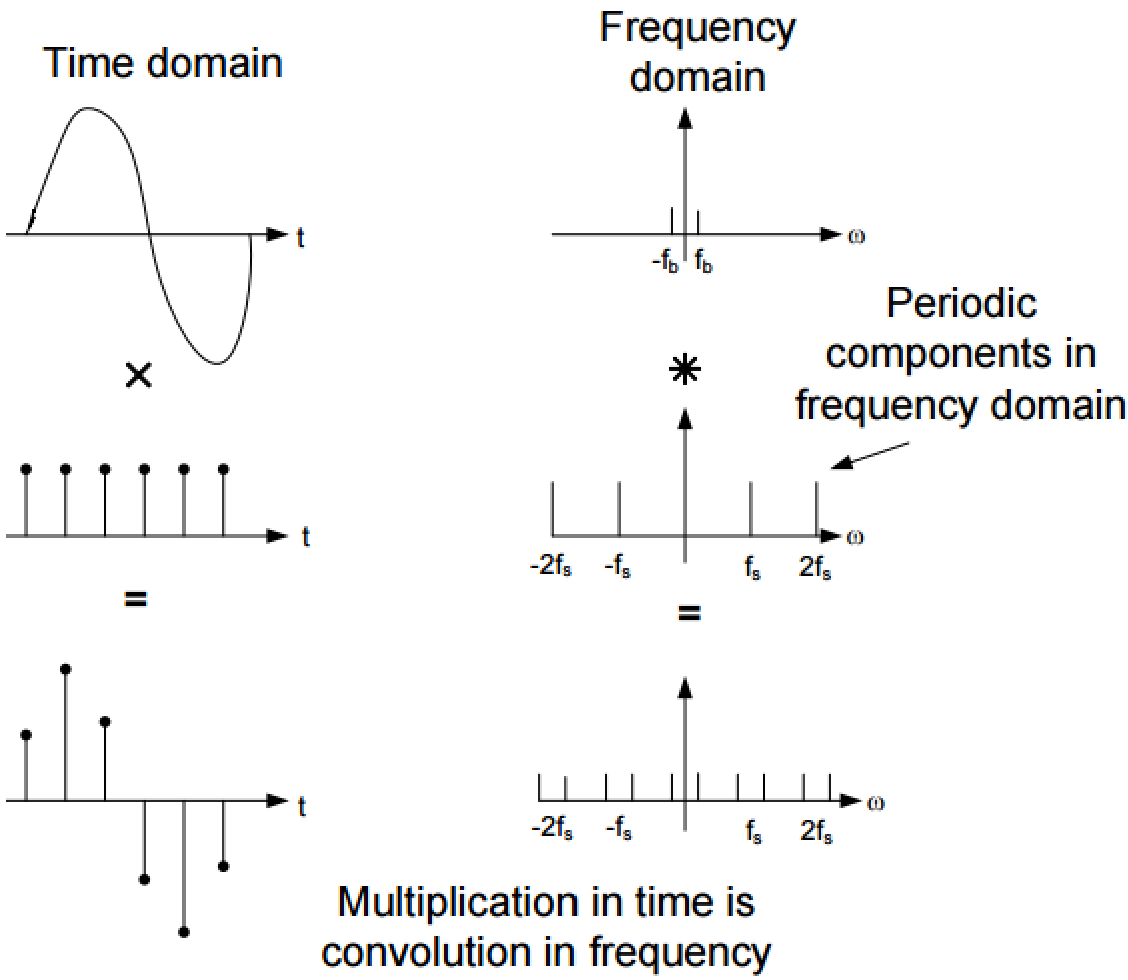


Figure 1: To illustrate how sampling a signal works in the time and frequency domain using convolution.

By using the result from before, the fourier series of a fully rectified sine wave :

$$|\sin t| = \frac{2}{\pi} - \frac{4}{\pi}\sum_{m=1}^{\infty}\frac{\cos 2mt}{(4m^2-1)}$$

We then find the Fourier transform of $\cos 2mt$ to plot the frequency spectrum of the full rectified sine wave.

$$\cos 2mt = \pi[\delta(w-2m) + \delta(w+2m)]$$

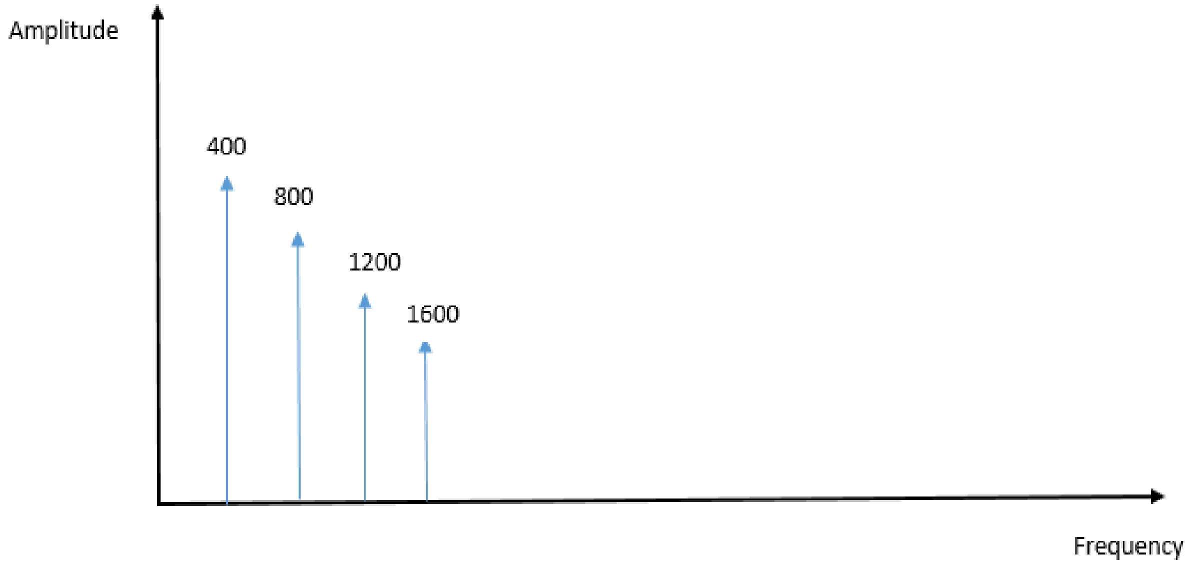


Figure 2: Shows the Fourier transform of the output when the input wave is a 200 Hz sine wave.

