

EE3-19 Real Time Digital Signal Processing

Coursework Report 2: Lab 3

Jeremy Chan*

Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
jc4913@ic.ac.uk | 00818433

Chak Yeung Dominic Kwok*

Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
cyk113@ic.ac.uk | 00827832

*These authors contributed equally to this work

Table of Contents

I.	Introduction	3
II.	Questions.....	3
A.	Why is the full rectified waveform centred around 0V and not always above 0V as you may have been expecting?	3
B.	Note that the output waveform will only be a full-wave rectified version of the input if the input from the signal generator is below a certain frequency. Why is this?	3
C.	What kind of output do you see when you put in a sine wave at around 3.8KHz? Can you explain what is going on?	6
III.	Operation of Code.....	8
A.	Pre-processor statements	8
B.	Global variables/declarations.....	9
C.	Initialisation Functions	9
D.	Main Testbench.....	9
E.	Sinegen functions	9
F.	Interrupt Function.....	9
G.	Exercise 1: Interrupt Service Routine	10
H.	Exercise 2: Interrupt Driven Sine Wave	10
IV.	Oscilloscope Traces.....	11
A.	Exercise 1	11
B.	Exercise 2	13
V.	Discussion on Oscilloscope Graphs.....	15
VI.	Conclusion	16
VII.	References.....	16
VIII.	Appendix.....	17
A.	High Pass Filter on input/output of AIC23 [1]	17
B.	Code.....	17

I. INTRODUCTION

A skeleton code for configuring the hardware was provided – modifications were made to implement a function that would be called by hardware interrupts, as well as linking the previous sine wave generation functions to that function. Oscilloscope traces as well as corresponding MATLAB traces are shown for multiple frequencies to prove the operation of the code, along with code explanations in both the main text and the appendix.

II. QUESTIONS

A. *Why is the full rectified waveform centred around 0V and not always above 0V as you may have been expecting?*

From Appendix.A, there are high pass filters on both the input and output sides of the AIC23 Audio Chip. These high pass filters attenuate the DC component of the rectified output. Hence, the expected DC offset to show the sine wave centred above 0V is removed; the resulting waveform is shifted negatively and becomes centred around 0V instead.

B. *Note that the output waveform will only be a full-wave rectified version of the input if the input from the signal generator is below a certain frequency. Why is this?*

Our hardware has the following configuration:



Fig. 1. Configuration of hardware

Given an f_{in} , the expected output frequency is $2f_{in}$, and can be proved as follows:

$$\begin{aligned}
 x(t) &= \sin(2\pi f_{in}t) \\
 y(t) &= \text{Full Wave Rectified } x(t) \\
 &= \text{abs}(\sin(2\pi f_{in}t)) \\
 \text{Periodic signal property: } y(t + T') &= \frac{y(t)}{x(t + T)} \\
 &= x(t) \\
 \therefore T &= \frac{1}{f}, \text{ let } T' = \frac{1}{2f_{in}} \\
 y\left(t + \frac{1}{2f_{in}}\right) &= \text{abs}\left(\sin\left(2\pi f_{in}\left(t + \frac{1}{2f_{in}}\right)\right)\right) \\
 &= \text{abs}(\sin(2\pi f_{in}(t + \pi))) \\
 &= \text{abs}(-\sin(2\pi f_{in}t)) \\
 &= |\sin(2\pi f_{in}t)| \\
 &= y(t)
 \end{aligned}$$

Therefore, $y(t)$ is periodic with period $\frac{1}{2f_{in}}$ and frequency $2f_{in}$, where f is the input signal frequency.

Let f_{proc} be the realized input frequency between the ADC and the DAC. To achieve a full-wave rectification of a sine wave, $f_{in} \bmod f_{s_{ADC}} \leq \frac{f_{s_{ADC}}}{2}$ has to be satisfied at input of the ADC. If so, $f_{proc} = f_{in} \bmod f_{s_{ADC}}$. Otherwise, $f_{proc} = f_{s_{ADC}} - f_{in} \bmod f_{s_{ADC}}$.

At the DAC, $2f_{proc} \leq \frac{f_{s_{DAC}}}{2}$ also has to be satisfied. If so, $f_{out} = 2f_{proc}$. Otherwise, $f_{out} = f_{s_{DAC}} - 2f_{proc}$.

The following truth table shows different input/output frequency relationships.

TABLE I.

	$2f_{proc} \leq f_{sDAC}$	$2f_{proc} > f_{sDAC}$
$f_{in} \bmod f_{sADC} \leq \frac{f_{sADC}}{2}$	$f_{out} = 2f_{in} \bmod f_{sADC}$	$f_{out} = f_{sDAC} - 2f_{in} \bmod f_{sADC}$
$f_{in} \bmod f_{sADC} > \frac{f_{sADC}}{2}$	$f_{out} = 2(f_{sADC} - f_{in} \bmod f_{sADC})$	$f_{out} = f_{sDAC} - 2(f_{sADC} - f_{in} \bmod f_{sADC})$

In conclusion, there are two criteria, $2f_{proc} \leq \frac{f_{sDAC}}{2}$, and $f_{in} \bmod f_{sADC} \leq \frac{f_{sADC}}{2}$, which have to be fulfilled the function of full-wave-rectification to be correct.

In this exercise, $f_{sADC} = 8000\text{Hz}$, $f_{sDAC} = 8000\text{Hz}$, $f_{in} \leq 4000\text{Hz}$, $f_{proc} \leq 2000\text{Hz}$. Hence, both the conditions are satisfied when $f_{in} \leq 2000\text{Hz}$.

