Imperial College London

Real-Time Digital Signal Processing

Laboratory 3

Ahmad Moniri, CID: 00842685

Pranav Malhotra, CID: 00823617

Declaration: We confirm that this submission is our own work. In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the College examination offences policy

Signed: Ahmad Moniri, Pranav Malhotra

April 29, 2016

Contents

1 Introduction

In laboratory 2, the polling technique of communicating with peripherals was explored. When polling a peripheral, the processor cannot execute any other instruction and this presents a stark limitation on the usefulness of polled I/O software. In laboratory 3, interrupt-driven programming is introduced. Interrupt-driven programs are more efficient; such programs have the capacity to fully utilise the computation power of the processor/embedded device.

This report starts off by introducing the hardware that enables interrupt-driven programming. Following this, an in-depth discussion of the laboratory experiment is presented. This discussion includes a mathematical description of a fully rectified sine wave and aliasing and its effects. All theory is corroborated through experimental results that are presented in the form of scope traces.

2 Interrupt Service Routines

A processor can communicate with multiple peripherals simultaneously. At any time, any one of these peripherals may need to send/receive data from the processor; when this happens, the peripheral raises a flag to signal that it needs servicing. De-conflicting interrupt requests (IRQs) is done on a dedicated piece of hardware called the interrupt handler. Each peripheral is allocated an interrupt priority; interrupts with higher priority are the first to be serviced. The interrupt handler contains a vector of addresses corresponding to the locations in memory where the code that services the specific interrupt is stored.

The interrupt handler will send this memory address to the processor's Program Counter (PC). An essential part of servicing interrupts is to ensure that the registers that the CPU is currently using are not corrupted. Thus, before the processor branches to a specific memory location, to service the interrupt, it performs a context save. This saves the current state of the registers that the Interrupt Service Routine (ISR) makes use of, on the stack. Once the ISR is completed, the state of the registers is restored. In many architectures, interrupts are classed into fast interrupts and slow interrupts. Fast interrupts contain additional hardware in the form of shadow registers that enable servicing of interrupts without the need for context saves. This can have a significant impact on the efficiency of the program if the interrupt needs to be serviced often. Mouse or keyboard actions are examples of events that may be assigned as fast interrupts.

The TMS320C6713 contains multiple on-chip peripherals. Texas Instrument provides Chip Support Libraries (CSLs) presenting the programmer an additional layer of abstraction that allows simple and effective communication with the on-chip peripherals. The AIC23 audio codec is not a simple memory-mapped peripheral and thus the Multi-Channel Buffered Serial Port (McBSP), which is an on-chip peripheral, is used to communicate with the it. The communication makes use of two serial ports; one uni-directional port is used to program the configuration registers in the audio codec, and one bi-directional port is used for data flows.

It is of great importance to note that it is the MsBSPs that raise the IRQ and not the audio codec. CSL modules are used to deal with interrupts generated by the MsBSPs.

3 Exercise 1

3.1 Hardware Initialisation

Before programs are run on the DSK board, initialising the hardware to the required settings is necessary. The function `init_hardware` performs this operation. As noted in section 2, the MsBSP needs to be connected to the AIC23 audio codec. This is achieved in line 11 of the listing 1. Following this, line 15 serves to define the event that raises the IRQ: the acquisition of a 32 bit word from the AIC23 audio codec. It is also clear that 2 serial ports need to be initialised; one for configuration and one for data flows.

```
1  void init_hardware(){
2      // Initialise the board support library, must be called first
3      DSK6713_init();
4
5      // Start the AIC23 codec using the settings defined above in config
6      H_Codec = DSK6713_AIC23_openCodec(0, &Config);
7
8      /* Function below sets the number of bits in word used by MSBSP (serial port) for
9      receives from AIC23 (audio port). We are using a 32 bit packet containing two
10     16 bit numbers hence 32BIT is set for  receive */
11     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
12
13     /* Configures interrupt to activate on each consecutive available 32 bits
14     from Audio port hence an interrupt is generated for each L & R sample pair */
15     MCBSP_FSETS(SPCR1, RINTM, FRM);
16
17     /* These commands do the same thing as above but applied to data transfers to
18     the audio port */
19     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
20     MCBSP_FSETS(SPCR1, XINTM, FRM);
21 }
```

Listing 1: Hardware Initialisation Settings

3.2 Interrupt Initialisation

Next, the hardware interrupts need to be enabled. The function `init_HWI` performs this operation. It is important to note that although the IRQ has been mapped to a physical interrupt in the DSP/BIOS configuration dialogue box, this process has to be repeated in the form of line 4 in listing 2.

```
1  void init_HWI(void){
2      IRQ_globalDisable();            // Globally disables interrupts
3      IRQ_nmiEnable();                // Enables the NMI interrupt (used by the debugger)
4      IRQ_map(IRQ_EVT_RINT1,4);       // Maps an event to a physical interrupt
5      IRQ_enable(IRQ_EVT_RINT1);      // Enables the event
6      IRQ_globalEnable();             // Globally enables interrupts
7  }
```

Listing 2: Interrupt initialisation for exercise 1

3.3 Full Wave Rectification Function

Listing 3 shows the ISR that is run every time the MsBSP raises an IRQ. The function reads a 16-bit[1] word from the MsBSP, takes it absolute value and writes it into the buffer of the MsBSP. The reading and writing is performed using the provided functions `mono_read_16Bit` and `mono_write_16Bit`.

```
1  void ISR_AIC(void){
2      // read sample, write absolute value
3      mono_write_16Bit(abs(mono_read_16Bit()));
4  }
```

Listing 3: ISR_AIC rectification function

[1]The AIC23 audio codec sends a 32-bit word however the function `mono_read_16Bit()` transform the 32-bit word into a 16-bit word. The function assumes that both the left and the right sample are the same and thus takes their average.

3.4 Limits on Input Signal

The input signal has the following restrictions:

1. The root-mean-square voltage of the signal cannot exceed $2V$[2].
2. The frequency of the signal is limited to $10Hz$ and $3.8kHz$.

The first restraint comes from the fact that Analogue-to-Digital Converter (ADC) in the AIC23 audio codec has a full-scale range of $1.0V_{rms}$. To avoid distortions, it is important to not exceed the full-scale range[1]. This, together with the fact that there is a potential divider at the line input with a gain of 0.5 sets the first restraint.

The lower limit on the input signal can be attributed to the high-pass filter at the line input. Signals below $10Hz$ will be significantly attenuated and will not utilise the full range of the ADC and thus result in reduced resolution.

The upper limit on the input signal is attributed to the fact that we wish to prevent aliasing when the signal is sampled. Since the AIC23 audio codec allows multiple sampling frequencies, it would be very costly to implement multiple anti-aliasing analogue filters at the input. There is only one anti-aliasing analogue filter with a cut-off frequency of $48kHz$ to support the maximum sampling frequency of $96kHz$. Thus, aliasing is prevented by manually ensuring that a signal with frequency greater than half that of the sampling frequency[3] is not used as an input.

[2] $V_{rms} = 2$ equates to an amplitude of $2.83V$, which corresponds to a peak-to-peak voltage of $5.66V$.

