Imperial College London

Real Time Digital Signal Processing

Lab 3 Report

Andrew Zhou, CID: 00938859

Jagannaath Shiva Letchumanan, CID: 00946740

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 In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others.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 offenses policy.
Signed: **Andrew Zhou, Jagannaath Shiva Letchumanan**

March 6, 2017

Contents

1 Answers to Questions

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

Normally, a full-wave rectified sine wave is strictly greater than zero, but the observed waveform is centred around 0V. This is due to the DC component of the signal being blocked by the high pass filters at the input and output of the AIC23 Audio chip (Figure 1). Additionally, the output is inverted implying that the circuit has an overall negative gain.

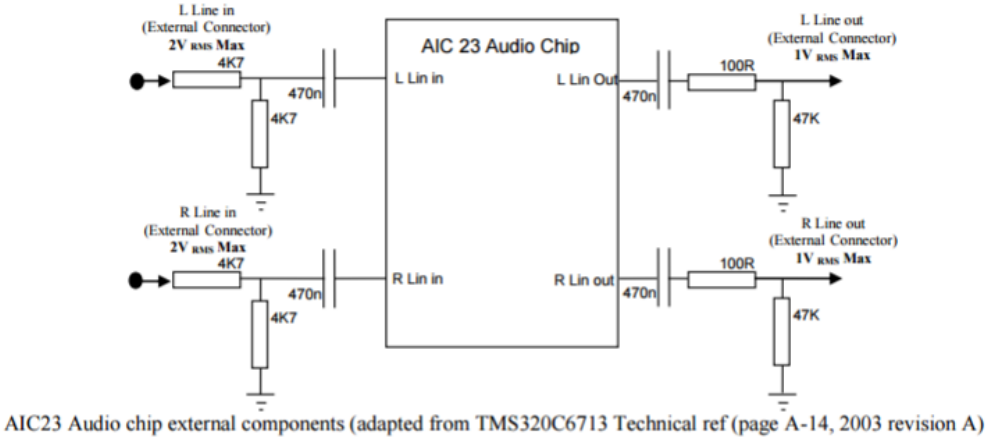

Figure 1: Filters at input and output of Audio Chip

The frequency components of the full rectified waveform can be found by Fourier analysis:

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

where the co-efficients a_0, a_n, and b_n are:

$$a_0 = \frac{1}{\pi}\int_0^{2\pi} f(x)dx$$

$$a_n = \frac{1}{\pi}\int_0^{2\pi} f(x)\cos(nx)dx$$

$$b_n = \frac{1}{\pi}\int_0^{2\pi} f(x)\sin(nx)dx$$

For a fully-rectified sine wave, the co-efficients are calculated as follows:

$$\begin{aligned}
a_0 &= \frac{1}{T}\int_0^{2\pi} |\sin(x)|dx \\
&= \frac{1}{T}\left[\int_0^{\pi} \sin(x)dx - \int_{\pi}^{2\pi} \sin(x)dx\right] \\
&= \frac{1}{\pi}\left[\big[-\cos(x)\big]_0^{\pi} - \big[-\cos(x)\big]_{\pi}^{2\pi}\right] \\
&= \frac{1}{\pi}\big[1+1+1+1\big] \\
&= \frac{4}{\pi}
\end{aligned}$$

$$\begin{aligned}
a_n &= \frac{1}{T}\int_0^{2\pi} |\sin(x)|\cos(nx)dx \\
&= \frac{1}{\pi}\left[\int_0^{\pi} \sin(x)\cos(nx)dx - \int_{\pi}^{2\pi} \sin(x)\cos(nx)dx\right] \\
&= \frac{1}{2\pi}\left[\int_0^{\pi} \sin((n+1)x) - sin((n-1)x)dx - \int_{\pi}^{2\pi} \sin((n+1)x) - sin((n-1)x)dx\right] \\
&= \frac{1}{2\pi}\left[\left[\frac{\cos((n+1)x)}{n+1}\right]_{\pi}^{0} + \left[\frac{\cos((n-1)x)}{n-1}\right]_{0}^{\pi}\right] - \frac{1}{\pi}\left[\left[\frac{\cos((n+1)x)}{n+1}\right]_{2\pi}^{\pi} + \left[\frac{\cos((n-1)x)}{n-1}\right]_{\pi}^{2\pi}\right] \\
&= \frac{1}{2\pi}\left[\frac{1}{n+1} - \frac{-1^{n+1}}{n+1} + \frac{-1^{n-1}}{n-1} - \frac{1}{n-1}\right] - \frac{1}{\pi}\left[\frac{-1^{n+1}}{n+1} - \frac{1}{n+1} + \frac{1}{n-1} - \frac{-1^{n-1}}{n-1}\right]
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2\pi}\left[\frac{2}{n+1} - \frac{2}{n-1} - \frac{2(-1)^{n+1}}{n+1} + \frac{2(-1)^{n-1}}{n-1}\right] \\
&= \frac{1}{\pi}\left[\frac{1}{n+1} - \frac{1}{n-1} - \frac{-1^{n+1}}{n+1} + \frac{-1^{n-1}}{n-1}\right] \\
&= \frac{1}{\pi}\left[\frac{-2}{n^2-1} + (-1)^{n-1}\left(\frac{1}{n-1} - \frac{1}{n+1}\right)\right] \\
&= \frac{1}{\pi}\left[\frac{-2}{n^2-1} + \frac{2}{n^2-1}(-1)^{n-1}\right] \\
&= \frac{2}{(n^2-1)\pi}\Big((-1)^{n-1} - 1\Big) \\
&= \frac{2}{(n^2-1)\pi}\Big(\cos((n-1)\pi) - 1\Big)
\end{aligned}
$$