However, in practice, the waveform is amplitude distorted around this theoretical limit. In Figure 2

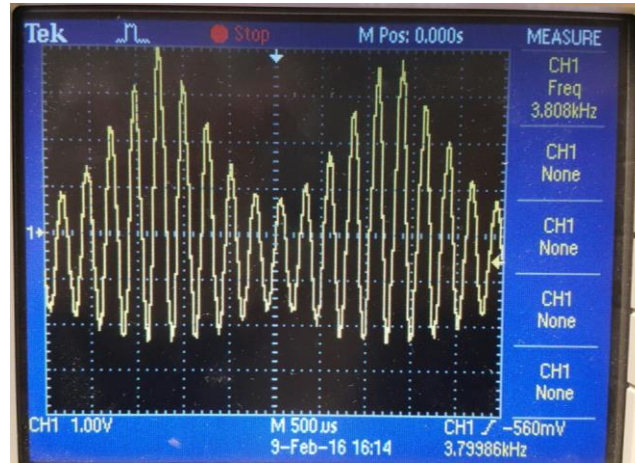


Fig. 2. Oscilloscope Output of 3800Hz (Input 1900Hz)

From the oscilloscope, the distortion frequency is 400Hz. This can be shown in the FFT, where there are pairs of peaks 400Hz apart, and again in using MATLAB to plot the sampled frequency spectrum of $f(t)$. Note the pairs of peaks 400Hz apart contributing to amplitude distortion. This amplitude distortion is due to the superposition of the frequency spectrum of $f(t)$ and its sampled spectrum. Since the Fourier series of a full-rectified sine wave has infinitely many harmonics, and as the sampled spectrum will be a shifted version of the original spectrum, sampling will cause samples to overlap. Hence, there will be extra frequency components in the passband and summation with unwanted sinusoidal signals in the time domain. As a result, the full rectified waveform is distorted.

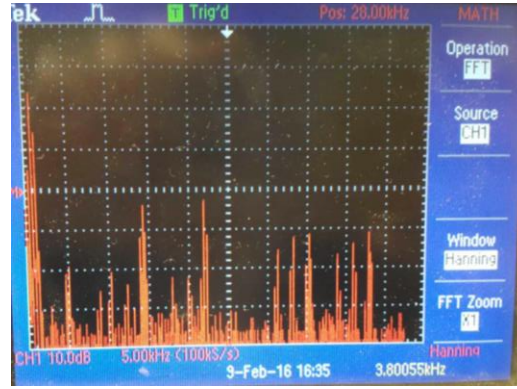


Fig. 3. Oscilloscope FFT of 3800Hz (Input 1900Hz)

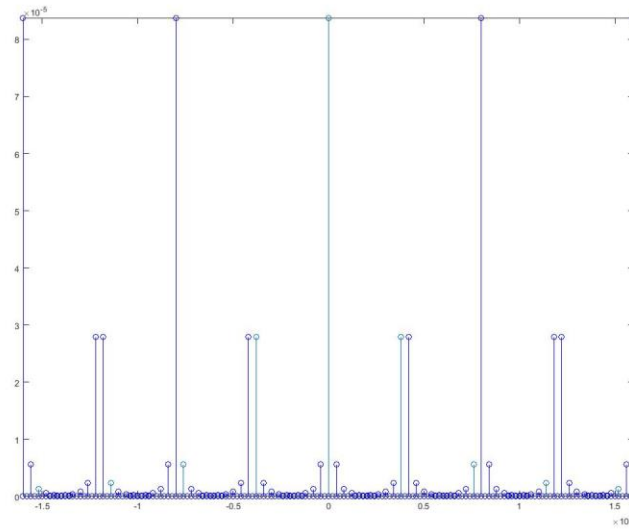


Fig. 4. MATLAB sampled frequency spectrum (Input 1900Hz)

This amplitude distortion effect is visible when the harmonics of the original spectrum and the sampled spectrum do not align perfectly. In practice, this happens when the f_{out} is not a factor of f_s . To ensure the shape of the rectified sine wave, there has to be at least 4 harmonics in the passband of the low pass filter. From the following two figures, the 4 component input of 500Hz produces a better full-wave rectification than the 3 component input of 800Hz; it looks more like a regular sine wave.

To calculate the number of components in the passband, the following formula can be used: $\left\lceil \frac{f_{SDAC}}{2f_{in}} \right\rceil$, assuming $f_{in} \leq \frac{f_{sADC}}{2}$. Hence, to achieve the closest full-wave rectified output, the input frequency should be below 500Hz, and f_{SDAC} should be divisible by $2f_{in}$.



Fig. 5. Oscilloscope Output of 1600Hz (Input 800Hz)

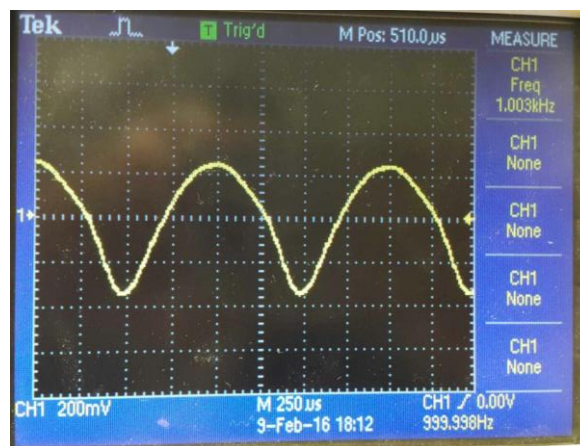


Fig. 6. Oscilloscope Output of 1000Hz (Input 500Hz)

C. What kind of output do you see when you put in a sine wave at around 3.8KHz? Can you explain what is going on?



Fig. 7. Oscilloscope Output of 400Hz (Input 3800Hz)

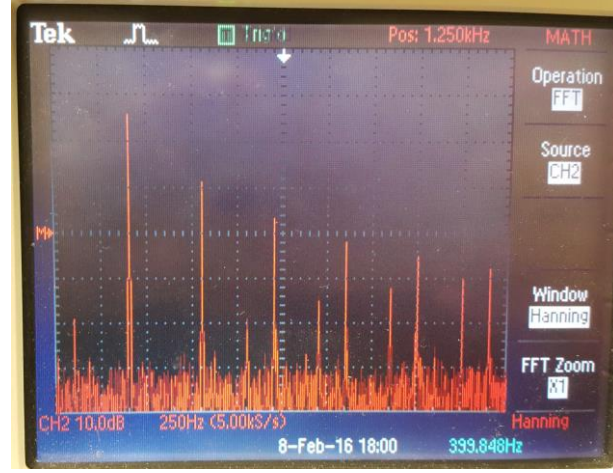


Fig. 8. FFT of output (Input 3800Hz)

Figure 7 shows $f_s = 8000\text{Hz}$, $f_{in} = 3800\text{Hz}$. Hence, $f_{in} \leq \frac{f_{s_{ADC}}}{2}$, $2f_{proc} > f_{s_{DAC}}$ are both satisfied. From Table I, $f_{out} = f_{s_{DAC}} - 2f_{in} \bmod f_s = 400\text{Hz}$.