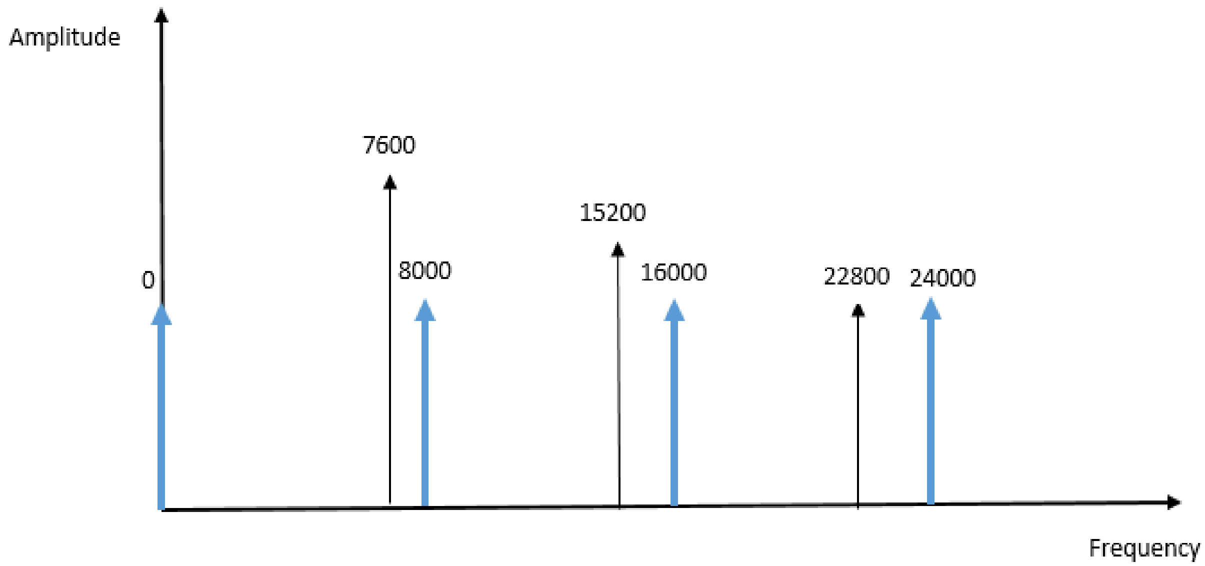


Figure 3: Shows the Fourier transform of the output (Black) when the input wave is a 3.8kHz sine wave & sampling signal (Blue).

Since when we multiple two signals in the time domain this corresponds to convolution in the frequency domain this causes the resultant Fourier transform.

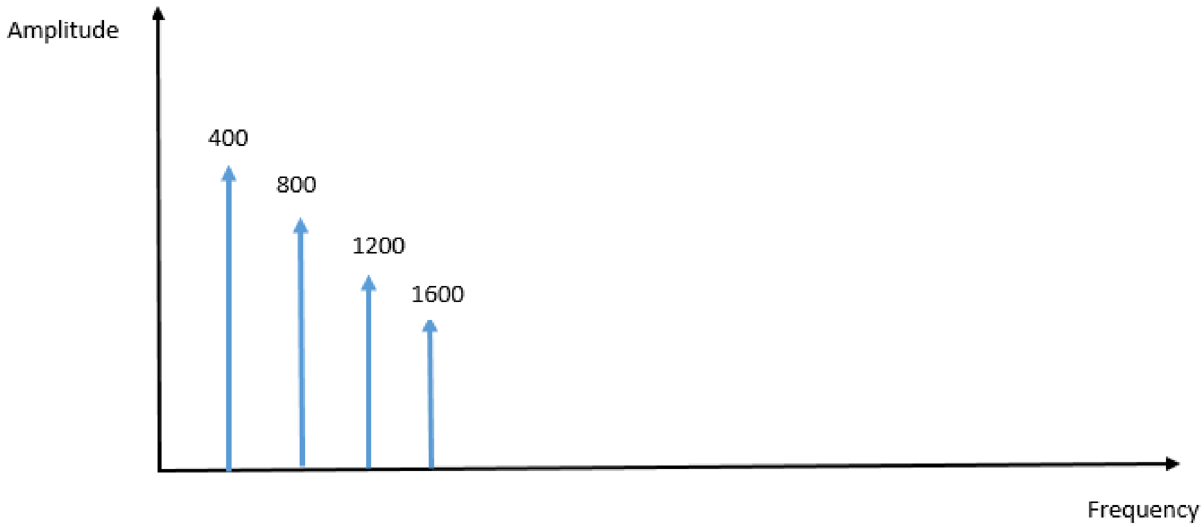


Figure 4: Shows the convolution between the output and sampling signal.

From the observations of figures two and four we can see the Fourier transform of the convoluted signal & the Fourier transform of the output of a 200Hz sine wave are similar thus this explains why the 2 waves have the same frequency and shape.

The 1 KHz to 3 KHz

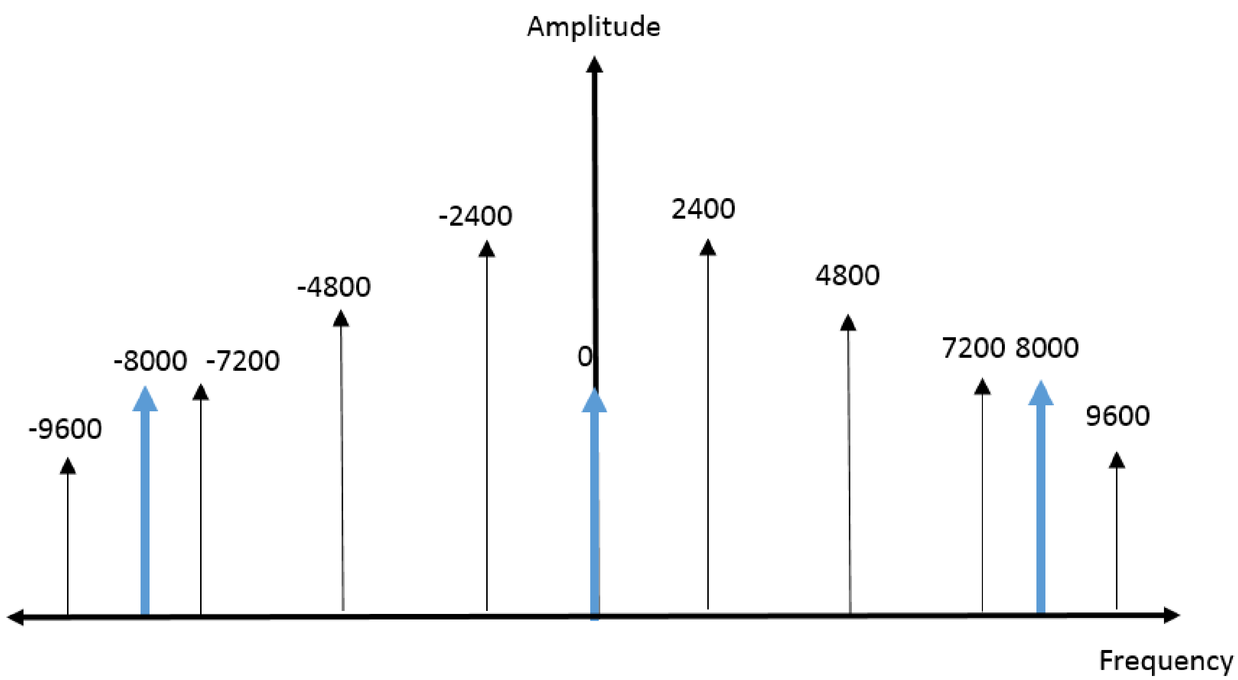


Figure 5: Show the output frequency spectrum of a 1.2 KHz input sine wave. We can see that there is clear symmetry. The blue arrows indicate the sampling frequency and the black indicates the harmonics of the output.