$$
\begin{aligned}
b_n &= \frac{1}{T}\int_0^{2\pi} |\sin(x)| \sin(nx)dx \\
&= \frac{1}{\pi}\left[\int_0^{\pi} \sin(x)\sin(nx)dx - \int_{\pi}^{2\pi} \sin(x)\sin(nx)dx\right] \\
&= \frac{1}{2\pi}\left[\int_0^{\pi} \cos((n-1)x) - \cos((n+1)x)dx - \int_{\pi}^{2\pi} \cos((n-1)x) - \cos((n+1)x)dx\right] \\
&= \frac{1}{2\pi}\left[\left[\frac{\sin((n-1)x)}{n-1} - \frac{\sin((n+1)x)}{n+1}\right]_0^{\pi} - \left[\frac{\sin((n-1)x)}{n-1} - \frac{\sin((n+1)x)}{n+1}\right]_{\pi}^{2\pi}\right] \\
&= 0
\end{aligned}
$$

Thus:

$$
f(x) = |\sin(x)| = \frac{2}{\pi} + \frac{2}{\pi}\sum_{n=1,2\ldots}^{\infty} \frac{\Big(\cos((n-1)\pi) - 1\Big)}{(n^2-1)} cos(nx)
$$

Note: For odd n, $\Big(\cos((n-1)\pi) - 1\Big) = 0$. Hence, the expression can be simplified to take only the even values of n.

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

Therefore, the fully rectified sine wave has a DC component of magnitude $\frac{2}{\pi}$.

The input filter is a high pass filter with a cut-off frequency at 9.675 Hz (to three decimal points). This was obtained using the line input resistance from the data sheet which was found to be 35 kΩ [1].

$$f = \frac{1}{2\pi * RC} = \frac{1}{2\pi * 35000 * 470 * 10^{-9}} = 9.675Hz$$

By plotting the frequency and phase responses (Figure 2), it is evident that the DC components of the input are blocked.

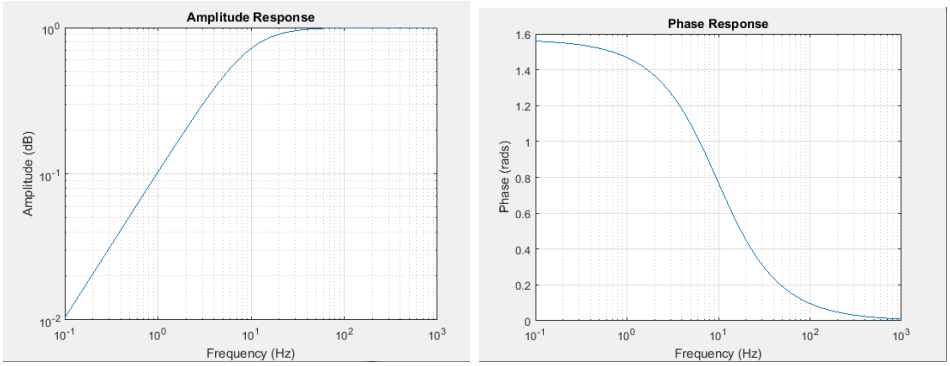

Figure 2: Amplitude and Phase Response of Input Filter

Similarly, the output is also a high pass filter with a pole at 7.189 Hz (to three decimal places).

$$f = \frac{1}{2\pi * RC} = \frac{1}{2\pi * (47000 + 100) * 470 * 10^{-9}} = 7.189Hz$$

Thus, it can be concluded that the DC component of the fully rectified sine wave is blocked, which causes the resulting waveform to be centred at 0V.

1.2 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? You may wish to explain your answer using frequency spectra diagrams. 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?

To understand why the output waveform will only be full-wave rectified if the input is below a certain frequency, the frequency components of both signals must be examined in more detail.

In the frequency domain, the input sine wave is represented by two delta functions at the positive and negative of the frequency of the wave.

$$\mathcal{F}(\sin(2\pi f_0 x)) = \frac{1}{2} i[\delta(f + f_0) - \delta(f - f_0)]$$

Considering again the fourier series of the full-wave rectified sine wave, the frequency domain representation of the wave is found to be a train of quadratically decaying delta functions at all even harmonics of the input sine wave frequency along with a DC component. This is represented mathematically and graphically below:

$$\mathcal{F}(|\sin(2\pi f_0 x)|) = \frac{2}{\pi}\delta(f) - \frac{4}{\pi}\sum_{n=1,2...}^{\infty} \frac{1}{4n^2 - 1}[\delta(f + 2nf_0) + \delta(f - 2nf_0)]$$