Figure 8 is the oscilloscope FFT with the above frequencies. This frequency spectra can be explained by computing the Fourier series of a rectified sine wave.

$$f(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(2\pi n f_{out} t) + 0$$

$$b_n = 0. a_n = \int_0^{\frac{1}{2f_{out}}} \sin(2\pi f_{out} t) \cos(4\pi f_{out} n t) dt = \frac{\cos^2(\pi n)}{\pi f_{out} - 4\pi f_{out} n^2}$$

$$a_0 = \int_0^{\frac{1}{2f_{out}}} 2f_{out} \sin(2\pi f_{out} t) dt = \frac{2}{\pi}$$

The Fourier Transform of $f(t)$ is

$$F(f) = \frac{2}{\pi} \delta(f) + \sum_{n=1}^{\infty} \frac{\cos^2(\pi n)}{2(\pi f_{out} - 4\pi f_{out} n^2)} \{ \delta(f - n f_{out}) + \delta(f + n f_{out}) \}$$

The oscilloscope output FFT is the sampled version of the frequency spectrum of $f(t)$, at the rate of $f_{s_{DAC}}$. Using MATLAB to approximate the frequency component of the sampled rectified sine wave, there are obvious peaks at multiples of 400Hz (Figure 9). In this case, the rectified spectrum is at 400Hz as the low pass filter passband consists of the components of the sampled spectrum, centred at multiples of $f_{s_{DAC}}$.

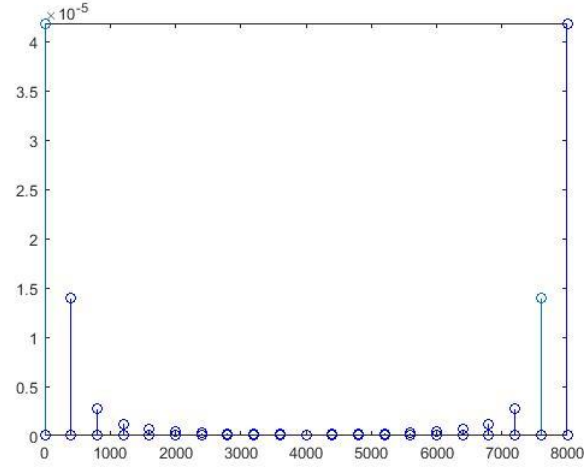


Fig. 9. MATLAB plot of sampled rectified sine wave

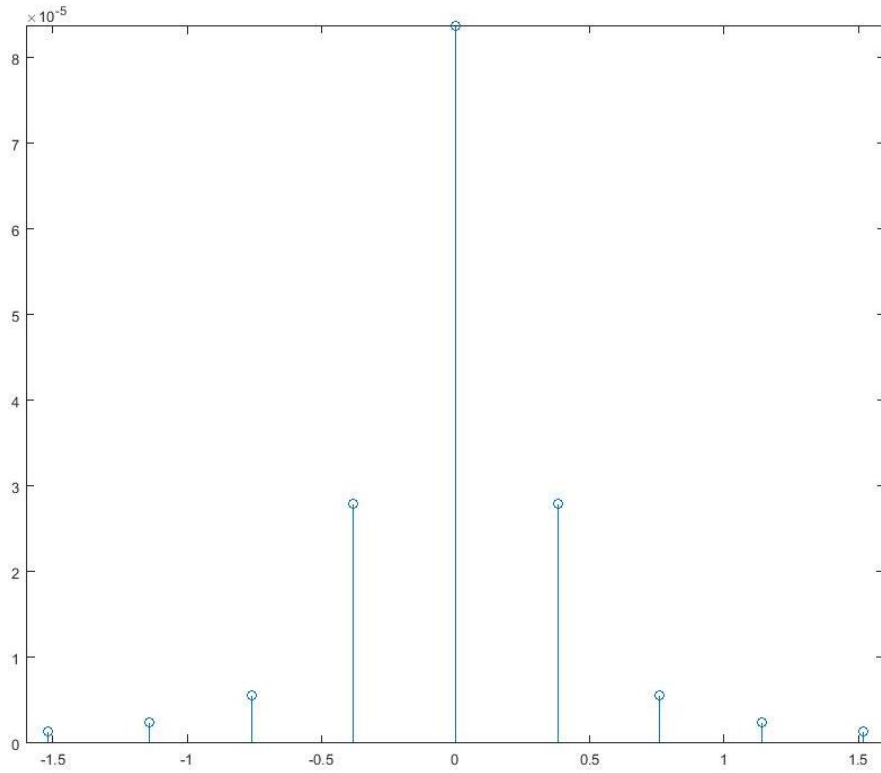


Fig. 10. MATLAB plot of un-sampled frequency spectrum of rectified wave

According to the aforementioned wrap-around effect, the components greater than $\frac{f_{s_{ADC}}}{2}$ will be ‘wrapped around’ to $|f_{s_{ADC}} - 2f_{in} \bmod f_{s_{ADC}}|$. In Figure 10, this effect appeared at the spectrum when the component had a frequency larger than $\frac{f_{s_{ADC}}}{2}$ - aliasing.

III. OPERATION OF CODE

A. Pre-processor statements

Libraries containing helper functions such as `math.h` for trigonometry are defined at the top of the code with `#include <file>`. Other global constants are also defined in this section, e.g. `PI`, with `#define`

`<text> <value>`. The pre-processor will look at these statements and replace the statements with their contents into the source file to produce an expanded source code file, before handing it over to the compiler for compilation.

B. Global variables/declarations

Instantiations of structs defined in library files can be found below the pre-processor statements – their aim is to define a set of configurations to be called in later functions, whether this is to initialize hardware or specify new data structures. For example, the codec config is defined here, and is called in `init_hardware()`, as the configuration used by the codec (`H_Codec = DSK6713_AIC23_openCodec(0, &Config);`) Other global variables are also defined in the given skeleton code, with examples being sampling frequency, output frequency, and the array to hold the coefficients of the sine wave.