[3] An anti-aliasing filter with a cut-off frequency of $4kHz$ can be implemented by oversampling the signal, filtering, then decimating the signal to obtain a signal with samples every $8kHz$. This is however not likely as inputting a signal above $4kHz$ still produces an output. Thus, the best way to prevent aliasing in this experiment is to do so manually.

3.5 Testing of Function

For the purpose of testing, all input waves have $V_{rms} = 1.90V$. This consistency is maintained to make any relative attenuation clear. An example of an input wave with $f = 100Hz$ is shown in Figure 1.

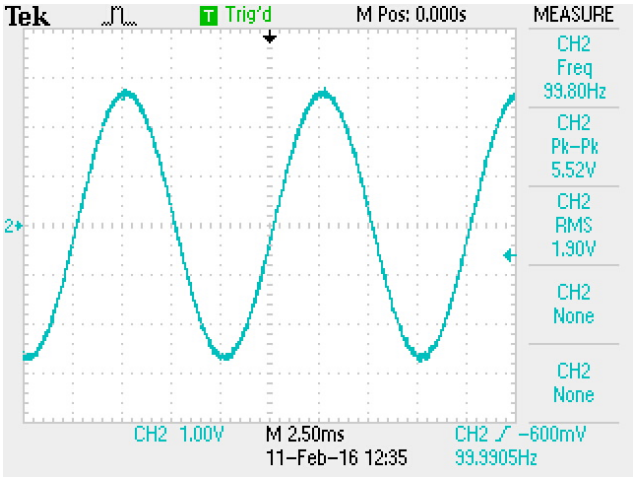

Figure 1: Reference input sine wave

The function `ISR_AIC` was tested with the input wave displayed in Figure 1 and Figure 2 shows that it performs as expected. There are however 2 interesting observations that are worth discussing.

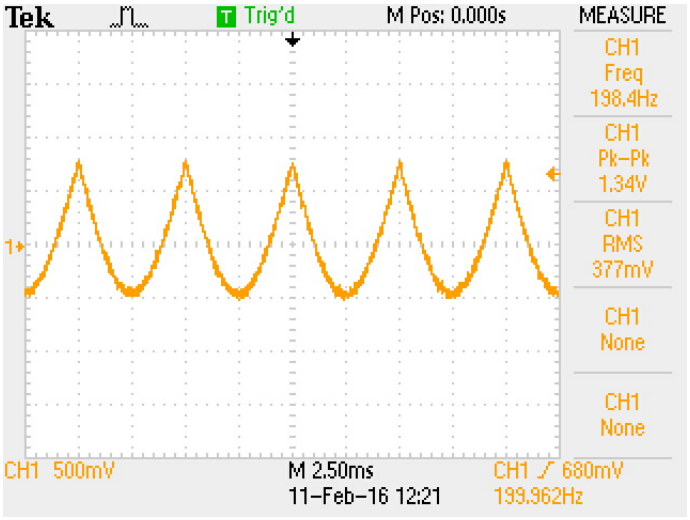

Figure 2: Fully rectified sine wave with $100Hz$ input

1. *The output is an inverted version of an ideal fully-rectified sine wave.*

This observation is attributed to the inversion of the signal at some point in the AIC23 audio codec[4]. From this point, all graphs in the report are inverted on the oscilloscope to account for this quirk.

2. *The output is a centred around* $0V$.

An ideal fully-rectified sine wave should have a DC component and thus should appear above the x-axis instead of being centred around it. Figure 3 shows that there is a high-pass filter at the input and the output of the AIC23 audio chip. The high-pass filter blocks the DC component and thus centres the rectified signal around the x-axis.

[4]The reason for the inversion is not conclusive and thus this is conjecture.

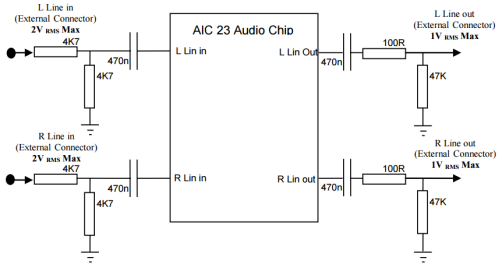

Figure 3: AIC23 audio chip external connections[1]

The high-pass filter has the following transfer function, where $R_1 = 100\Omega$, $R_2 = 47k\Omega$ and $C = 470nF$:

$$H(s) = \frac{V_o(s)}{V_i(s)} = \frac{R_2 C s}{1 + (R_1 + R_2)Cs} \tag{1}$$

The cut-off frequency of the filter is:

$$f_p = \frac{1}{2\pi(R_1 + R_2)C} = \frac{1}{2\pi(100 + 47 \cdot 10^3)(470 \cdot 10^{-9})} = 7.1895Hz \tag{2}$$

Lastly, the filter's frequency response is graphed in Figure 4.

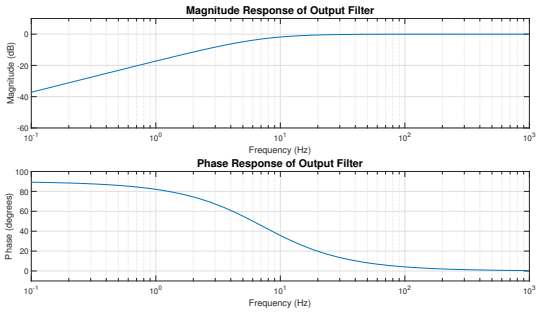

Figure 4: Frequency response of high-pass filter at the output of AIC23 audio codec

The analogue filter at the output is designed to block the DC component however, it cannot be implemented perfectly. As calculated above, the cut-off frequency is $7.18Hz$. When a $10Hz$ sine wave with $V_{rms} = 1.90V$ is used as an input, the output is not only attenuated but is also skewed. When the frequency of the input wave is reduced to $5Hz$, both effects become more pronounced. The skewing is due to the fact that the analogue filter has a non-linear phase response at these frequencies. Besides skewing, the amplitude of the output wave also decreases with a decrease in frequency. The decay in amplitude is due to the high-pass nature of the filter. These observations are illustrated in Figure 5.