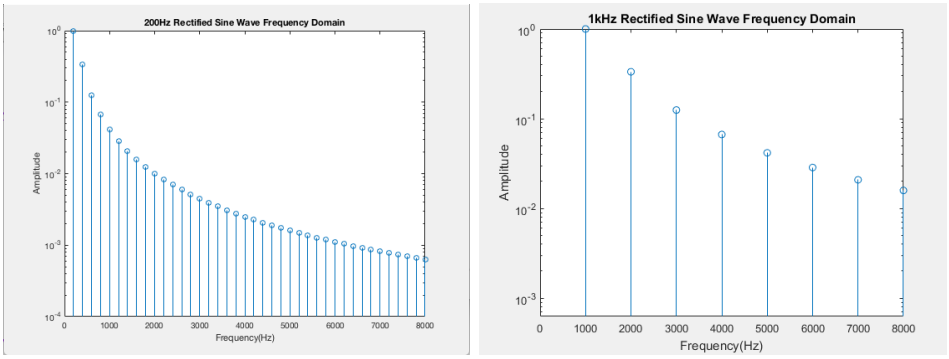

Figure 3: 200Hz and 2kHz fully-rectified sine wave in frequency domain

Theory: An ideal full rectified sine wave will have an infinite number of harmonics as demonstrated by the math above. This is quite similar to an ideal square wave in which, the larger the number of harmonics, the better the approximation to the required signal. As a result, the main task presented at this stage was to determine the number of harmonics that need to be below Nyquist frequency (4 kHz) for a faithful description of a fully rectified sine wave, as the higher harmonics are a lot smaller due to the quadratic decay.

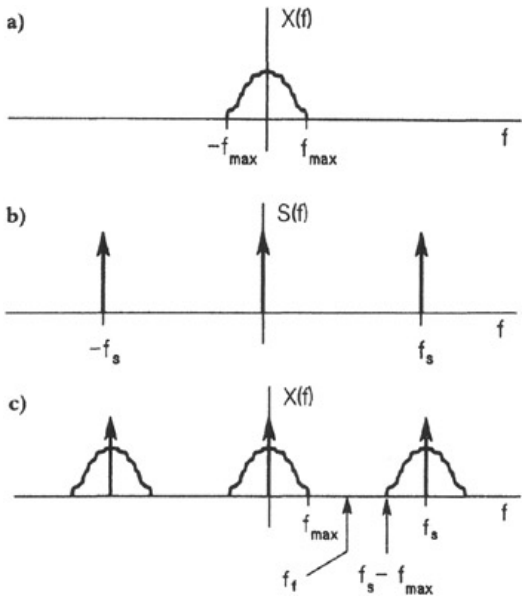

Figure 4: Effect of sampling [2]

From the above figure, it can be seen that multiplying by a train of dirac deltas or impulses spaced out by $\frac{1}{F_s}$, i.e. a sampling train, results in convolution in the frequency domain with another train of impulses centred at multiples (harmonics) of F_s. The result is a periodic spectrum with period F_s.

As a result, any harmonics that occur at frequencies above the Nyquist frequency will be mapped (**aliased**) to frequencies within the baseband. Frequencies between 4kHz and 8kHz will be reflected about the Nyquist frequency (or *folding* frequency) to be within the baseband, for a sampling frequency of 8kHz. Since the spectrum is symmetric about 0 Hz and sam-

pling causes periodicity in the spectrum, the spectrum from -4kHz to 0Hz will be the same as the spectrum from 4kHz to 8kHz (after a 'period' of 8kHz), i.e. a reflected version of the spectrum from 0Hz to 4kHz.

Similarly, the spectrum's infinite and periodic nature means that any harmonics occurring beyond 8kHz will also be observed beyond 0Hz, as contributed from the spectrum centred at -8kHz. As a result, any signal above 8kHz will technically be 'wrapped around' to baseband although it was, in fact, the harmonics that occurred beyond F_s for the spectrum centred at -8kHz or multiples of it.

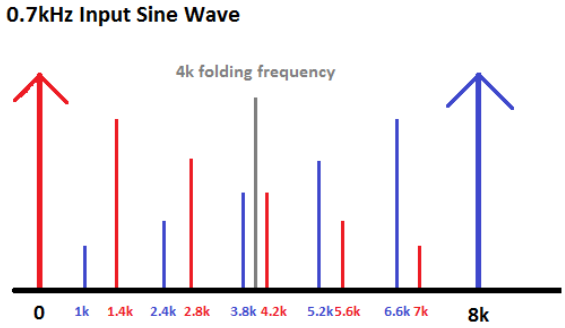

Figure 5: 'Wrap around' the Nyquist for 700Hz Input

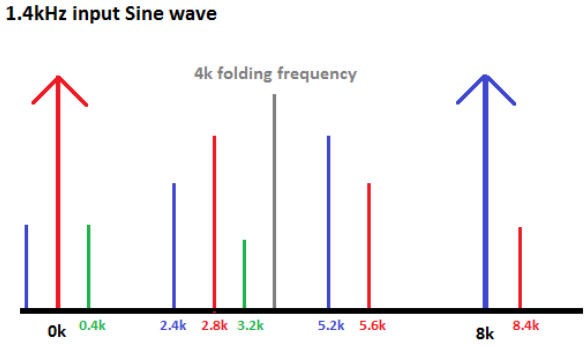

Figure 6: 'Wrap around' the Nyquist and periodicity of spectrum for 1.4kHz Input

For a 700Hz input (Figure 5), the first harmonic will be at 1.4kHz, the second at 2.8kHz and the third at 4.2kHz. The third harmonic will effectively be wrapped around to 3.8kHz as shown in blue as the corresponding -4.2kHz impulse when centred about 8kHz will create an impulse of same amplitude at 3.8kHz. Similarly, the next harmonic at 5.6kHz will be wrapped around to 2.4kHz.