C. Initialisation functions

`void init_hardware()` configures the hardware using defined global variables. The struct of the config for configuring the handle can be found in the included header files; the instantiation in the global declarations section. This function is duly executed as the first line of main before any testbench code is executed.

`void init_HWI()` configures hardware interrupts; it disables all interrupts, assigns events to interrupts, then enables all interrupts again.

D. Main testbench

As this code functions on interrupts, there is no testbench code to execute apart from the initialization of hardware. After initialization, the program enters a `while(1)` infinite loop, doing nothing, waiting for interrupts.

When a hardware interrupt is encountered, the program runs a pre-defined function based on the function defined in `dsp_bios_.tcf`. In our case, it is `void MyFunk(void)`.

E. Sinegen functions

Sine wave coefficients are generated by `sin()` in `math.h`. The previous improvements of improving the resolution are disregarded for this exercise. A local variable with the keyword `static` (keep its value between invocations) is used to keep track of the array index instead of a global variable as the value is only used in `sinegen()`. This array index is incremented by an amount determined by the desired frequency. Logic is included to handle overflow of this index (when it is larger than array size).

The function to generate the coefficients can be replaced by a file (use `#include <file>`) that includes an array of pre-computed sine wave coefficients. This method saves a bit of compute time at the start, but loses out on flexibility as the array size is fixed.

F. Interrupt function

The function that runs when a specific hardware interrupt is encountered is defined in the `dsp_bios_.tcf` file (Figure 11).

`void MyFunk()` contains only one line of code for both exercises, as there is no need for intermediate variables. It uses `mono_write_16Bit()` to write to both channels. `abs()` is used to rectify the sine wave. In the second exercise, the sinegen result is first cast to 16 bits as required by `mono_write_16Bit()`, then multiplied by a large gain; else, there is no discernable output on the oscilloscope.

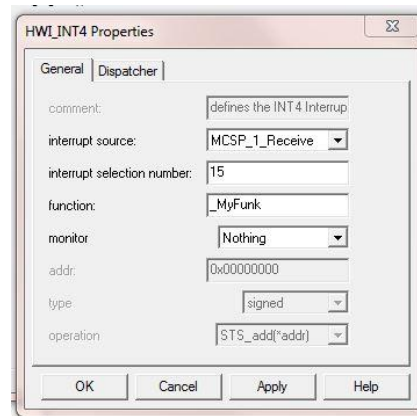


Fig. 11. Hardware Interrupt Config

G. Exercise 1: Interrupt Service Routine

```
void MyFunk(void){
    mono_write_16Bit(-abs(mono_read_16Bit()));
}
```

This exercise used the defined interrupt function to read an input stereo sample, compute the average, and output a mono sample to both channels. The function which did these tasks was provided, and is called `mono_read_16Bit()`, and `mono_write_16Bit()`. Quite simply, this exercise can be done by writing the absolute value of the read in sample.

The waveform on the oscilloscope is inverted when connecting the L/R channel to the oscilloscope. This can be solved by using `-abs()` instead of `abs()`, or by using the 'Phones' output, which does not invert the waveform.

H. Exercise 2: Interrupt Driven Sine Wave

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

float sinegen2(void){
    static float index=0;
    float result=table[(int)index];
    index+=(float)(freq*SINE_TABLE_SIZE/sampling_freq);
    if(index>SINE_TABLE_SIZE-1){
        index-=SINE_TABLE_SIZE;
    }
    return result;
}