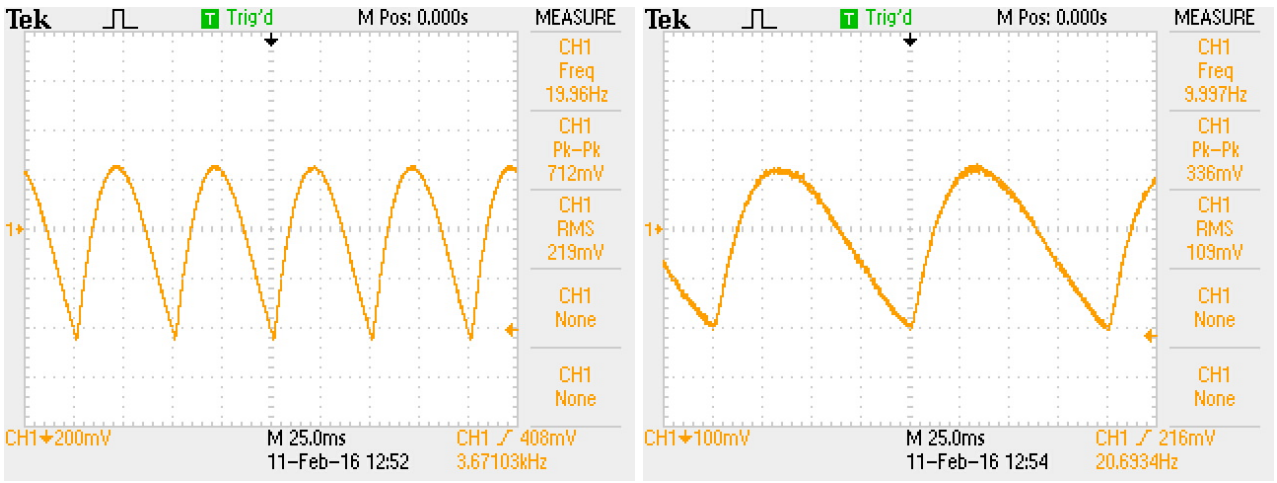

Figure 5: Skewing of output waveform at low frequencies

3.6 Fourier Series of a Rectified Sine Wave

One of the most important learning lessons from laboratory 3 is that simple operations in the time domain have very intricate implications in the frequency domain. Taking the absolute value of a sample seems simple enough in the time domain, however it causes harmonics of the fundamental frequency to appear in the frequency spectrum. This is explained mathematically by considering the Fourier Series of a sinusoid.
The Fourier Series of a signal $f(x)$ is represented as:

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n cos(nx) + b_n sin(nx) \tag{3}$$

The process of calculating the coefficients of the Fourier Series is called Fourier Analysis. The equations below are used for Fourier Analysis:

$$a_0 = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)dx \tag{4}$$

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)cos(nx)dx \tag{5}$$

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)sin(nx)dx \tag{6}$$

To find the DC components of the rectified sine wave, equation (4) is used:

$$a_0 = -\frac{1}{\pi}\int_{-\pi}^{0} sin(x)dx + \frac{1}{\pi}\int_{0}^{\pi} sin(x)dx \tag{7}$$

$$= \frac{2}{\pi}\int_{0}^{\pi} sin(x)dx \tag{8}$$

$$= \frac{2}{\pi}\left[-cos(x)\right]_0^{\pi} \tag{9}$$

$$= \frac{4}{\pi} \tag{10}$$

Next, to find the general form of the a_n components, equation (5) is used:

$$a_n = -\frac{1}{\pi}\int_{-\pi}^{0} sin(x)cos(nx)dx + \frac{1}{\pi}\int_{0}^{\pi} sin(x)cos(nx)dx \tag{11}$$

$$= \frac{2}{\pi}\int_{0}^{\pi} sin(x)cos(nx)dx \tag{12}$$

$$= \frac{1}{\pi}\int_{0}^{\pi} sin(x+nx) + sin(x-nx)dx \tag{13}$$

$$= \frac{1}{\pi}\left[-\frac{cos(x+nx)}{1+n} - \frac{cos(x-nx)}{1-n}\right]_0^{\pi} \tag{14}$$

$$= \frac{1}{\pi}\left[\frac{cos(x+nx)}{1+n} + \frac{cos(x-nx)}{1-n}\right]_{\pi}^{0} \tag{15}$$

$$= \frac{1}{\pi}\left(\left(\frac{1}{1+n} + \frac{1}{1-n}\right) - \left(\frac{cos(\pi+n\pi)}{1+n} + \frac{cos(\pi-n\pi)}{1-n}\right)\right) \tag{16}$$

$$= \frac{1}{\pi}\left(\frac{2}{1-n^2} - \frac{cos(\pi)cos(n\pi) - sin(\pi)sin(n\pi)}{1+n} - \frac{cos(\pi)cos(n\pi) + sin(\pi)sin(n\pi)}{1-n}\right) \tag{17}$$

$$= \frac{1}{\pi}\left(\frac{2}{1-n^2} + \frac{cos(n\pi)}{1+n} + \frac{cos(n\pi)}{1-n}\right) \tag{18}$$

$$= \frac{1}{\pi}\left(\frac{2}{1-n^2} + \frac{2}{1-n^2}cos(n\pi)\right) \tag{19}$$

$$= -\frac{2}{\pi}\ \frac{1+cos(n\pi)}{n^2-1} \tag{20}$$

Lastly, to find the general form of the b_n components, equation (6) is used:

$$b_n = -\frac{1}{\pi}\int_{-\pi}^{0} sin(x)sin(nx)dx + \frac{1}{\pi}\int_{0}^{\pi} sin(x)sin(nx)dx \tag{21}$$

$$= \frac{2}{\pi}\int_{0}^{\pi} sin(x)sin(nx)dx \tag{22}$$

$$= \frac{1}{\pi}\int_{0}^{\pi} cos(x-nx) - cos(x+nx)dx \tag{23}$$

$$= \frac{1}{\pi}\left[\frac{sin(x-nx)}{1-n} - \frac{sin(x+nx)}{1+n}\right]_0^{\pi} \tag{24}$$

$$= \frac{1}{\pi}\left(\left(\frac{sin(\pi-n\pi)}{1-n} - \frac{sin(\pi+n\pi)}{1+n}\right) + (0)\right) \tag{25}$$

$$= \frac{1}{\pi}\left(\frac{sin(\pi)cos(n\pi) - cos(\pi)sin(n\pi)}{1-n} - \frac{sin(\pi)cos(n\pi) + cos(\pi)sin(n\pi)}{1+n}\right) \tag{26}$$

$$= \frac{1}{\pi}\left(\frac{sin(n\pi)}{1-n} - \frac{sin(n\pi)}{1+n}\right) \tag{27}$$

$$= \frac{1}{\pi}\ \frac{2n}{1-n^2}\ sin(n\pi) \tag{28}$$

From the Fourier Analysis applied above, it is clear that the rectified sine wave has the following equation:

$$f(x) = |sin(x)| = \frac{2}{\pi} - \frac{4}{\pi}\sum_{n=2,4,6,\cdots}^{\infty}\frac{cos(nx)}{n^2-1} \tag{29}$$

As mentioned above, the repercussions of a simple time-domain operation of taking the absolute value has the profound effect of producing an infinitely long Fourier Series. **It should be noted that a perfect fully-rectified sine wave will only be produced if the output is a sum of infinite harmonics, weighted according to equation (29).** There are 2 reasons why this is not possible.

1. Producing infinite harmonics is not physically possible.
2. An anti-aliasing filter[5] is placed at the output such that frequencies above $4kHz$ are attenuated.

To compare the quality of the outputs, a $200Hz$ fully-rectified sine wave is graphed next to a $1kHz$ fully rectified sine wave in Figure 6.