For the 1.4kHz input (Figure 6), the first harmonic will be at 2.8kHz. The second harmonic at 5.6kHz will be wrapped around to 2.4kHz as shown in blue. However, the third harmonic at 8.4kHz will occur at 400Hz, shown in green, as a contribution from the spectrum centred at -8kHz (-8kHz+8.4kHz = 400Hz).

These 'wrap arounds' occur when the sampling is done at the input. However, the anti-aliasing filter at the output of the Codec will filter out components beyond the Nyquist frequency. This filter is digitally implemented to accommodate the various supported sampling frequencies, and is set at 4kHz due to the 8kHz sampling frequency (there is only one analogue anti-aliasing filter in the Codec which is placed at 48kHz to support the maximum sampling frequency of 96kHz [1]). This moves the complexity from the analogue domain to the digital domain without actually having to vary resistor or capacitor values for each sampling frequency.

Frequencies from 0Hz to 35Hz :

Output waveforms corresponding to input signals at very low frequencies such as 35Hz or below are possibly subject to the non-linear phase response of the analogue high-pass filters at the input and output of the AIC23 chip (Figure 2). This manifests itself as a non-uniform group delay which causes different harmonics to be delayed by different amounts. The result is a skewed signal as shown in Figure 7 below.

The extent to which these signals are skewed depends on the number of harmonics that are in the transition band of the filter (between 1/10 and 10 times the cut-off frequency $\sim$10 Hz in this case), meaning the smaller the input frequency, the more skewed the output will be. This is evident when the 5Hz signal on the left of Figure 7 is compared to the 10Hz signal on the right.

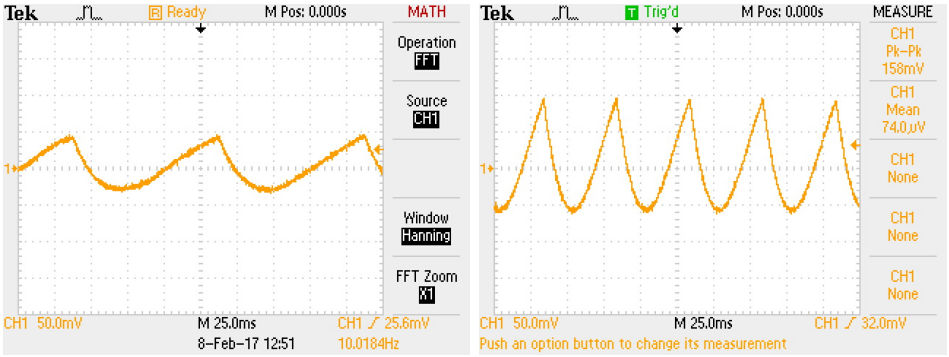

Figure 7: 5Hz and 10Hz fully rectified waveforms

In addition to this, low frequency signals will be subject to reduced gain from the transition band of the amplitude response of the filters (Figure 7). This is apparent when the amplitudes for the two signals above, 5Hz and 10Hz input, are compared. The former has a peak-peak amplitude of 70mV; while the latter, 158mV.

Frequencies from 35Hz to 400Hz:

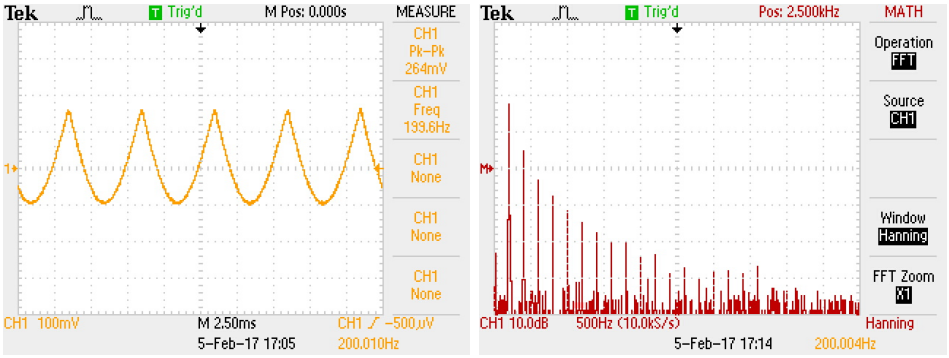

Figure 8: Time and frequency domain outputs for a 100Hz input sine wave

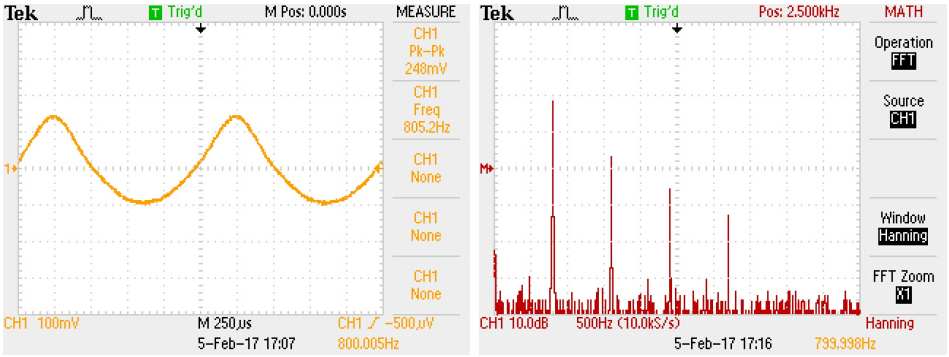

Figure 9: Time and frequency domain outputs for a 400Hz input sine wave

As illustrated above, when a 100Hz sine wave is input, an accurate 200Hz fully-rectified sine wave is output, and 18/19 harmonics are observed before the 4kHz cut-off (Figure 8). However, when a 400Hz sine wave is input, the resulting 800Hz fully-rectified sine wave has only 4/5 harmonics in the frequency domain and the resulting signal has smooth peaks (Figure 9). This is non-ideal, as perfect fully-rectified sine waves should have sharp peaks, similar to that produced by a 100Hz input. Therefore, an empirical lower limit was set on the frequency that would produce an acceptable fully-rectified sine wave and this was set at 5 harmonics within Nyquist frequency, or an input frequency below 400Hz. In theory, the output is quite similar to a fully-rectified sine wave and the limit can be as set as a minimum of 4 harmonics (500Hz input) or even 3 ($\sim$600Hz) depending on how stringent the conditions are for the output. However, to get a clean waveform the limit was set at 5.

Another observation that can be made is that the amplitude of the output seems to grow and shrink periodically. This effect is especially pronounced when there is a harmonic at or very close to the Nyquist frequency due to 4kHz harmonic being read at a slightly lower value (such as 3999.96Hz) causing it to have an amplitude modulation component with a large time period for the envelope (25 seconds in this case).

Frequencies from 400Hz to 1kHz:

For frequencies above 400Hz onwards, the number of harmonics within

the Nyquist are 4 or below. As stated above, up to 500Hz or even 650Hz, the output is quite similar to the required fully-rectified waveform as can be seen for a 500Hz input with the 4th harmonic at Nyquist in Figure 10 below.

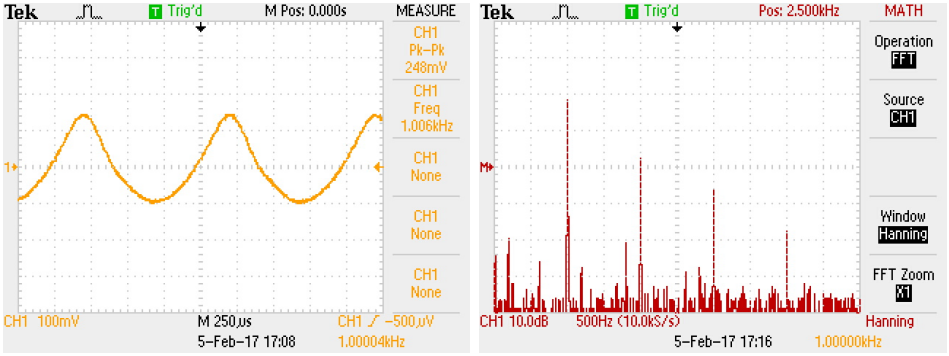

Figure 10: Time and frequency domain outputs for a 500Hz input sine wave

As the number of harmonics within the Nyquist frequency reduces, the output will no longer be sharp as the higher harmonics control features such as sharpness and vertical cut-offs. Although these harmonics are wrapped around to within the Nyquist, they are not at the required frequency (as specified by the Fourier Series equation) for obtaining a fully-rectified sine wave. For instance, in the case of an 800Hz input (Figure 11) the third harmonic will be at 3.2kHz when it is actually meant to be at 4.8kHz. As a result, the output will not be a very good approximation of a fully-rectified sine wave and the output will instead be the resultant sum of all the sine waves in baseband (after anti-alias/reconstruction filtering at the output).

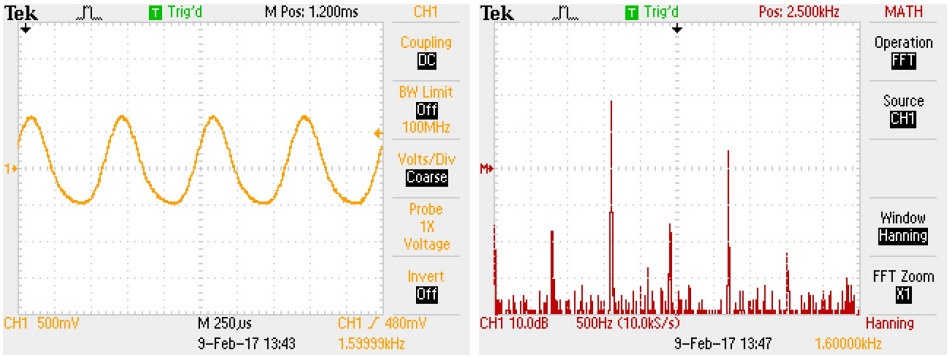

Figure 11: Time and frequency domain outputs for a 800Hz input sine wave

As the number of 'true' harmonics (not wrapped around) increases, the sharpness of the edges and the curvature of the sides of the inverted 'U' keep gradually reducing until the edges are no longer sharp and the curvature has inverted towards the edges to result in a sine wave at 1kHz (Figure 12). This occurs as there is only one unmapped harmonic within the baseband and as a result, the output will be a sine wave at 2kHz and the component at Nyquist causes behaviour similar to amplitude modulation as the amplitude grows and shrinks.