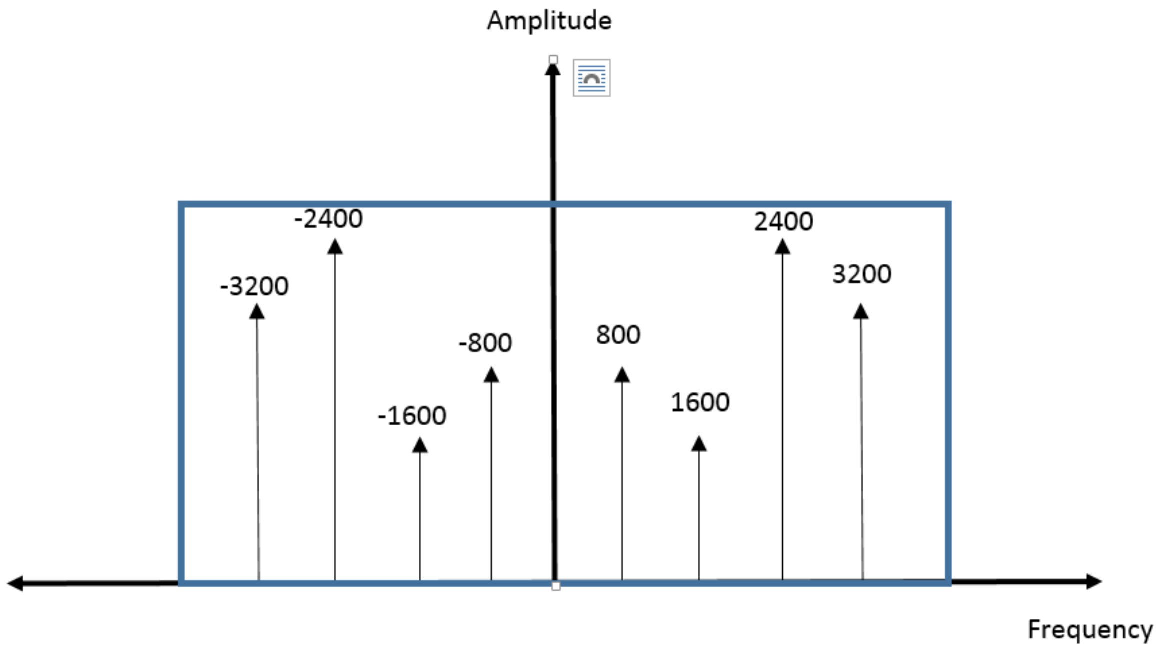
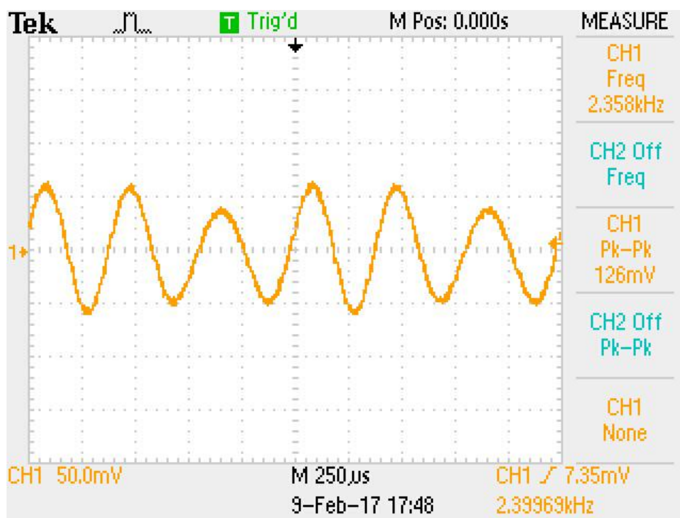


Figure 6: Shows the Anti – aliasing filter at half the sampling frequency, which is around the 4 KHz mark.



Trace 4: Shows a 2.4 KHz output wave from a 1.2 KHz at 8 Hz sampling frequency. From here we can clearly see that it is not a fully rectified waves.

From the frequency spectrum it shows that the output does not look like staircase formation similar to figure 4 thus the output from 1000-3000Hz does not look like a fully rectified sine wave instead it looks like a distorted waveform.

Summary

To conclude from our results the system only generates a full-wave rectified version of the system when the input frequency is between 0-1000Hz and 3000-4000Hz. Also, to note when the frequency is above 4000Hz, the output will simply be a shifted version of the system between 0-4000Hz however due to the properties of the anti-aliasing filter beyond 4000Hz, there will no output.

Exercises

In order to proceed with the exercises we first must understand and be able to explain all core initialisation steps.

```
void main(){


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

  /* initialize hardware interrupts */
  init_HWI();

  /* loop indefinitely, waiting for interrupts */

  while(1)
  { };

}
```

In our main function, we begin by initialising the hardware and hardware interrupts. It also consists on an infinite loop in order for the program to run indefinitely so we can make best use of our future interrupt functions.

```
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);


}
```

Within the initialising hardware function we must first call DSK6713_init (); which ensure all hardware is running at default settings.

We then call H_Codec = DSK6713_AIC23_openCodec(0, &Config) which essentially configures all hardware, more importantly it sets the audio ports bit resolution and sampling frequency which will later on be crucial for both exercises as they use the audio chip and the pre-defined function of mono_read_16Bit( ) and mono_write_16Bit( ).

The four MCBSP_FSETS functions are used to set up the MCBSP ports allowing communication between the AIC23 audio port and the DSP. These functions are pre-defined.

We conceptually think of these predefined functions as black box's, we know their operation but don't know the inner workings.

We then begin to configure the interrupt settings through 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

}
```

The pre-defined function IRQ_globalDisable ( ) is critical in avoiding interrupt corruption due to interrupting critical sections of code, which are essentially sections of code that must have complete and undisturbed access to a block of data. Technically we only need to disable certain interrupts that could interrupt a critical section, but IRQ_globalDisable ( ) takes a heavy handed solution and disables all interrupts which is a lot simpler to implement. This does ultimately make the entire system less responsive than it should be, however in this lab its effects will be negligible.

Exercise 1

The task required for exercise was to read a sample from the codec, these would be samples of a digital signal generator on a Windows computer then fully rectify the wave and then send out the samples to the left and right channels all within an interrupt function.

```c
void Audio_ISR (void){

    /* Takes a sample from the digital signal generator into a 16 bit short variable,
       to ensure we get a recified signal we take the absolute vale of each sample.
       It is then pass on to Mono_write where we output the rectified signal   */

    short read_sample = abs( mono_read_16Bit());

    mono_write_16Bit(read_sample);

}
```

Figure 7: Shows the interrupt service routine that reads a sample, rectifies and sends a copy to the output.

Audio_ISR is the interrupt function we used to implement this function. Very few lines of code is required to execute the required task. The essential part of the ISR is the two special functions **mono_write_16Bit** and **mono_read_16Bit**. **Mono_read_16Bit** provides a mono input by reading left and right samples from an audio port, divides by two and sums them up. The result is then returned as a 16 bit integer.

We store this mono result in a short variable named read_sample. Short is used as its 16 bit signed allowing us to store the returned mono sample. To ensure we have fully rectified the input signal we have to make all negative value samples from the input turn into positives at the same magnitude. This can be easily done by simply just finding the absolute value using the **abs ()** function.

Read_sample is then passed through **mono_write_16Bit** where in which it sends the rectified samples to left and right output channels to achieve a mono output.

Exercise 2

In this exercise we are given the task of fusing sections of lab 2 into lab 3 where in which we are required to use a interurupt service routine to generate a sine-wave using a look up table. Here is how we impletement this.

Firstly we have to fill a look up table with one cycle of a sin wave. This is done within the main which calls a function **sine_init ( )**.