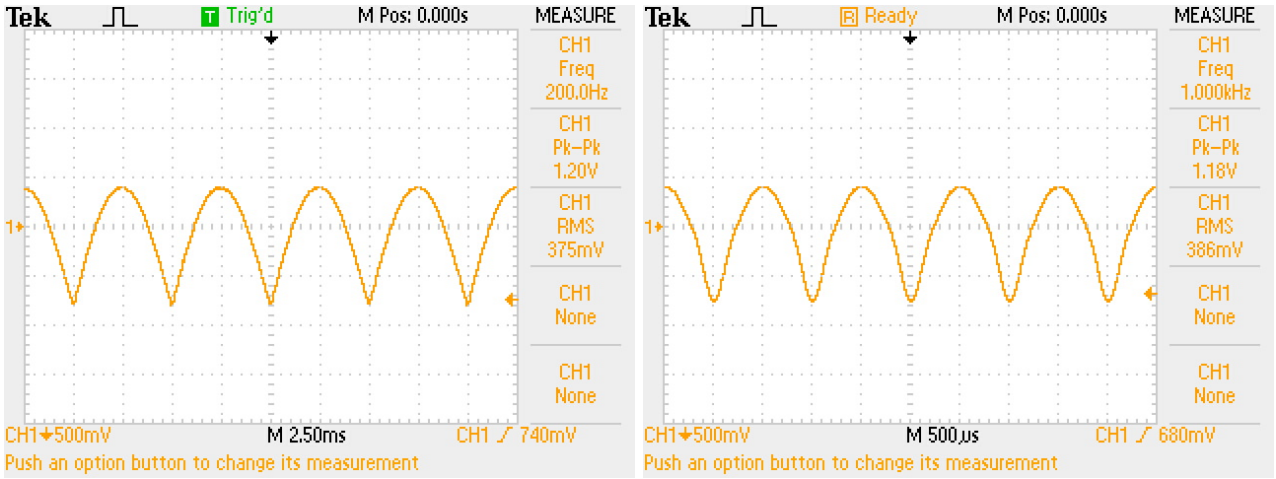

Figure 6: Comparison of quality of outputs

[5]The anti-aliasing filter is a digital filter. The AIC23 audio codec allows for multiple sampling frequencies and thus it only makes sense to have 1 analogue anti-aliasing filter with a cut-off frequency of $48kHz$ to support the maximum sampling frequency of $96kHz$. Anti-aliasing filters for other sampling frequencies are implemented digitally

It is clear that the $200Hz$ output has much sharper edges than the $1kHz$ output. The discrepancy in the quality of the outputs is explained by considering the Fourier Series representation the outputs.

For the $200Hz$ output:

$$f_1(x) = |sin(2\pi(100)nt)| = \frac{2}{\pi} - \frac{4}{\pi}\left(\frac{cos(2\pi(200)t)}{3} + \frac{cos(2\pi(400)t)}{15} + \frac{cos(2\pi(600)t)}{35} + \cdots\right) \tag{30}$$

Whereas for the $1kHz$ output:

$$f_2(x) = |sin(2\pi(500)nt)| = \frac{2}{\pi} - \frac{4}{\pi}\left(\frac{cos(2\pi(1000)t)}{3} + \frac{cos(2\pi(2000)t)}{15} + \frac{cos(2\pi(3000)t)}{35} + \cdots\right) \tag{31}$$

The theoretical spectrum of the two output waves are plotted in Figure 7. These were generated using a MATLAB script which is presented in Appendix B. Note that the spectra are plotted up to $16kHz$ however the low-pass anti-aliasing filter will attenuate frequencies above $4kHz$.

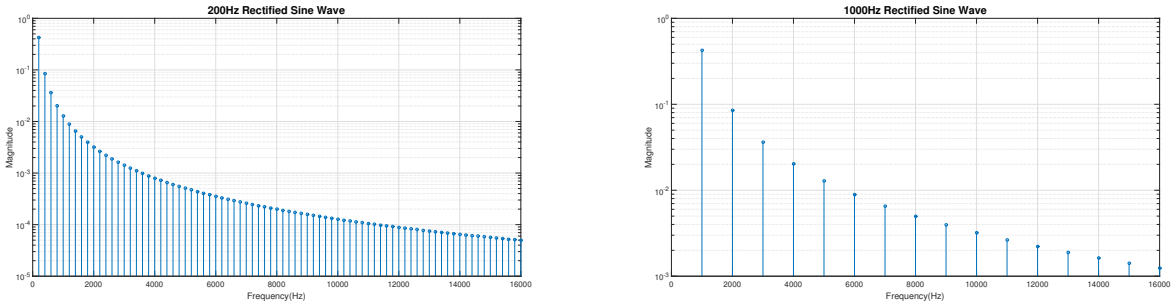

Figure 7: Theoretical spectrum

From Figure 7 it is clear that before the $4kHz$ cut-off frequency is reached, $f_1(x)$ will have 20 terms in its Fourier Series whereas $f_2(x)$ will only have 4 terms. The shorter Fourier Series accounts for imperfections in the output, most notably the rounded edges.

The findings are corroborated by the Fast Fourier Transform (FFT) plots in Figure 8.

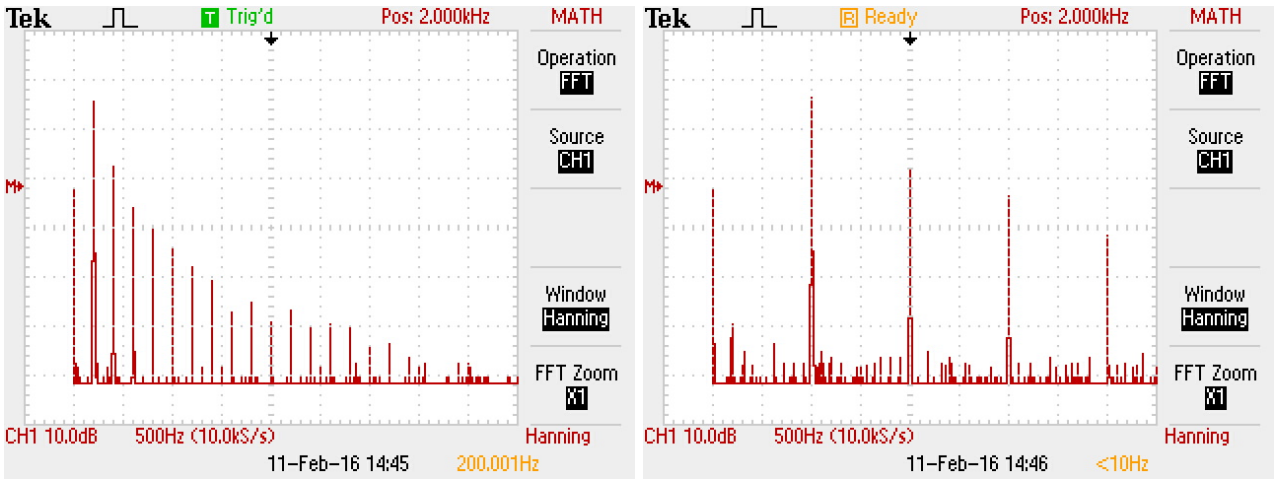

Figure 8: FFT of $200Hz$ and $1kHz$ output waves

The truncation of the Fourier Series also has a profound effect at high frequencies. The Fourier Series of a $3kHz$ output has the following form:

$$f_3(x) = |sin(2\pi(1500)nt)| = \frac{2}{\pi} - \frac{4}{\pi}\left(\frac{cos(2\pi(3000)t)}{3} + \frac{cos(2\pi(6000)t)}{15} + \frac{cos(2\pi(9000)t)}{35} + \cdots\right) \tag{32}$$