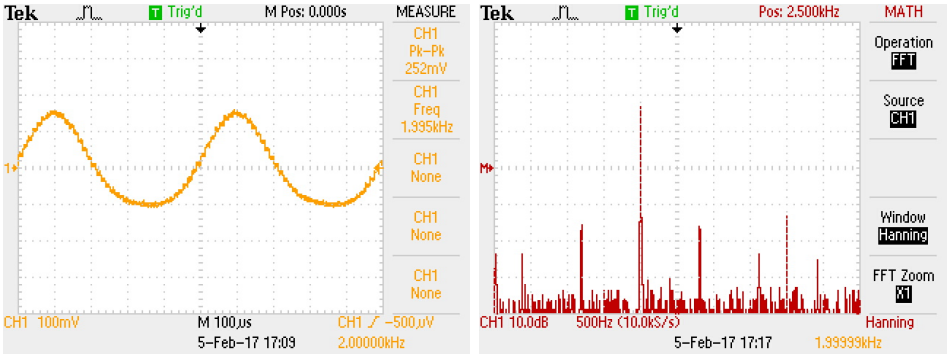

Figure 12: Time and frequency domain outputs for a 1kHz input sine wave

Frequencies from 1kHz to 1.8kHz:

For input frequencies beyond 1kHz, there will only be one 'true 'harmonic in baseband and hence the output will be a sine wave. However, the harmonics will wrap around to baseband and add on to the signal in a more pronounced way than they did for frequencies below 1kHz. This is because the second harmonic itself wraps around and due to its significant amplitude, adds on to the baseband signal. The output was still at the correct frequency as when multiple harmonics sum together the resultant frequency would be the least common multiple of them i.e. the first harmonic's frequency with a varying envelope.

If the input was at 1.5kHz, the first harmonic would be at 3kHz, the second at 2kHz (wrap around of 6kHz) and the third one at 1kHz (periodic contribution of 9kHz from -8kHz). This can be seen in Figure 13 below:

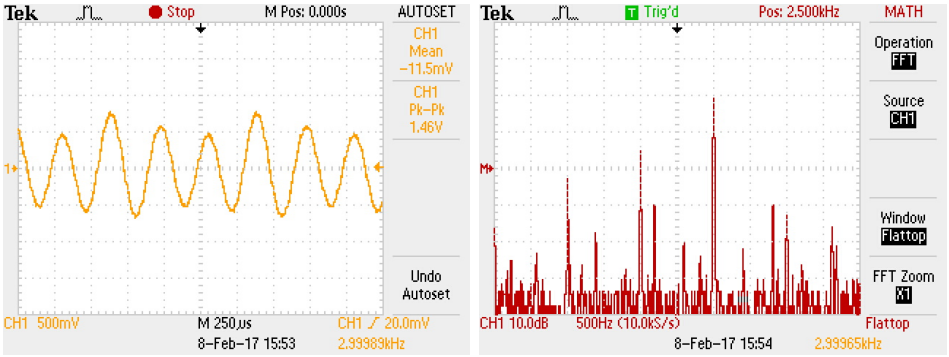

Figure 13: Time and Frequency domain outputs for a 1.5kHz input sine wave

To further the mathematical justification of this observation, the time domain components of the three large harmonics observed were plot in Wolfram Alpha with their corresponding amplitudes being obtained by approximating the logarithmic amplitudes of the spectra. The wave plotted was: $(0.508\cos(x) + 0.5843\cos(2x) + cos(3x))$ (Figure 14) which was quite similar to the observed waveform in Figure 13.

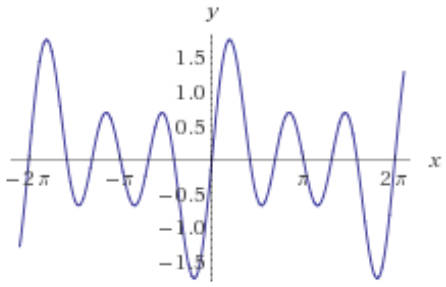

Figure 14: Mathematical synthesis of output waveform for 1.5kHz input

Frequencies from 1.8kHz to 2kHz:

The input frequencies in this range are quite identical in behaviour to the previous range as they will also be amplitude modulated. However, as the spectrum from 4kHz to 8kHz is the same as that from -4kHz to 0Hz, the first harmonic will have an image close to Nyquist (slightly above it) that will not be cut out by the anti-aliasing/reconstruction filter response at the output as it does not decay fast enough, i.e. the filtering is non-ideal. This can be

seen clearly for the case of a 1.9kHz input signal in Figure 15.