```
void main(){


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

   sine_init(); //fill the look up table with sine wave samples

  /* initialize hardware interrupts */
  init_HWI();

  /* loop indefinitely, waiting for interrupts */

  while(1)
  { };

}
```

Figure 8 : Shows the main function, hardware interrupts are initalised along side filling the look up table with a one cycle sinwave.

Within this function we are filling a look-up table with samples of a one cycle sin wave. The size of the look up table is of size 256 hence a For loop is required to cycle through the entire table with each array block being filled by a sample from equation: $\frac{\sin 2i\pi}{256}$. Here is a example of how it visually it does this.

I	62	63	64	65	66......
$\frac{\sin 2i\pi}{256}$	0.9987954	0.9996988	1.0	0.9996988	0. 9987954
Table[i]					

These values again can be seen from the 'watch' window on the code composer studio.

Name	Value	Address	Type	Format
(x)= [61]	0.9972904	0x000099BC	float	Natural
(x)= [62]	0.9987954	0x000099C0	float	Natural
(x)= [63]	0.9996988	0x000099C4	float	Natural
(x)= [64]	1.0	0x000099C8	float	Natural
(x)= [65]	0.9996988	0x000099CC	float	Natural
(x)= [66]	0.9987954	0x000099D0	float	Natural
(x)= [67]	0.9972904	0x000099D4	float	Natural
(x)= [68]	0.9951847	0x000099D8	float	Natural
(x)= [69]	0.9924796	0x000099DC	float	Natural
(x)= [70]	0.9891765	0x000099E0	float	Natural
(x)= [71]	0.9852777	0x000099E4	float	Natural

Figure 9 : Shows a snippet of ten values [index from 61 – 71] within the lookup table.

```
void sine_init(void) // initialises all values on the look up table. i.e generates a whole cycle of a sine wave.
{
    int i; // counter variable for the For loop below.

    for (i = 0 ; i < SINE_TABLE_SIZE ; i++) /*Size of For loop is limited to the size of the look up table, to
                                              ensure there are no redudant values.*/
    {
        table[i] = sin((2*i*PI)/SINE_TABLE_SIZE); // Fills the entire look up table with 256 sample of
              //                                       one whole cycle of a sin wave. */
    }
}
```

Figure 10 : Shows the function that implements and fills a look up table of size 256 with samples of one cycle of a sine wave

Further to this, to actually generate a sine wave from the look up table we have to edit the interrupt service routine to give a rectified output signal.

```
void Audio_ISR (void){

    realstepsize += ((sine_freq*SINE_TABLE_SIZE)/(sampling_freq)); // determines the number of samples skipped in

    indexcounter = (realstepsize < 0 ? (realstepsize - 0.5 ) : (realstepsize) + 0.5); /*Since the index of the table only takes integer values
                                                                                        we have to do rounding on the index for certain frequencyies
                                                                                        in order to get accurate output                     */
    if (indexcounter >= SINE_TABLE_SIZE) /* if condition ensures the index doesn't exceed the
                                            size of the look up table for the next sample reading.*/
    {
    realstepsize = realstepsize - SINE_TABLE_SIZE;  //^^
    indexcounter = indexcounter- SINE_TABLE_SIZE;   //^^
    }

    mono_write_16Bit(abs((short)floor(table [indexcounter] * 32767)));  // Reduces float values in table to shorts and multiplies by 32767 which is the heighest
                                                                        // in a short in order to display on ossciliscope. We also take the absolute value
                                                                        // in orderto rectify the signal