Equation (32) maps to the frequency spectrum shown in Figure 9:

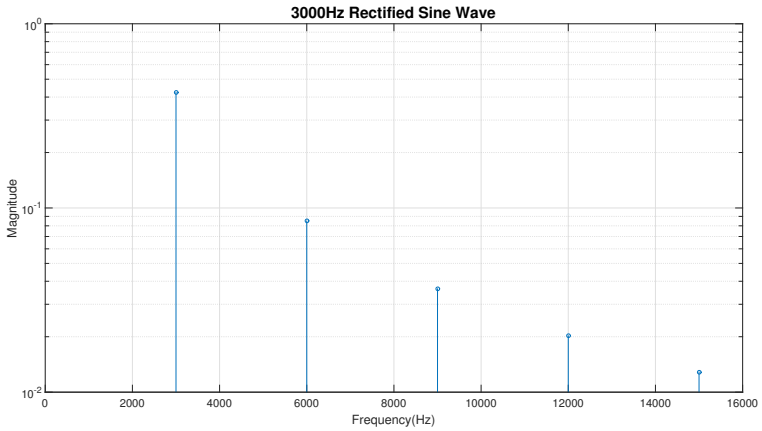

Figure 9: Frequency spectrum of $3kHz$ rectified wave

A serve shortening of the Fourier Series is expected when a $1.5kHz$ sine wave is used as an input. There is only one spectral component below $4kHz$ and thus the output should be a $3kHz$ sine wave that is not rectified. Figure 10 shows the findings. A severely distorted output was observed and thus the FFT of the output is also provided in Figure 10.

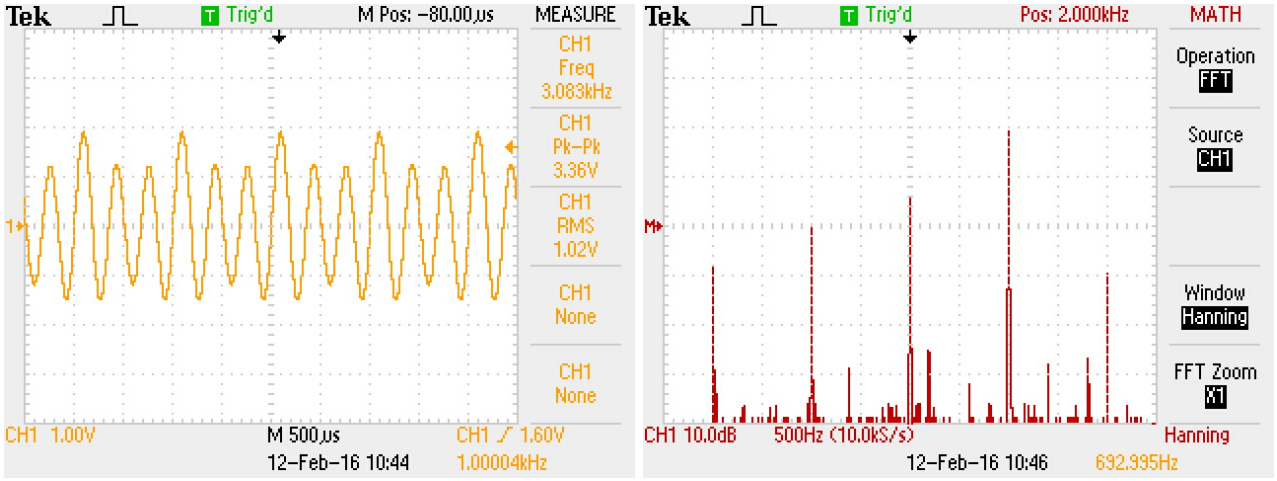

Figure 10: $1.5kHz$ input produces an output that is not rectified

A very strong spectral component at $3kHz$ is observed. This is expected. However, spectral components at other frequencies also exist. Their existence is explained in the next section.

3.7 Frequency Domain Analysis

As alluded to in the previous section, two interesting phenomena were observed when a $1.5kHz$ sine wave is used as an input; firstly the signal's amplitude is modulated and secondly, the signal's FFT had multiple harmonics that equation (29) does not account for. The phenomena is more pronounced when a $1.9kHz$ sine wave is used as an input.

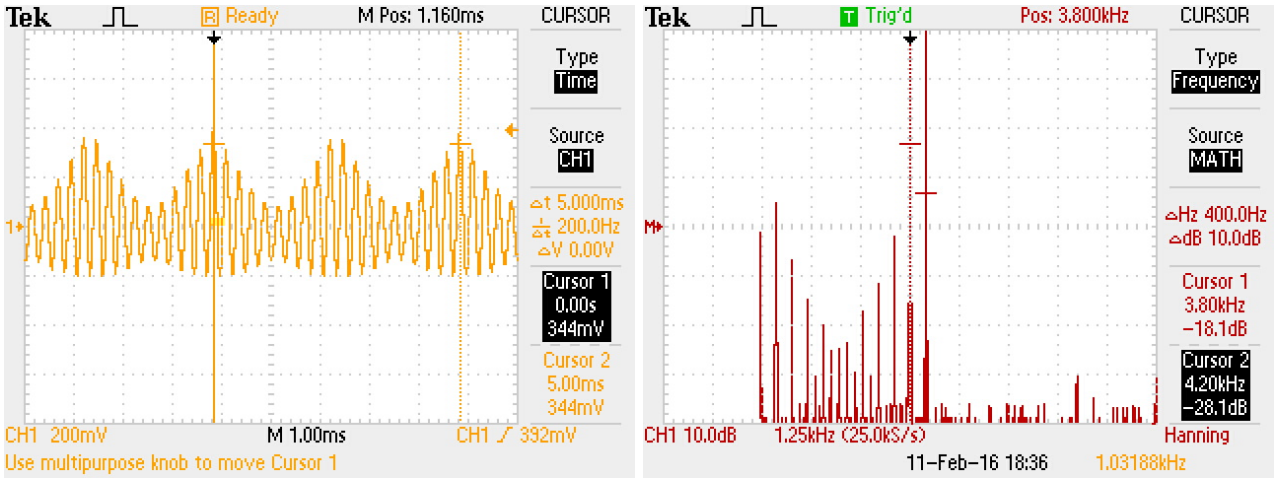

Figure 11: Aliasing and amplitude modulation with $1.9kHz$ sine wave used as an input

Explanation of this requires mathematical analysis in the frequency domain instead of in the time domain. There are three important concepts that are needed to understand this interesting phenomena.