void MyFunk(void){
    mono_write_16Bit((Int16)(-abs(sinegen2()*gain)));
}
```

The second exercise used the same `mono_write_16Bit()` function as the first exercise to write to both channels, but instead of writing a sample that was read in with `mono_read_16Bit()`, it writes a pre-computed sample from `sinegen2()` – a single precision floating point coefficient of a sine wave. These coefficients are generated at runtime with `sine_init()`, and are saved into an array before entering the aforementioned `while(1)` loop. The interrupt function looks up these coefficients by array index, and increments the index accordingly (`index+=(float)(freq*SINE_TABLE_SIZE/sampling_freq);`).

The testbench code was also changed such that the interrupt function is called when the hardware is in 'transmit', rather than 'receive' mode. This was done by modifying `MSCP_1_RECIEVE` to `MSCP_1_TRANSMIT` in `dsp_bios_.tcf`, as well as in `init_HWI()`, where `IRQ_EVT_RINT1` was changed to `IRQ_EVT_XINT1`.

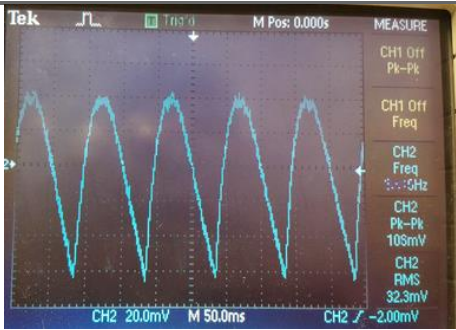
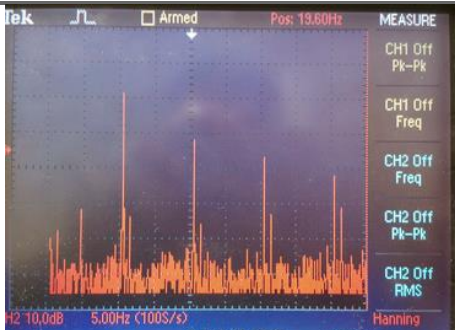
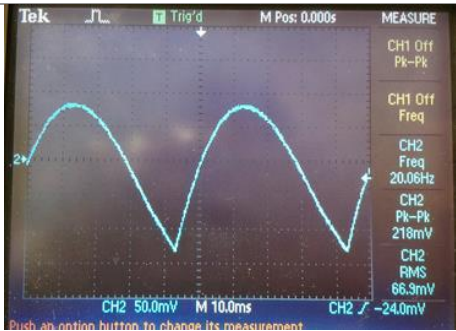

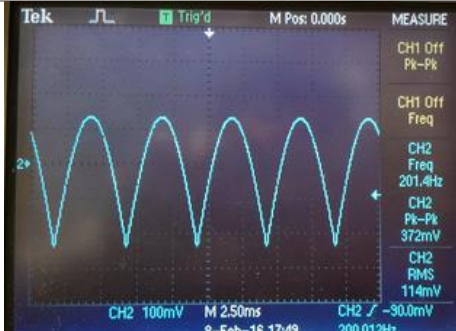
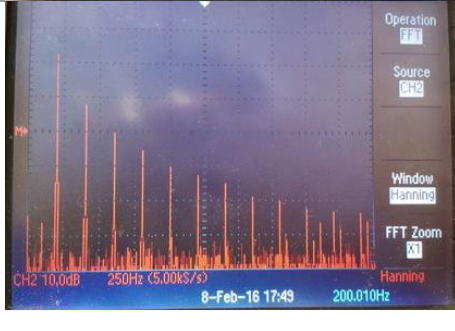
The resultant sine wave coefficient was multiplied by a large gain to make a visible sine wave on the oscilloscope. Since the output is set to 16 bits, the maximum gain is $2^{15} - 1$ (MSB is used for sign in 2's complement representation).

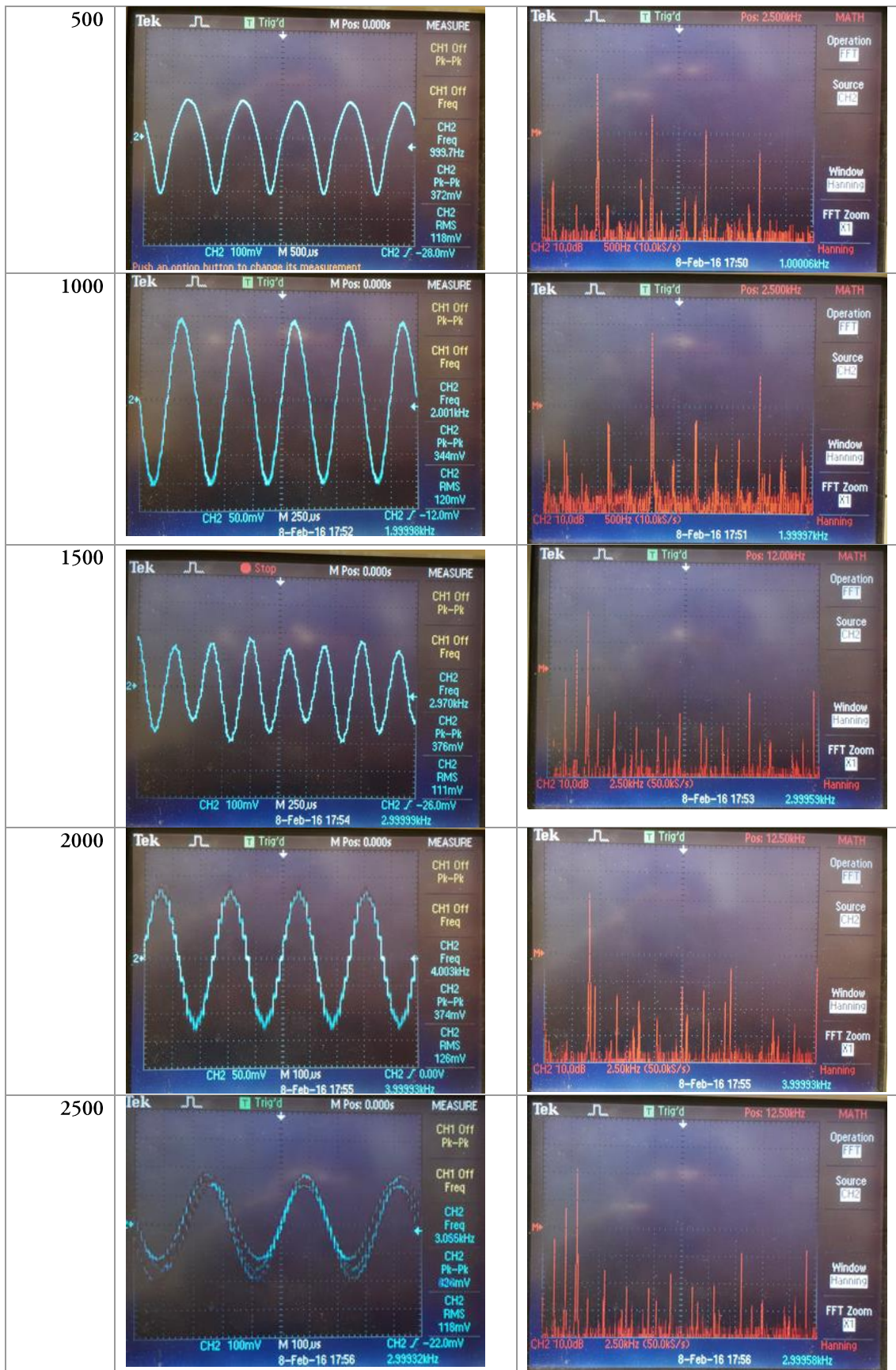
Again, the waveform on the oscilloscope is inverted when connecting the L/R channel to the oscilloscope. This can be solved by using `-abs()` instead of `abs()`, or by using the 'Phones' output, which does not invert the waveform.

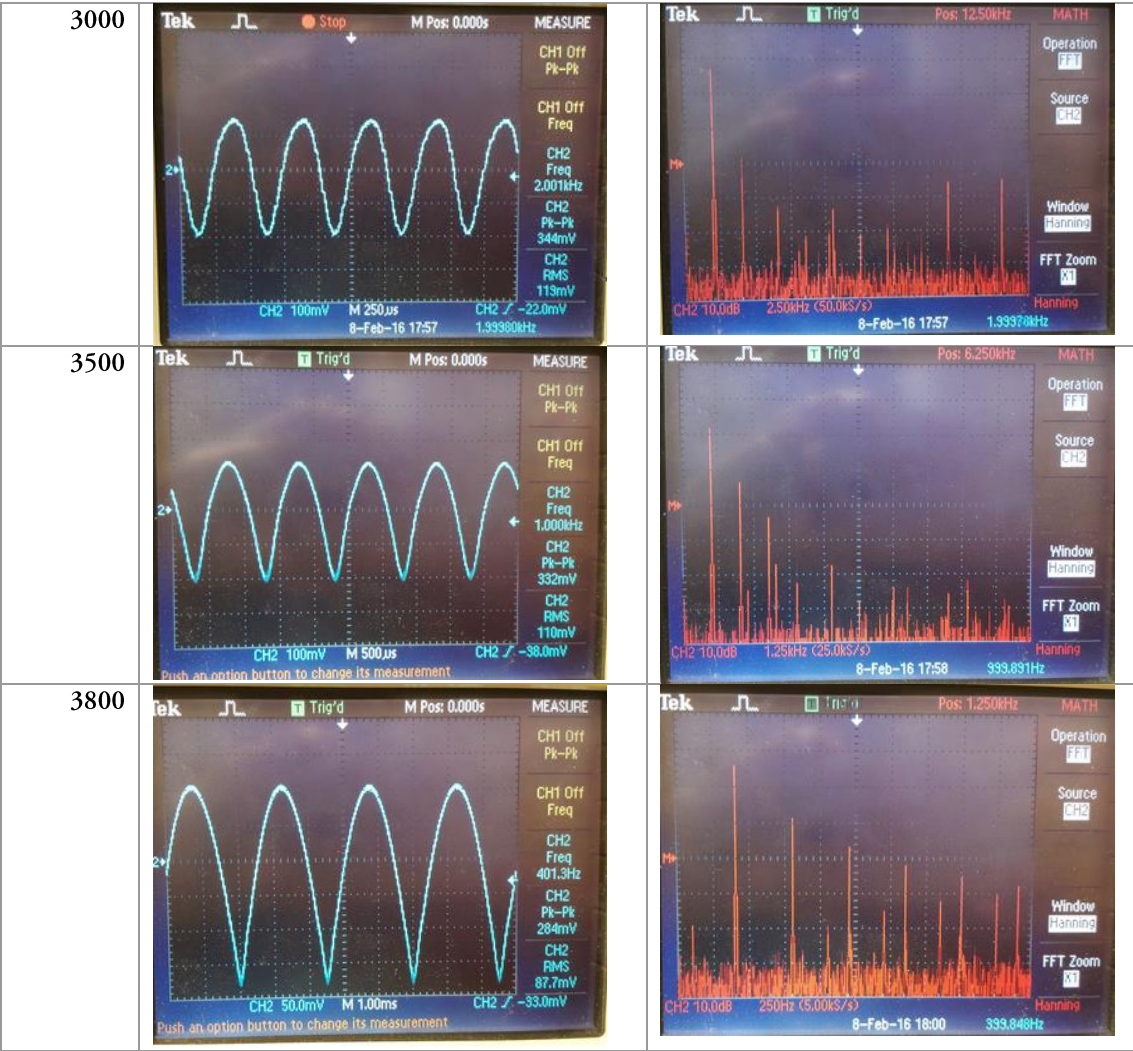
IV. OSCILLOSCOPE TRACES

This section shows the oscilloscope output for a range of frequencies, and the corresponding FFT graph. A discussion on these results will be provided in the next section.

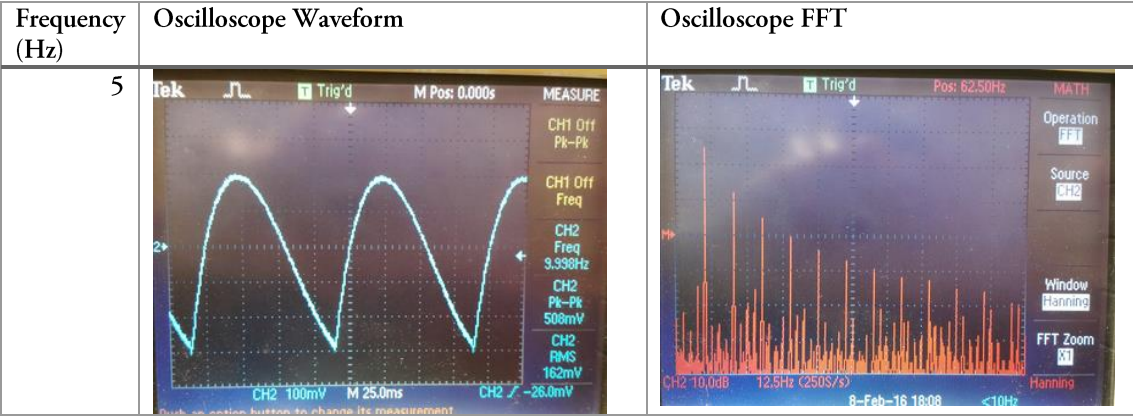
A. Exercise 1

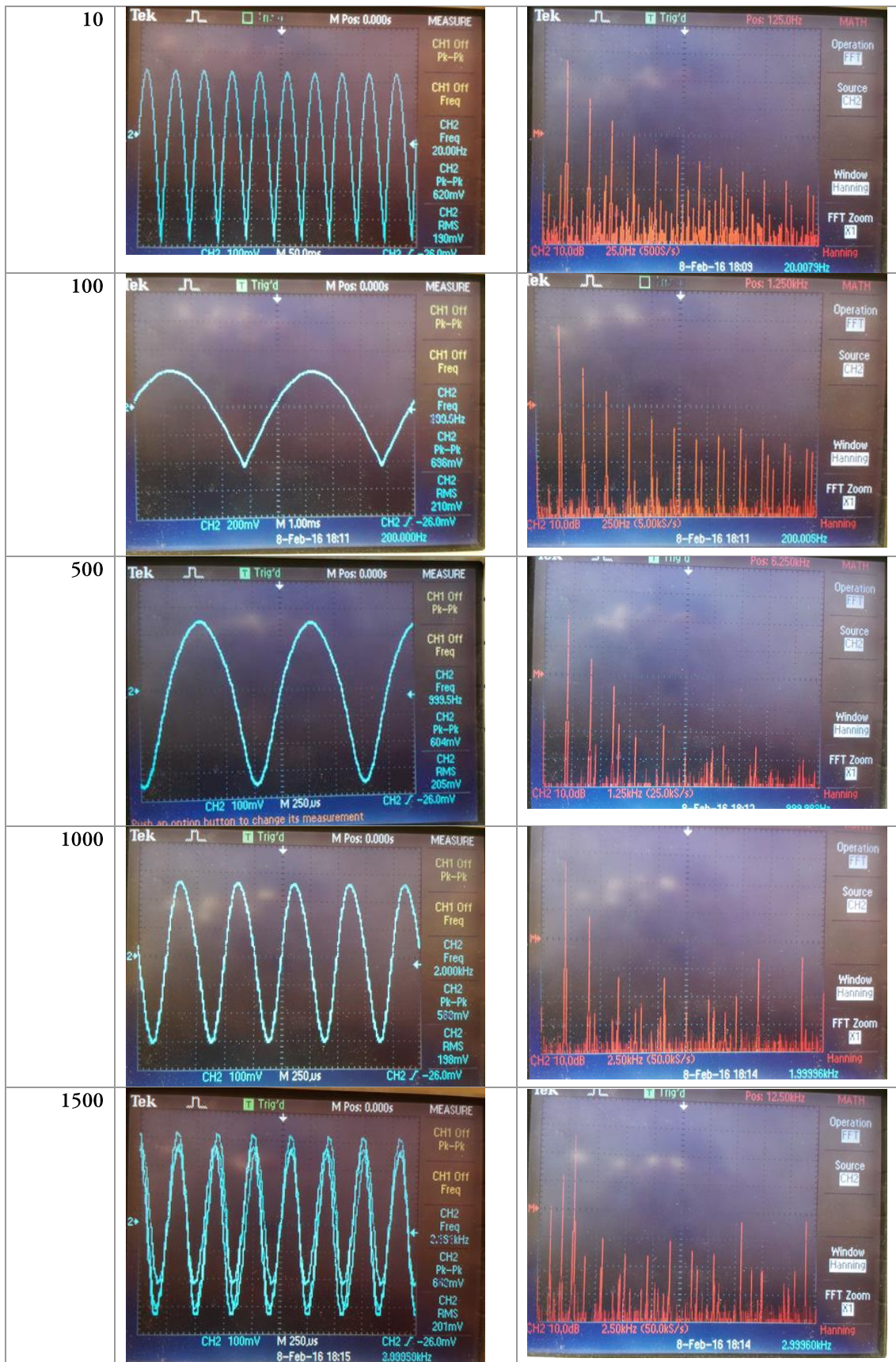