}
```

Figure 11: Shows the ISR that will read from the look up table, rectifies and does a mono write.

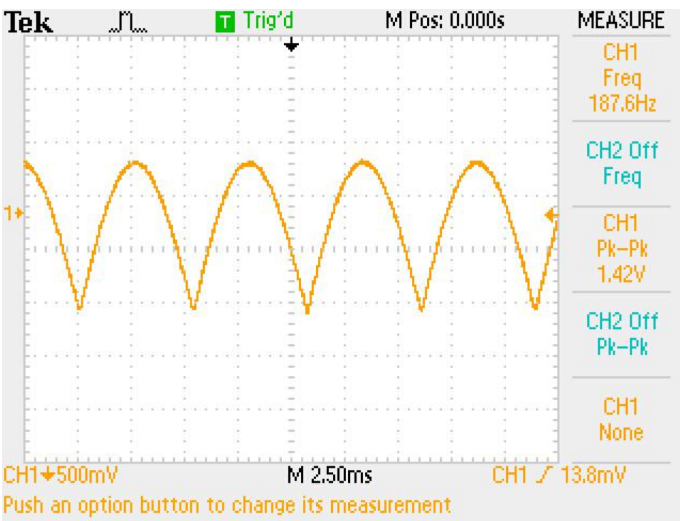
In order to get our desired frequency we have to determine how many sample we have to skip. To do this we use equation:

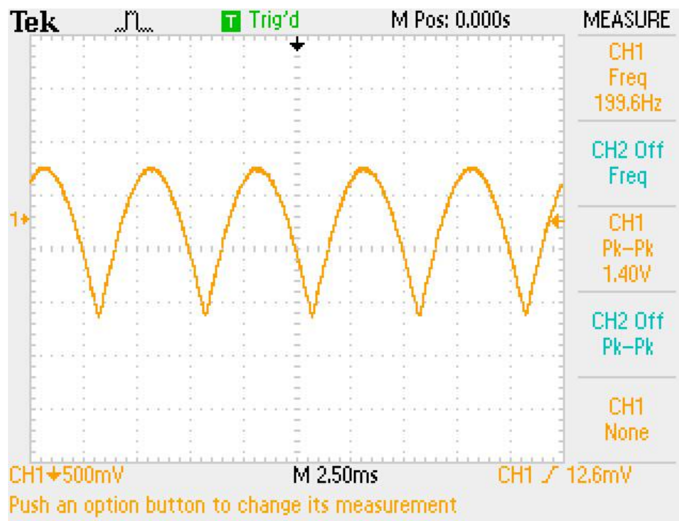
$$\frac{Sampling\ freq\ *\ skipped\ samples}{Sine\_table\_size} = Sine\ frequency$$

$$\therefore \frac{Sine\ frequency * Sine\_table\_size}{Sampling\ freq} = skipped\ samples$$

This has been implemented with line 'realstepsize += ((sine_freq*SINE_TABLE_SIZE)/(sampling_freq))' which equates the global variable **'realstepsize'** of type float to a certain sample of the one cycle sine wave. Furthermore due to the **'+='** we increment **'realstepsize'** by the amount we need to skip by determined by the equation above.

The reason we use type float for **'realstepsize'** is because for certain frequencies we need to skip in decimal numbers. An example would be at a 100 Hz where, using the above equation we get $\frac{100*256}{8000} = 3.2$ skipping steps. If integer was used the numbers after the decimal would be ignored, i.e in this case 0.2 would be ignored and would just skip 3. This becomes a problematic issue if say the number we need to skip is 3.9, in integer form we again would just skip 3, resulting in us not always getting our desired frequency. To solve this we used **' indexcounter = (realstepsize < 0 ? (realstepsize - 0.5) : (realstepsize) + 0.5);'** which does this rounding function.

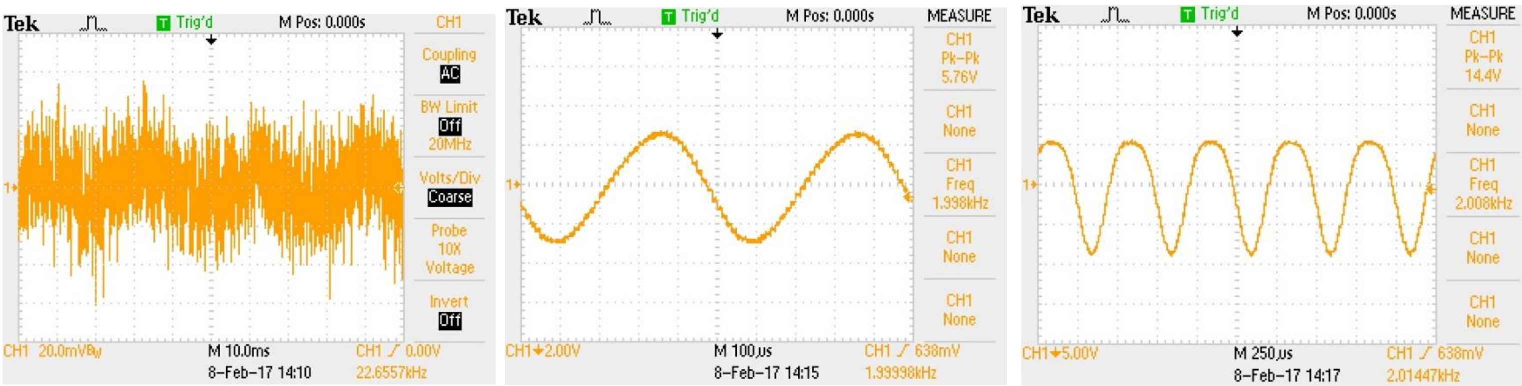




Traces 4,5: Show how the rounding feature results in more accurate outputs at 100 Hz input at 8 KHz sampling frequency. The left trace does not use the rounding function whereas the right trace does indeed use the rounding function.

To ensure that the index counter always remains between 0 – 255 and not extend the reach of the table size we run a condition. The If loop checks if the **'indexcounter'** and **'realstepsize'** is larger than the table size of 256, if it is the we subtract the size value of the look up table in order for the **'indexcounter'** and **'realstepsize'** to keep incrementing in our desired skip size.