1. **Taking the absolute value of a sampled sine wave is equivalent to sampling a fully-rectified wave of double the frequency and doing no processing. In memory, both values result in the same floating point number. Thus, for the following analysis, it is assumed that a rectified sine wave is provided as an input.**

2. **Equation (29) shows that the frequency spectrum of a fully-rectified sine wave is not band-limited[6]. This means that no matter what sampling frequency is used, aliasing will still be present. Aliasing, however was not observed up until this point because the high frequency spectral components decay and the aliased harmonics were insignificant**

3. **The anti-aliasing filter is not an ideal brick wall filter[7] and thus, spectral components above $4kHz$ may appear in the output.**

The spectrum of a $3.8kHz$ fully-rectified sine wave is shown in Figure 12. As mentioned above this, it is assumed that a $3.8kHz$ fully-rectified sine wave is used as the input.

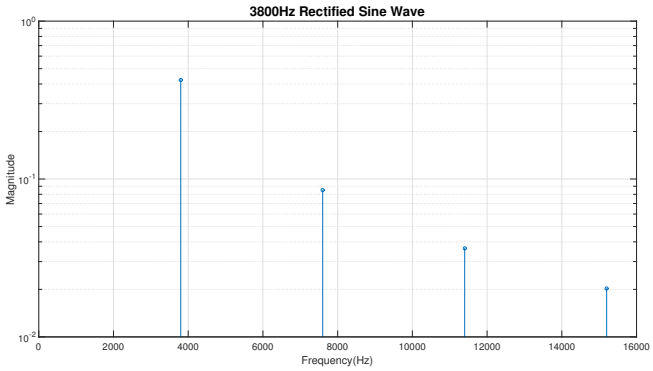

Figure 12: Frequency spectrum of a $3.8kHz$ rectified sine wave

[6] Albeit the equation shows the Fourier Series and not the spectral components, the coefficients of the Fourier Series are directly mapped into the frequency domain, and thus this loose use of language is acceptable.

[7] An ideal filter will require implementing a filter with an infinite number of coefficients.

The spectrum of a fully-rectified sine wave that has been sampled at $8kHz$ is shown is Figure 13.

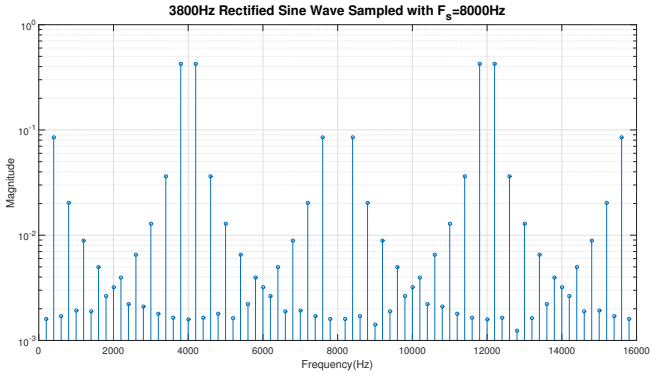

Figure 13: Frequency spectrum of a $3.8kHz$ rectified sine wave sampled at $8kHz$

Thus, it is clear that the sampling process has caused aliasing to occur. In this case, the aliasing is observable because spectral components above $4kHz$ are significant. The aliasing explains the multiple harmonics that are observed in the FFT of the output. It alone cannot explain the amplitude modulation. **The amplitude modulation occurs because the anti-aliasing filter is not perfect. The anti-aliasing filter does not fully attenuate a harmonic at $4.2kHz$. This together with the fundamental harmonic at $3.8kHz$ cause amplitude modulation that is observed.** Equation (34) shows what happens when two sine waves of different frequencies are added together.

$$acos(\theta) + bcos(\phi) = (a-b)cos(\theta) + bcos(\theta) + bcos(\phi) \tag{33}$$

$$= (a-b)cos(\theta) + bcos(\frac{\theta+\phi}{2})cos(\frac{\theta-\phi}{2}) \tag{34}$$

Applying the above trigonometric identity, equation (36) explains the $200Hz$ envelope that is observed. **It should be noted that although a $200Hz$ envelope is observed in the time domain, this component will not appear in frequency domain. The time domain view only shows the effect that is manifested as a result of equation (36).**

$$sin(2\pi(3800)t) + sin(2\pi(4200)t) = 2sin(2\pi\frac{(3800+4200)}{2}t)cos(2\pi\frac{(3800-4200)}{2}t) \tag{35}$$

$$= 2sin(2\pi(4000)t)cos(2\pi(200)t) \tag{36}$$

3.8 Effective Folding Frequency

Lastly, because of a fully-rectified sine wave has twice the frequency as that of a sine wave, the effective folding frequency due to sampling at $8kHz$ is at $2kHz$ and not at $4kHz$. Thus, using a $3.8kHz$ sine wave as an input should result in the same output as using a $200Hz$ sine wave. This is confirmed in Figure 14.

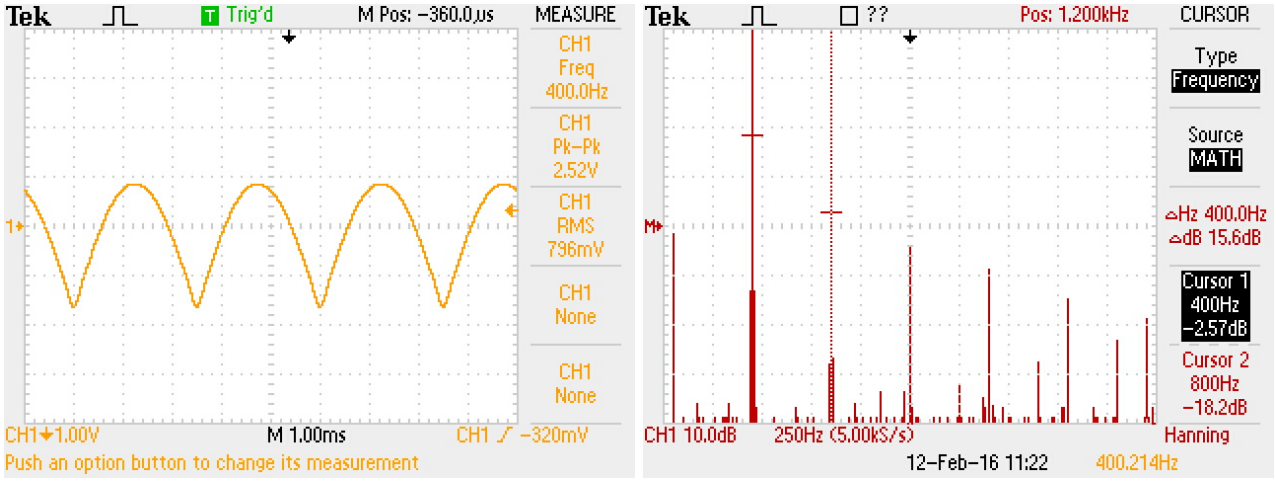

Figure 14: Output when $3.8kHz$ sine wave is used as input

3.9 Further Discussion

It is important to note that the phenomena observed above can be viewed from another perspective. To explain how multiple unexpected harmonics appear in the frequency spectrum of the fully-rectified sine wave, the convolution between the spectrum of a sine wave and a square wave can be considered. **The process of taking an absolute value was viewed above as simply sampling a fully-rectified wave. It can however also be viewed as the multiplication of a sine wave and a square wave in the time-domain. The mathematical analysis requires computing the convolution of two spectra and can be performed easily on MATLAB.** It provides the same results but the discussion above is more intuitive.