Frequency (Hz)	Oscilloscope Waveform	Oscilloscope FFT
5		
10		
100		

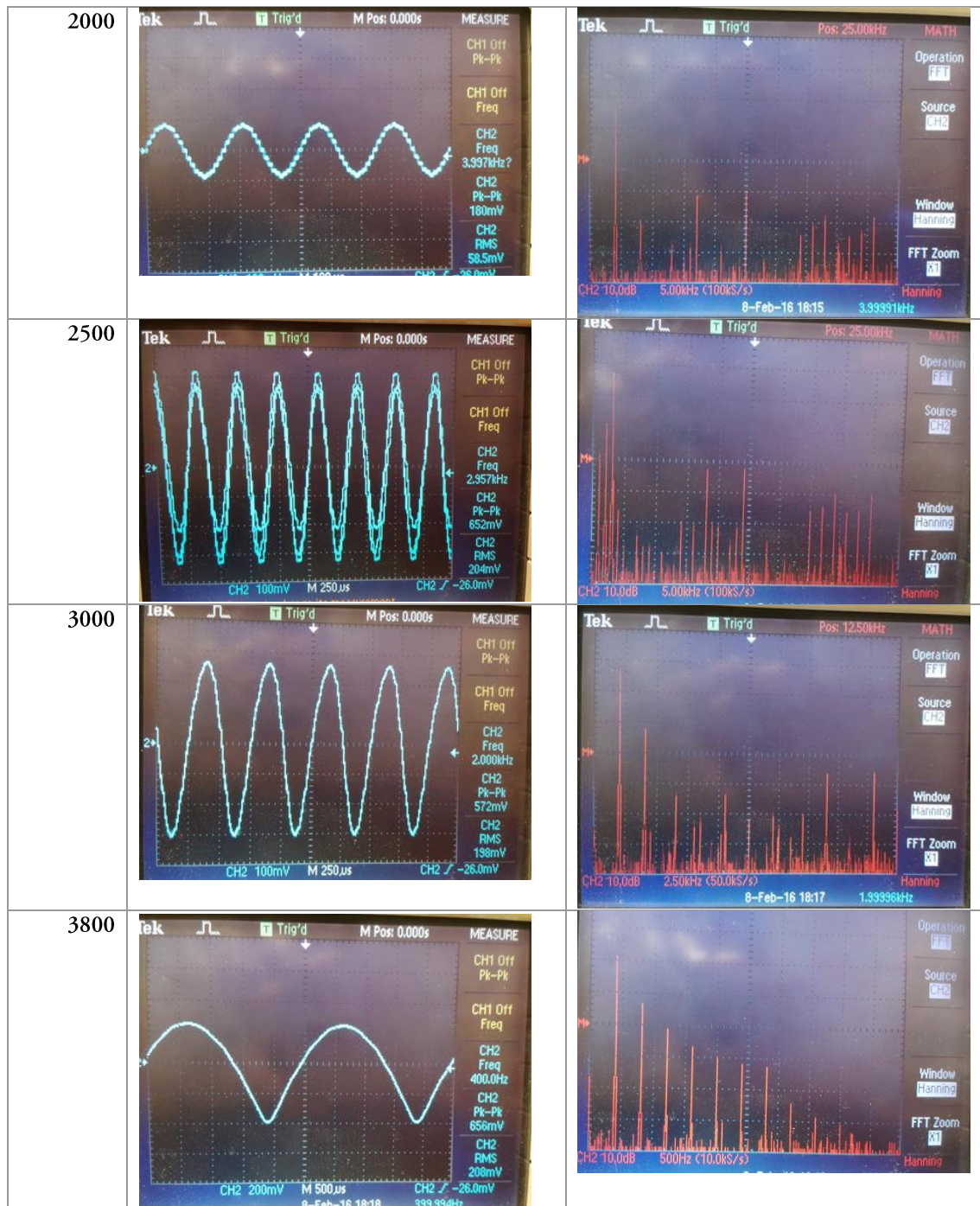




B. Exercise 2







V. DISCUSSION ON OSCILLOSCOPE GRAPHS

In addition to prior discussion, the oscilloscope waveforms from both exercise 1 and 2 can be separated into three categories:

1. A fully rectified sine wave
2. A rectified sinewave with amplitude distortion (due to the superposition with the sampled spectrum)
3. A normal sine wave due to an insufficient amount of harmonics in the passband region in the frequency domain (needed to maintain the rectified output in the time domain).

The following table shows the distribution of frequencies into these three categories.

Frequency (Hz)	Category(-ies)
5	1
10	1
100	1
500	1
1000	3
1500	2, 3
2000	3
2500	2, 3
3000	1, 2
3500	1, 2
3800	1, 2

5Hz to 500Hz function as expected. Note that the waveform for exercise 1's 5Hz is noisier than exercise 2's. This is because exercise 1 outputs a sampled sample to the DAC, while exercise 2 outputs a generated sample. The block diagram (Appendix.A) shows the sample in exercise 1 passing through two high pass filters; the sample in exercise 2 passes through only one. Hence, exercise 1's 5Hz output is noisier due to a lower SNR ratio. Around 500Hz is when the number of harmonics ($\left\lfloor \frac{f_{sDAC}}{2f_{in}} \right\rfloor$) in the passband region decrease to <4 . This causes the output waveform's shape to be a normal sine wave, instead of the expected rectified version.

Around 2500Hz is when there is amplitude distortion. From the FFT graphs, this happens when the components do not overlap when superposition occurs. This happens when $2f_{in}$ is not a factor of f_{sADC} .

Above 3000Hz, the aforementioned wrap-around effect ($f_{out} = |f_{sADC} - 2f_{in} \bmod f_{sADC}|$ if $f_{in} > \frac{f_{sADC}}{2}$) is visible; for example, the frequency of 3800Hz has a realized frequency of 400Hz instead, and this is what the oscilloscope measures.

VI. CONCLUSION

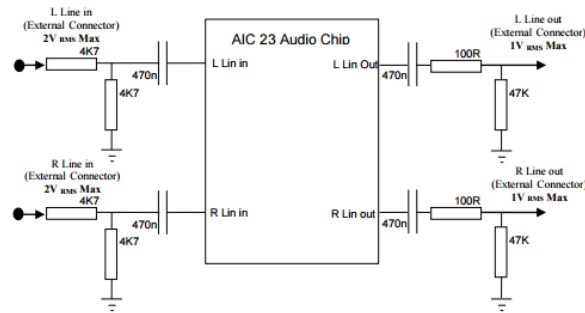