When we go ahead to write the sample, we are doing several things all at once. Firstly we have to take note that the look up table is of type float which is 32 Bit signed but the pre-defined function **mono_write_16Bit** can only take a copy of a 16 bit value. Hence there is a compatibility issue. To solve this issue we use function **(short) floor(…)** which reduces a value from the table to a 16 bit short solving the compatibility issue. However as the values from the look up table are too small to be seen on an oscilloscope we have to have some gain. In this case we multiply by 32767 which is the max number of a 16 bit signed value.

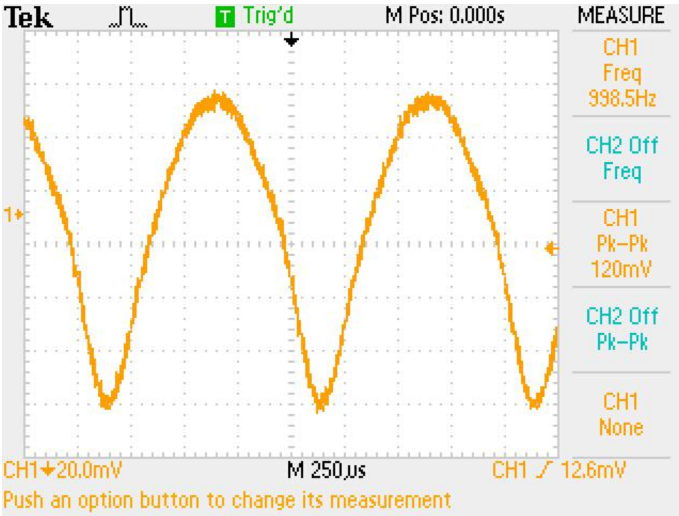


Figures 12: Show how multiplying by a gain makes the signal clear to read of an oscilloscope. Taken at 1000 Hz

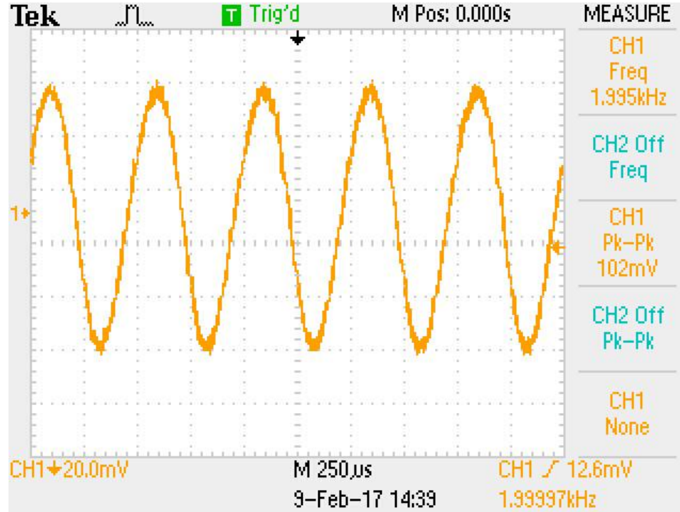
Furthermore to get a rectified output signal we take the absolute value of read samples just in the same fashion we did in exercise 1. We could have done this rectification at other stages also in the code. The Function **mono_write_16Bit** then sends copies to the left and right sections channels.

Traces

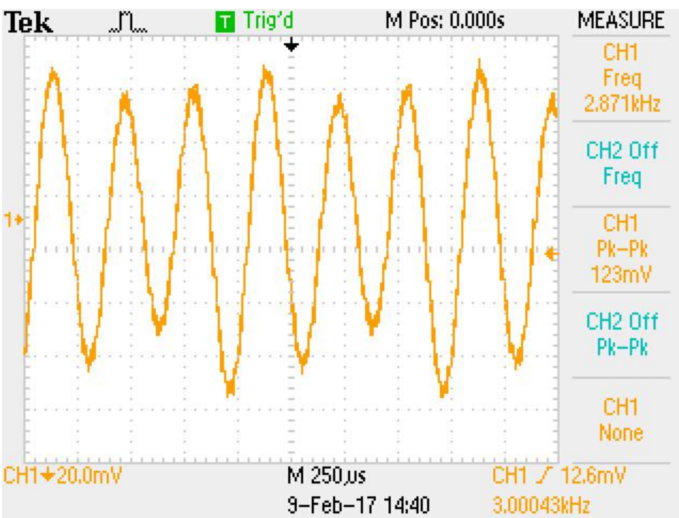
Exercise 1:



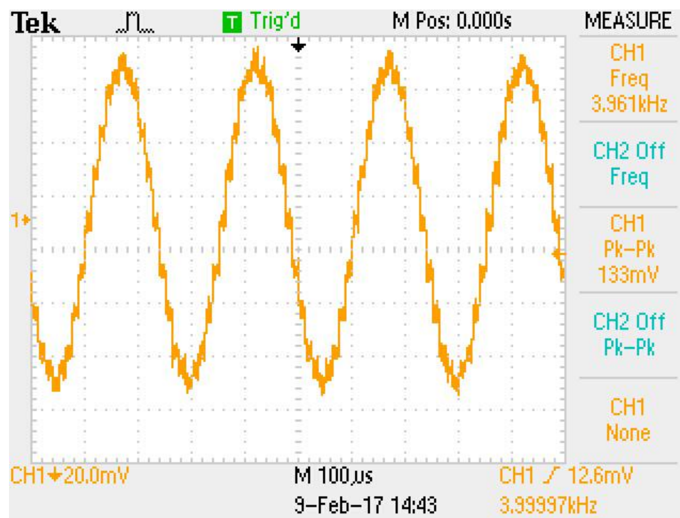
Shows a 1000 Hz output from a 500 Hz input generated by a digital signal generator



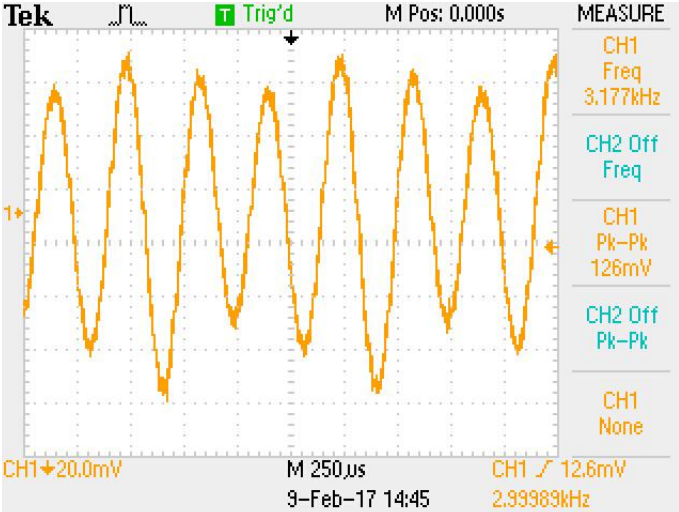
Shows a 2000 Hz output from a 1000 Hz input generated by a digital signal generator



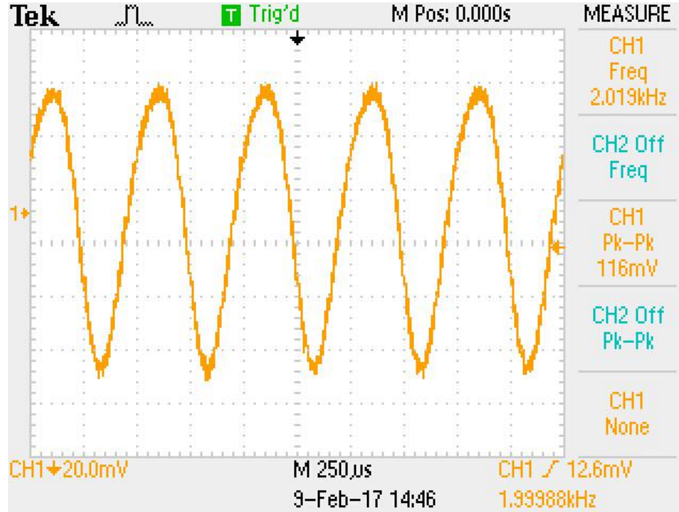
Shows a 3000 Hz output from a 1500 Hz input generated by a digital signal generator



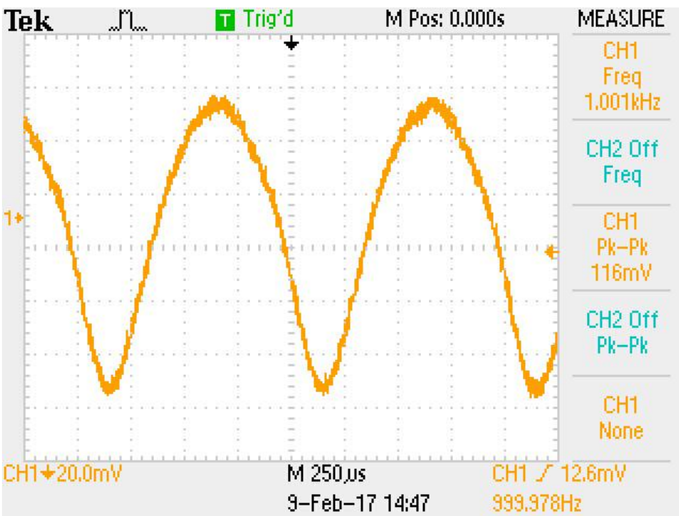
Shows a 4000 Hz output from a 2000 Hz input generated by a digital signal generator



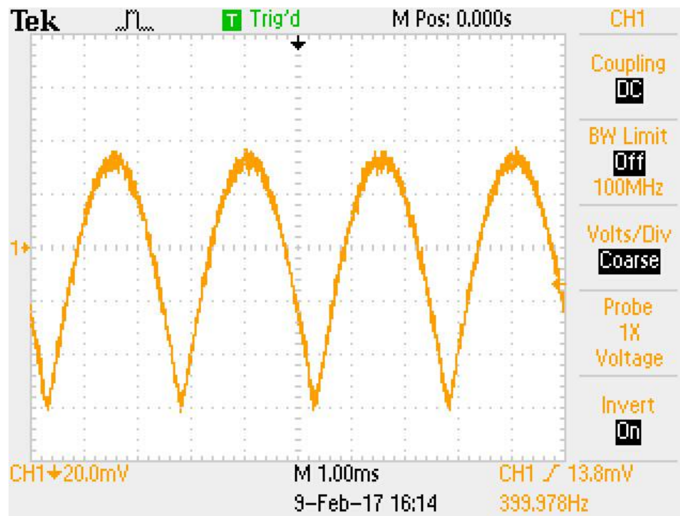
Shows a 3000 Hz output from a 2500 Hz input generated by a digital signal generator



Shows a 2000 Hz output from a 3000 Hz input generated by a digital signal generator

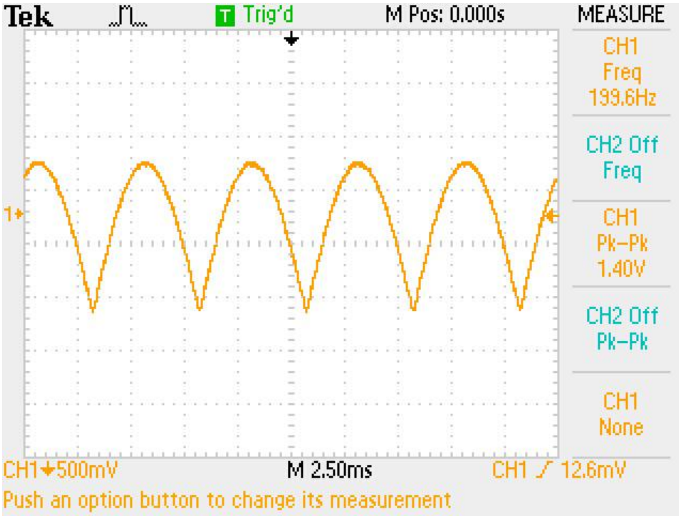


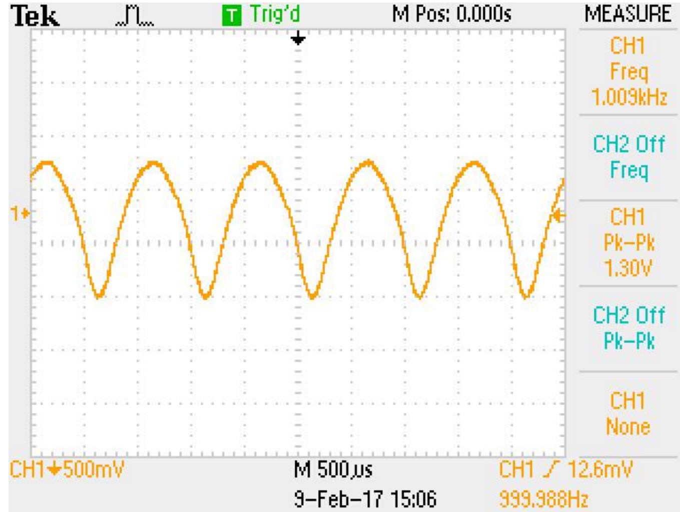
Shows a 1000 Hz output from a 3500 Hz input generated by a digital signal generator



Shows a 400 Hz output from a 3800 Hz input generated by a digital signal generator

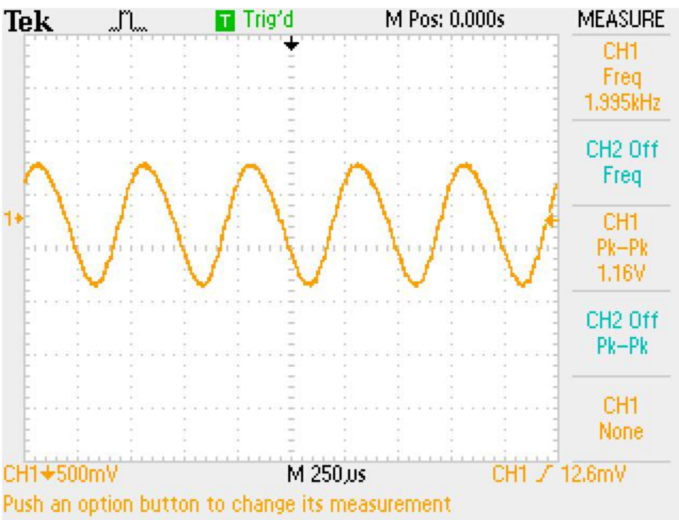
Exercise 2

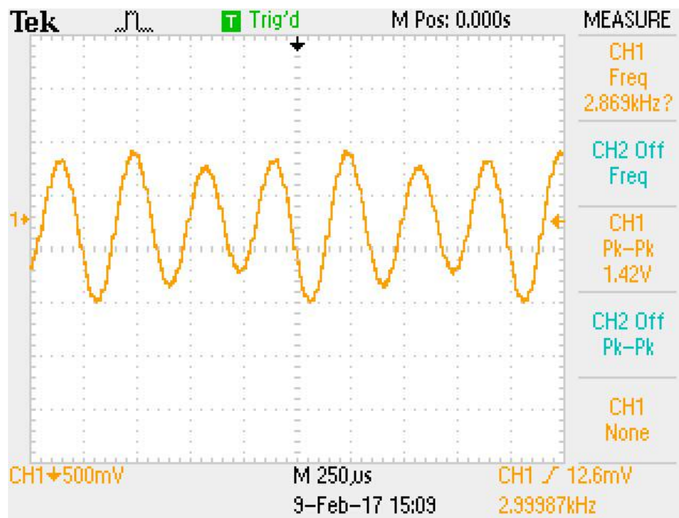




Shows a 200 Hz output from a 100 Hz input generated by a digital signal generator

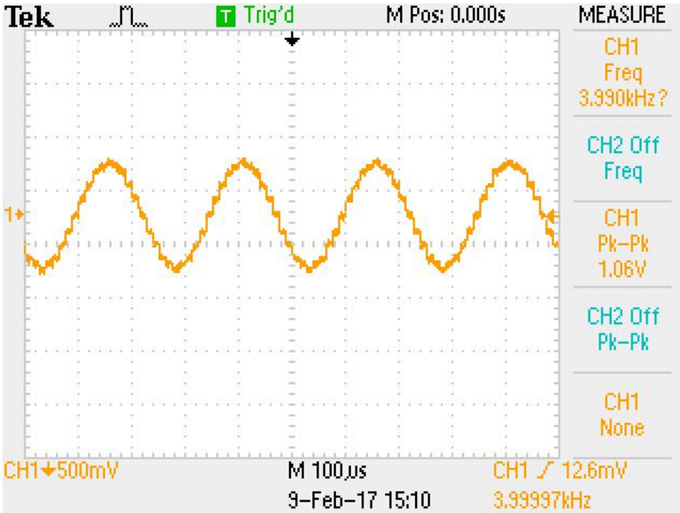
Shows a 1000 Hz output from a 500 Hz input generated by a digital signal generator



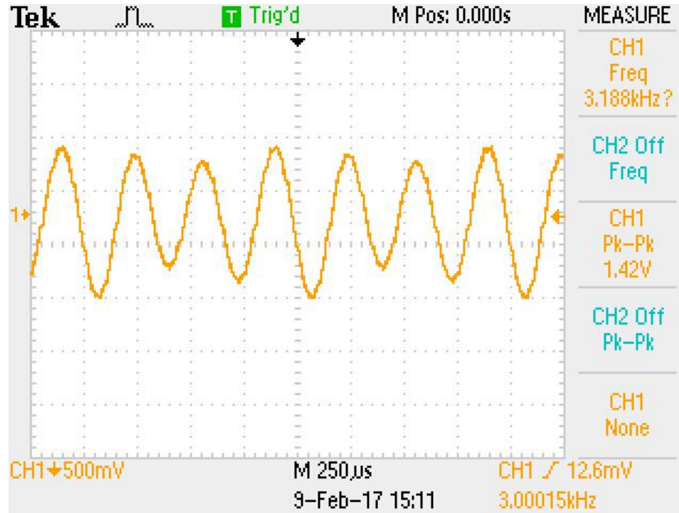


Shows a 2000 Hz output from a 1000 Hz input generated by a digital signal generator

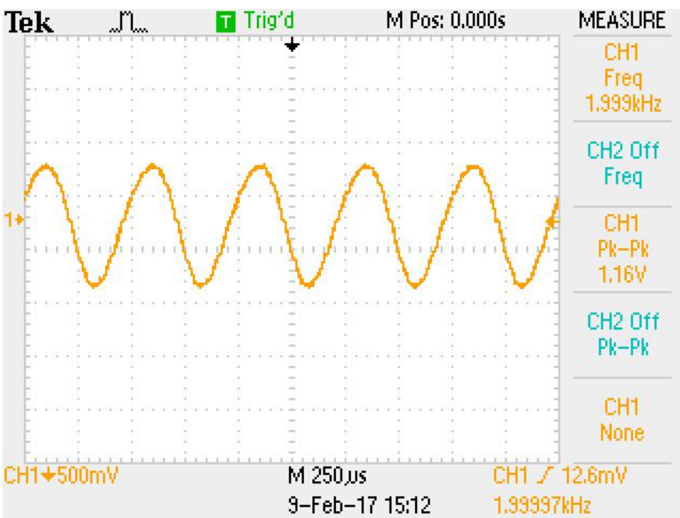
Shows a 3000 Hz output from a 1500 Hz input generated by a digital signal generator



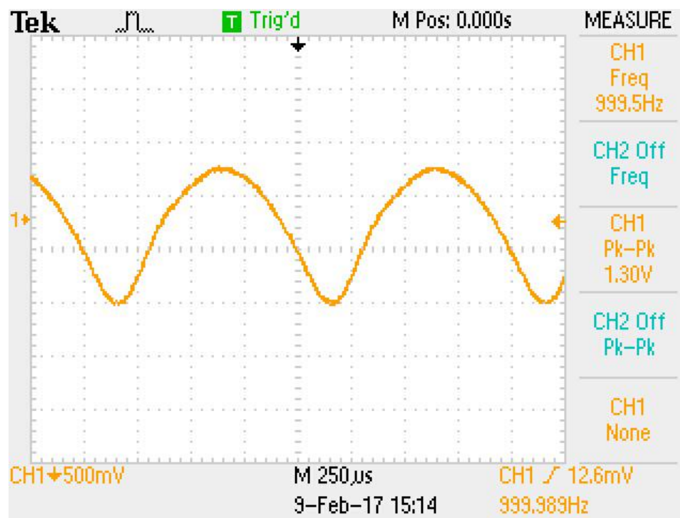
Shows a 4000 Hz output from a 2000 Hz input generated by a digital signal generator