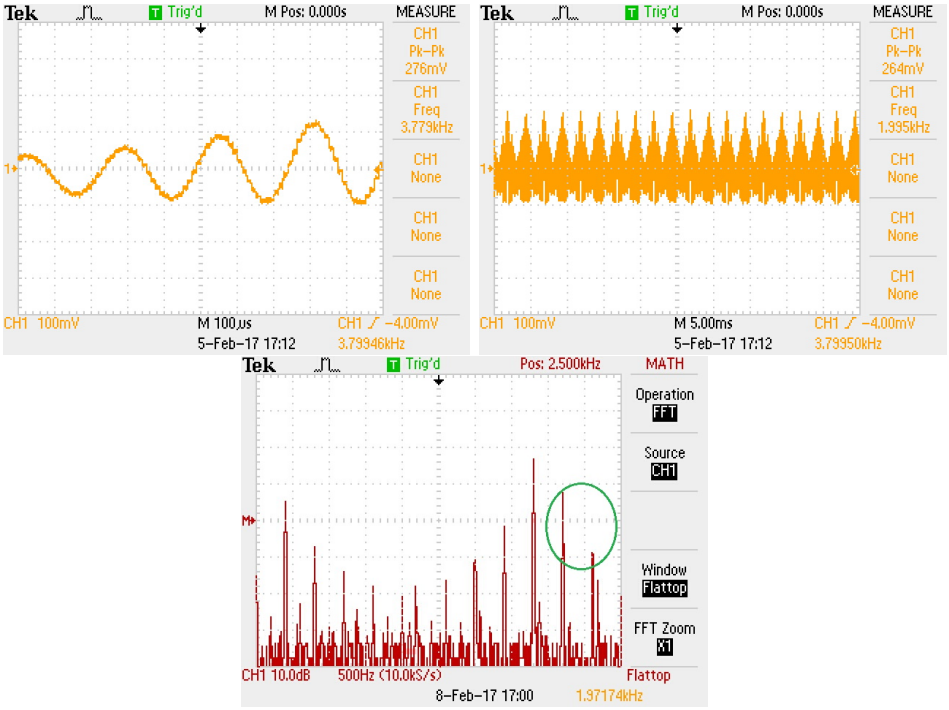

Figure 15: Carrier, Envelope and Frequency domain signals for 1.9kHz input

The repeated harmonics above Nyquist are circled in green. The presence of these signals seems to add on to the amplitude modulated nature of the output as can be seen in the figure above. The output is an AM signal with a sinusoidal carrier varying at 3.8kHz (only harmonic within baseband) and an envelope that appears to be rectified.

It is also noteworthy to observe that the harmonics occur alternatively on the right then the left till they reach the center. The first harmonic is at 3.8kHz, the second at 400Hz (7.6kHz wrapped around), third at 3.4kHz (11.4kHz-8kHz), fourth at 800Hz (15.2kHz-8kHz=7.2kHz wrapped around) and so on.

Frequencies from 2kHz to 4kHz:

When the input sine wave is between 2kHz and 4kHz, none of the true harmonics occur in the baseband; those that do, correspond to harmonics belonging to spectrums not centred at the origin. The same observations are obtained as with the above mentioned frequency ranges but wrapped around this time, i.e. the output for an input frequency of 4000 - f will be the same as that for input frequency f where $0 \leq f \leq 2000$. For instance, the output for an input of 3.2kHz (4000-800Hz) will be the same as that for an input of 800Hz.

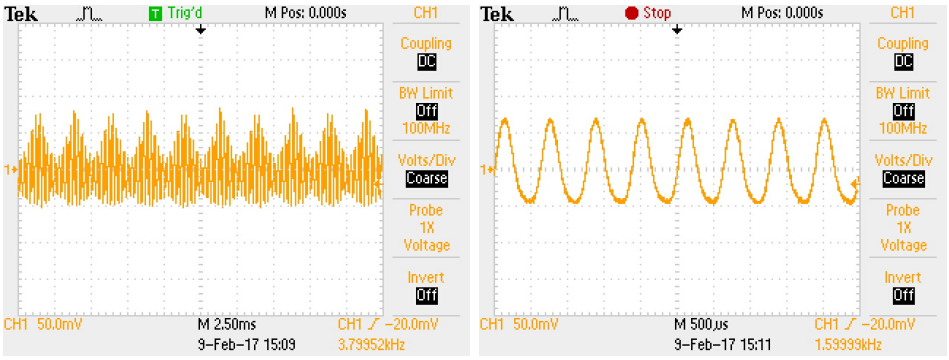

Figure 16: Outputs for input frequencies of 2.1kHz and 3.2kHz

If the signals in Figure 16 (2.1kHz and 3.2kHz) are compared to those in Figures 15 (1.9kHz) and 11 (800Hz), it can be seen that they are identical, thereby confirming the above observation.

The reason for this wrap around in results is due to the mapping of all harmonics to baseband. For instance, if the input frequency is 3.8kHz, the first harmonic will be at 400Hz(7.6kHz), the second at 800Hz(15.2kHz-8kHz=7.2kHz wrapped around) and so on, which is identical to position of the harmonics for a 200Hz input.