This task was completed successfully and a fully rectified sine wave was successfully generated using hardware interrupts and a look up table. When this rectification fails, explanations were provided. Formulas to guarantee a fully rectified sine wave is generated are given as part of the question answers.

VII. REFERENCES

- [1] P. D. Mitcheson, "Lab 3 – Interrupt I/O," 2 1 2013. [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-591057-dt-content-rid-2714543_1/courses/DSS-EE3_19-15_16/lab3.pdf. [Accessed 16 2 2016].

VIII. APPENDIX

A. High Pass Filter on input/output of AIC23 [1]



B. 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;
// Pre-processor definitions
#define PI 3.14159265
#define SINE_TABLE_SIZE 256
/***** Function prototypes *****/
void init_hardware(void);
void init_HWI(void);
/* The following MyFunk function is defined in the .tcf file to be the function that
runs
* when the hardware interrupt is called.
*
* The function does not have a return value or type. It calls supporting functions
sinegen2.
*/
void MyFunk(void);
float sinegen2(void);
void sine_init(void);

float freq = 1000.0;
float sampling_freq=8000;
float table[SINE_TABLE_SIZE]={0};
float gain = 16384;

/***** Main routine *****/
/**
* Main testbench section. Only responsible for simple initialisations and then
* does nothing as program functions off interrupts.
*/
void main(){

    // initialize board and the audio port
    init_hardware();
    sine_init();
    /* initialize hardware interrupts */
    init_HWI();

    /* loop indefinitely, waiting for interrupts */
    while(1)
    {};

}

/***** hardware() *****/
/**
* This function initialises the hardware. The codec configuration is defined in
* the global declarations part. Otherwise, all commands are as specified by TI.
*/
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(PCR1, 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(PCR1, XINTM, FRM);

}

/***** HWI() *****/
/**
 * Function to initialise interrupts. Change IRQ_EVT_RINT1 to IRQ_EVT_XINT1 when
 * transitioning from a receive to transmit interrupt. Make sure functions are