It can be concluded that to obtain a proper fully-rectified sine wave without,

- Significant rounding of the edges
- Aliasing
- Amplitude modulation
- Amplitude attenuation
- Skewing

The frequencies have to be limited to $10Hz$ to $400Hz$.

4 Exercise 2

In exercise 2, the sine wave is generated using a look-up table instead of taking as input a sine wave generated by software or hardware. An ISR is run when the McBSP is ready to write a sample rather than when it has received one.

4.1 Interrupt Initialisation

The configuration file was modified to account for this change[8].

```
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
}
```

Listing 4: Interrupt initialisation for exercise 2

4.2 Full Wave Rectification Function

Listing 5 shows the two functions necessary to implement a full wave rectification using a look up table. Firstly, the function `sine_init` initialises a look-up table that contains 256 samples from 1 cycle of a sine wave. A more involved discussion of this generation process was presented in the previous report and is thus excluded from this report. Next, the function `ISR_AIC` defines the ISR. Note that the increment value is re-calculated each time the ISR is called to enable the use of the watch window to change the signal frequency. The number of samples that are skipped is stored in the variable `increment` which can be monitored in the watch window for debugging purposes[9].

```
void sine_init(void){
    // initilisies look-up table
    // RESOLUTION provides the option to only store one quadrant of the sine wave

    int i;
    // Store an entire sine wave (or single quadrant) in look-up table
    for(i=0; i<SINE_TABLE_SIZE; i++)
        table[i] =  sin(2*PI*i/(RESOLUTION*SINE_TABLE_SIZE));
}

void ISR_AIC(void){
    // interrupt service routine to fully rectify sine wave
    // uses global variables, increment, index, table
    // code allows use of watch window to monitor variables
    // RES_OPTIMISE allows use of optimised code

    // calculate increment here so that we can exploit watch window
    increment = sine_freq*SINE_TABLE_SIZE*RESOLUTION/sampling_freq;

    index+=increment;

    // modulo index so that we do not exceed size of look-up table
    index = fmod(index,SINE_TABLE_SIZE*RESOLUTION);

    // Read sample then write absolute value
    // Note: to create a visible scope output, we magnify by 30000
    #ifdef RES_OPTIMISE
        mono_write_16Bit(abs( 30000*getSineValue(round(index)) ));
    #else
        mono_write_16Bit(abs( 30000*table[(int)round(index)] ));
    #endif
}
```

Listing 5: Code to generate interrupt driven rectified sine wave (using a look-up table)

[8]The interrupt source was changed from `MCSP_1_Receive` to `MCSP_1_Transmit`. The function `init_HWI` was also modified to map the ISR to the correct physical event `IRQ_EVT_XINT1`

[9]Again, a more involved discussion is provided in the previous report.

Since the ISR should be as fast as possible, an option to disable resolution optimisation, using an `#ifdef` directive defined in the pre-processor statements, is included. If resolution optimisation is enabled, only one quadrant of the sine wave is stored and the function `getSineValue` is called to manipulate the index so as to output the correct value from the look-up table.

```
/******************************** getSineValue() ********************************/
// This function is used if resolution optimisation is enabled
// Since only one quadrant of the sine wave is stored
// this function returns the correct entry in the look-up table
// index1 used to prevent confusion with global variable index
// index1 is within function's local scope
#ifdef RES_OPTIMISE
  float getSineValue(int index1){
      int quadrant = index1/SINE_TABLE_SIZE; // Quadrant number
      int modulo = index1 % SINE_TABLE_SIZE; // Distance from start of quadrant

    if (quadrant == 0)
      return(table[index1]);
    else if (quadrant == 1)
      return(table[SINE_TABLE_SIZE-modulo-1]);
    else if (quadrant == 2)
      return(-1*table[modulo]);
    else if (quadrant == 3)
      return(-1*table[SINE_TABLE_SIZE-modulo-1]);
    else
      return(0); // return null in case of error
  }
#endif
```

Listing 6: Optimised code

The look-up table consists of values between -1 and 1. Before a sample is written to the McBSP, amplification of the signal is necessary to obtain a visible trace on the oscilloscope. In laboratory 2, a gain of 2100000000[10] was used. The function `mono_write_16Bit` is used to write to the McBSP and thus a gain of 30000[11] is used. After amplification, the function `abs` is used to rectify the sine wave.

4.3 Experimental Findings

Using the look-up table to implement the rectification process results in identical output as sampling an input wave. Figure 15 is included to verify that our code functions well.

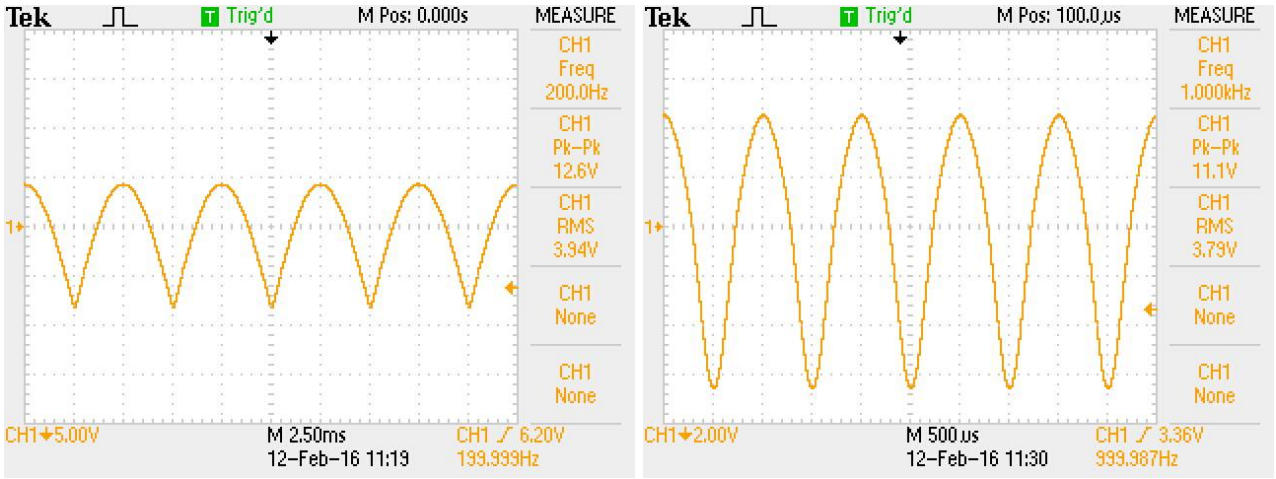

Figure 15: Traces that validate code for exercise 2

All other observations are identical to those obtained in exercise 1 and are excluded for the sake of brevity.

[10] The value sent to the McBSP should be below 2^{31} to prevent overflow/truncation errors, since the function `DSK6713_AIC23_write` accepts 32-bit integers

[11] The value sent to the McBSP should be below 2^{15} to prevent overflow/truncation errors, since the function `mono_write_16Bit` accepts 16-bit integers.