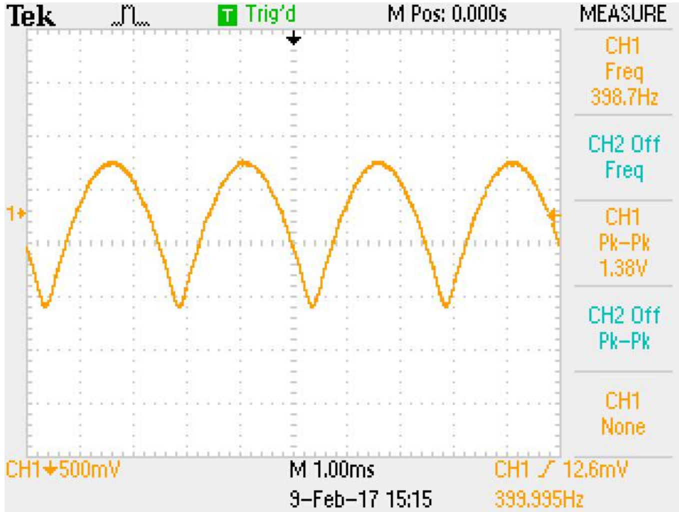
Shows a 3000 Hz output from a 2500 Hz input generated by a digital signal generator



Shows a 2000 Hz output from a 3000 Hz input generated by a digital signal generator



Shows a 1000 Hz output from a 3500 Hz input generated by a digital signal generator



Shows a 400 Hz output from a 3800 Hz input generated by a digital signal generator

Appendix : Exercise 1 complete code:

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

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

                                 LAB 3: Interrupt I/O

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

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

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

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

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

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

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

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

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
             /**********************************************************************/
             /*   REGISTER             FUNCTION                      SETTINGS         */
             /**********************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                   */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                   */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                   */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                   */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on       */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ                 */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                    */\
```

```
        /*******************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

 /******************************** Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
void Audio_ISR (void); // ISR function

/********************************** 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() **************************************/
```

```c
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 Audio_ISR (void){

    Short read_sample = abs (mono_read_16Bit()); // reads a sample and takes its absolute value in order to rectify the wave.

    mono_write_16Bit(read_sample);                  //sends copies of smaples to left and right channels.

}
```