 * tied to their respective interrupt in the configure tcf file.
 */
{
    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*****/
//Exercise 1 - read a sample, and output it to both channels as a mono sample //
//void MyFunk(void){
    mono_write_16Bit(-abs(mono_read_16Bit()));
}
*/
/**
 * The following function fills an array with coefficients of a sine wave.
 * Coefficients for a whole cycle (2PI) are generated. Resolution can be
 * increased by only generating one quadrant. However, this requires extra logic
 * in the function to look-up the coefficients.
 *
 * For simplicity, this improvement is ignored here.
 */
void sine_init(void){
    int i=0;
    for (i=0;i<SINE_TABLE_SIZE;i++){
        table[i]=sin(2*PI*i/SINE_TABLE_SIZE);
    }
}

/**
 * This function generates a single precision floating point representation of a
 * coefficient of a sine wave. This function uses a static variable instead of

```

```

* using global variables as the array index variable is not used anywhere else.
* On every function call, the array index is incremented by a step size
* corresponding to the required frequency. The last if statement handles
* overflow of the array index.
* @return float
*/
float sinegen2(void){
    static float index=0;
    float result=table[(int)index];
    index+=(float)(freq*SINE_TABLE_SIZE/sampling_freq);
    if(index>SINE_TABLE_SIZE-1){
        index-=SINE_TABLE_SIZE;
    }
    return result;
}

/**
* This is the function that is called when the hardware interrupt is executed.
* For compactness, there are no variables defined as everything can be
* expressed in one line. The sinegen2 return value is cast as 16 bits as
* required by mono_write_16Bit. The return value from sinegen2 is also
* multiplied by a pre-defined gain - otherwise there is no output on the
* oscilloscope.
*
* Using the L/R output channels of the DSKC6713 board inverts the output.
* Either use the headphone output, or change abs() to -abs().
*/
void MyFunk(void){
    mono_write_16Bit((Int16)(-abs(sinegen2()*gain)));
}

```