5 Conclusion

This report has presented the theory behind ISRs and the advantages that interrupt-driven programs have over polled I/O software. It has also provided a detailed mathematical analysis of the many interesting phenomena that were observed due to the sampling and rectification processes. All observations were supported by explanations from the mathematical and intuitive perspective. These form the ground work for material to be covered in the rest of the Real-Time Digital Signal Processing Course.

References

[1] Instruments, T. (2004). TLV320AIC23B, Stereo Audio CODEC, Data Manual. Retrieved February 04, 2016, from http://www.ti.com/lit/ds/symlink/tlv320aic23b.pdf

A C Code

Full code description for exercise 2 is provided below.

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

// Size of lookup table
#define SINE_TABLE_SIZE 256

// Activate Resolution optimization by only storing one quadrant of the sine wave
// (comment out to de-activate)
#define RES_OPTIMISE

#ifdef RES_OPTIMISE
  #define RESOLUTION 4
#else
  #define RESOLUTION 1
#endif

/******************************* 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;

// Holds current sample number
float index = 0;

// Holds the increment value (to acheive desired output frequency)
float increment = 0;

// Desired sinewave frequency (be wary of nyquist criterion)
float sine_freq = 1000.0;
int sampling_freq = 8000;

// Define look up table as global variable
float table[SINE_TABLE_SIZE];

/****************************** Function prototypes *******************************/
```

```
72 void init_hardware(void);
73 void init_HWI(void);
74 void ISR_AIC(void);
75 float sinegen(void);
76
77 #ifdef RES_OPTIMISE
78   float getSineValue(int index);
79 #endif
80
81 void sine_init();
82
83 /********************************* Main routine ***********************************/
84 void main(){
85     // initialize board and the audio port
86     init_hardware();
87
88     /* initialize hardware interrupts */
89     init_HWI();
90
91     // initialize lookup table
92     sine_init();
93
94     /* loop indefinitely, waiting for interrupts */
95     while(1)
96     {};
97 }
98
99 /********************************* init_hardware() ********************************/
100 void init_hardware()
101 {
102     // Initialize the board support library, must be called first
103     DSK6713_init();
104
105     // Start the AIC23 codec using the settings defined above in config
106     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
107
108     /* Function below sets the number of bits in word used by MSBSP (serial port) for
109     receives from AIC23 (audio port). We are using a 32 bit packet containing two
110     16 bit numbers hence 32BIT is set for  receive */
111     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
112
113     /* Configures interrupt to activate on each consecutive available 32 bits
114     from Audio port hence an interrupt is generated for each L & R sample pair */
115     MCBSP_FSETS(SPCR1, RINTM, FRM);
116
117     /* These commands do the same thing as above but applied to data transfers to
118     the audio port */
119     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
120     MCBSP_FSETS(SPCR1, XINTM, FRM);
121
122
123 }
124
125 /********************************* init_HWI() *************************************/
126 void init_HWI(void)
127 {
128     IRQ_globalDisable();            // Globally disables interrupts
129     IRQ_nmiEnable();                // Enables the NMI interrupt (used by the debugger)
130     IRQ_map(IRQ_EVT_XINT1,4);       // Maps an event to a physical interrupt
131     IRQ_enable(IRQ_EVT_XINT1);      // Enables the event
132     IRQ_globalEnable();             // Globally enables interrupts
133 }
134
135 /************************* INTERRUPT SERVICE ROUTINE ******************************/
136 void ISR_AIC(void)
137 {
138     // calculate increment here so that we can exploit watch window
139     increment = sine_freq*SINE_TABLE_SIZE*RESOLUTION/sampling_freq;
140
141     index+=increment;
142
143     // modulo index so that we do not exceed size of look-up table
144     index = fmod(index,SINE_TABLE_SIZE*RESOLUTION);
145
146     // Read sample then write absolute value
147     // Note: to create a visible scope output, we magnify by 30000
```

```
148     #ifdef RES_OPTIMISE
149         mono_write_16Bit(abs( 30000*getSineValue(round(index)) ));
150     #else
151         mono_write_16Bit(abs( 30000*table[(int)round(index)] ));
152     #endif
153 }
154 
155 /******************************** sine_init() ***********************************/
156 void sine_init(void){
157     int i;
158     // Store an entire sinewave (or single quadrant) in look-up table
159     for(i=0; i<SINE_TABLE_SIZE; i++)
160       table[i] =  sin(2*PI*i/(RESOLUTION*SINE_TABLE_SIZE));
161 }
162 
163 /******************************** getSineValue() ********************************/
164 // This function is used if resolution optimisation is enabled.
165 // Since only one quadrant of the sine wave is stored,
166 // this function returns the correct entry in the look-up table.
167 #ifdef RES_OPTIMISE
168     float getSineValue(int index1){
169         int quadrant = index1/SINE_TABLE_SIZE; // Quadrant number
170         int modulo = index1 % SINE_TABLE_SIZE; // Distance from start of quadrant
171 
172         if (quadrant == 0)
173             return(table[index1]);
174         else if (quadrant == 1)
175             return(table[SINE_TABLE_SIZE-modulo-1]);
176         else if (quadrant == 2)
177             return(-1*table[modulo]);
178         else if (quadrant == 3)
179             return(-1*table[SINE_TABLE_SIZE-modulo-1]);
180         else
181             return(0); // return null in case of error
182     }
183 #endif
```

Listing 7: C-code to generate interrupt driven rectified sine wave (using a look-up table)

B MATLAB Code

MATLAB code used to generate spectra is provided below.

```
lim = 1e5; % Defines maximum frequency of consideration
n = -lim:lim;
fsig = 3000; % Sine wave input frequency
fsamp = 8000; % Sampling frequency
frect = 2*fsig; % Frequency of rectified sine wave is twice that of original signal

rectified_sine = zeros(lim*2+2,1); % Initialising frequency spectrum

n = floor(lim/frect); % Number of terms in Fourier series of rectified sine wave

% Creating spectral peaks
for i = 1:n
rectified_sine(-i*frect + lim+1) = 4/(pi*((2*i)^2-1)); % Positive frequency term
rectified_sine(i*frect + lim+1) = 4/(pi*((2*i)^2-1));  % Negative frequency term
end

% Initialising frequency spectrum for impulse train (to mimic sampling)
train = zeros(lim*2+2,1);

train(lim+1) = 1; % Fundamental spectrum

n=floor(lim/fsamp); % Number of images
for i = 1:n
train(-fsamp*i+lim+1) = 1;
train(fsamp*i+lim+1) = 1;
end

n = -2*lim:2*lim+2;
y=conv(rectified_sine,train); % Convolution of two spectra
stem(n,y)
set(gca,'yscal','log') % Converting to log scale, removes 0's, makes plot clear
xlim([0 2*fsamp]) % Limiting range shown in plot

%Configuration settings
title('\fontsize{25}3000Hz Rectified Sine Wave Sampled with F_{s}=8000Hz')
xlabel('\fontsize{20}Frequency(Hz)')
ylabel('\fontsize{20}Magnitude')
a=get(gca,'XTickLabel');
set(gca,'XTickLabel', a, 'fontsize', 18);
grid on
```

Listing 8: Matlab code to generate frequency spectrum for a (sampled) rectified sinewave