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

                               ********* 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>

// PI defined here for use in your code
#define PI 3.141592653589793
#define SINE_TABLE_SIZE 256

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

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
             /**********************************************************************/
             /*   REGISTER             FUNCTION                      SETTINGS         */
             /**********************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                   */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                   */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                   */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                   */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
```

```
    0x0000,  /* 6 DPOWERDOWN Power down control           All Hardware on       */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit             */\
    0x008d,  /* 8 SAMPLERATE Sample rate control           8 KHZ                 */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                 */\
             /*************************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

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

// Holds the value of the current sample
float sample;

/* 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;
float table [SINE_TABLE_SIZE];
int indexcounter = 0;
float realstepsize=0;

 /******************************** Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
void Audio_ISR (void);
void sine_init(void);

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


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

    sine_init(); //fill the look up table with sine wave samples

  /* 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
```

```c
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

    /* Configures interrupt to activate on each consecutive available 32 bits
    from Audio port hence an interrupt is generated for each L & R sample pair */
    MCBSP_FSETS(SPCR1, RINTM, FRM);

    /* These commands do the same thing as above but applied to data transfers to
    the audio port */
    MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
    MCBSP_FSETS(SPCR1, XINTM, FRM);


}

/*********************************** init_HWI() **************************************/
void init_HWI(void)
{
    IRQ_globalDisable();            // Globally disables interrupts
    IRQ_nmiEnable();                // Enables the NMI interrupt (used by the debugger)
    IRQ_map(IRQ_EVT_XINT1,4);       // Maps an event to a physical interrupt
    IRQ_enable(IRQ_EVT_XINT1);      // Enables the event
    IRQ_globalEnable();             // Globally enables interrupts

}

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

void Audio_ISR (void){

    realstepsize += ((sine_freq*SINE_TABLE_SIZE)/(sampling_freq)); // determines the number of samples skipped in

    indexcounter = (realstepsize < 0 ? (realstepsize - 0.5 ) : (realstepsize) + 0.5); /*Since the index of the table only takes integer values
                                                                                        we have to do rounding on the index for certain frequencyies
                                                                                        in order to get accurate output                      */
    if (indexcounter >= SINE_TABLE_SIZE) /* if condition ensures the index doesn't exceed the
                                            size of the look up table for the next sample reading.*/
    {
    realstepsize = realstepsize - SINE_TABLE_SIZE;  //^^
    indexcounter = indexcounter- SINE_TABLE_SIZE;   //^^
    }

    mono_write_16Bit(abs((short)floor(table [indexcounter] * 32767)));  // Reduces float values in table to shorts and multiplies by 32767 which is the heighest
                                                                         // in a short in order to display on ossciliscope. We also take the absolute value
                                                                         // in orderto rectify the signal

}
 void sine_init(void) // initialises all values on the look up table. i.e generates a whole cycle of a sine wave.
{
    int i; // counter variable for the For loop below.

    for (i = 0 ; i < SINE_TABLE_SIZE ; i++) /*Size of For loop is limited to the size of the look up table, to
                                              ensure there are no redudant values.*/
    {
        table[i] = sin((2*i*PI)/SINE_TABLE_SIZE); // Fills the entire look up table with 256 sample of
                //                                   one whole cycle of a sin wave. */
    }
}
```

(All snippets is in high resolution, please zoom in if its appears too small)