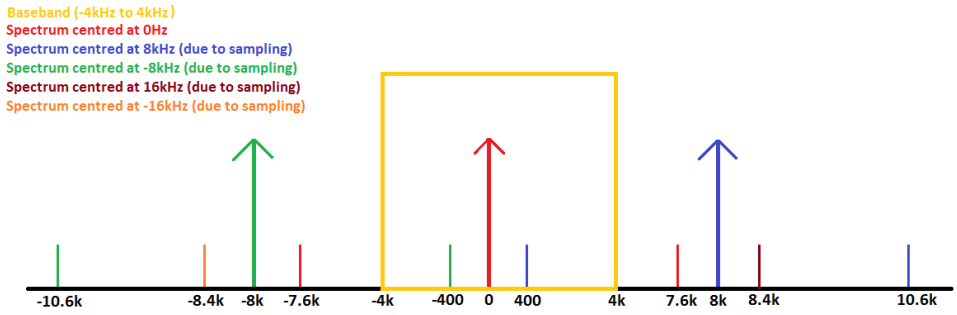

Figure 17: Frequency domain illustration of "wrap around" effect for 3.8kHz

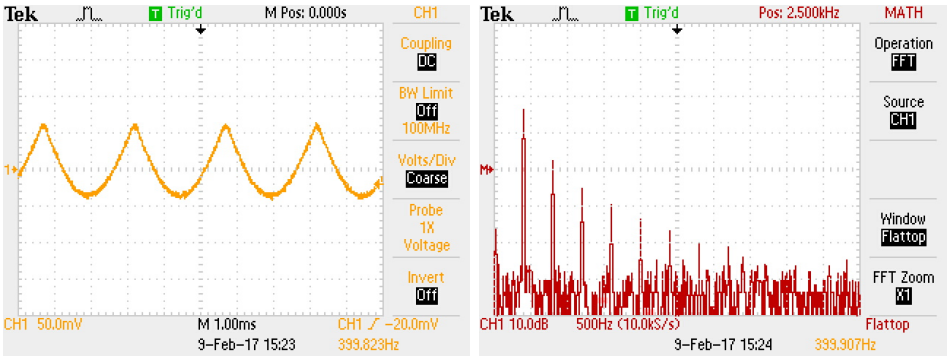

Figure 18: Time and frequency domain outputs for a 3.8kHz input sine wave

The shape of the output waveform in Figure 18 is the same as that of a 200Hz input sine wave. However, the former was of a smaller amplitude, i.e. 310mV on average compared to the 400mV for a 200Hz input.

Frequencies above 4kHz:

As the anti-aliasing filter at the input is not ideal, i.e. the decay was not fast enough, some input signals above 4kHz (up to 4.2kHz) reached the output with their amplitudes reduced significantly. Beyond 4.2kHz, no signal was observed at the output as the filter cut them out.

2 Operation of the Code

2.1 Data sampling using interrupts

The aim of this lab is to utilise interrupt driven code to improve upon the performance of the polling code used previously.

It is important to note the difference between polling and interrupts. Whilst both allow a computer to interact with an external device by vectoring from the running program to service the device, polling is done within the software, and interrupts, generally through hardware.

In polling, the code checks the readiness of an external device in a synchronous fashion. If this external device requires servicing, the code will vector to the appropriate instructions. There are two inefficiencies to this technique. Firstly, time may be wasted if servicing is required before the code is due to check whether or not the external device is ready. Secondly, the code may be stalled by checking for readiness when the external device does not require servicing.

Hardware driven interrupts, on the other hand, can asynchronously cause the program to vector to the Interrupt Service Routine (ISR) whenever an event occurs by sending a Interrupt Request (IRQ) to the processor, and thereby avoid the two aforementioned inefficiencies. Interrupts are therefore significantly more efficient than polling, time-wise, but require additional hardware such as shadow registers to store the context of the code before the interrupt, in order to be re-loaded after the interrupt has been serviced.

The DSP chip in use has 16 hardware interrupts in descending order of priority. The first hardware interrupt is reserved for the reset button on the DSK, the second for non-maskable interrupts (NMI) and the subsequent two for other purposes that cannot be accessed or triggered by the programmer. The rest of the interrupts are available for user requirements. However, there are a lot of interrupt sources but a limited number(12) of hardware interrupts and hence, mapping must be done to these hardware interrupts. For the following tasks the audio port interrupt was implemented on the fifth interrupt, HWI_INT4.

To link the physical interrupt with the ISR, an association was made within the DSP BIOS Configuration by naming the interrupt function with the same name as the ISR function in the code. The name chosen was _ISR_AIC. As the port connecting the AIC23 audio port and the DSP is the MCBSP serial port, the interrupt source was set as either `MCSP_1_Transmit` or `MCSP_1_Receive`.

2.2 Exercise 1: Interrupt service routine

In the provided code, it was observed that there are major differences from the code provided in Lab 2.

First of all, many of the global declarations such as `PI` and `table` are not included or required as the data is being read rather than generated and transmitted. The data is read through a standard 3.5mm audio jack, and can either be generated by hardware or the signal generator software provided. For simplicity, the software was chosen.

It is also important to note that `init_hardware()` has been supplemented with `init_HWI()` which configures the interrupt settings with certain IRQ functions (must be customised according to the number of interrupts used).

In addition to this, `main()` (Figure 19) has been greatly changed. Where the function used to run an infinite loop calling `sinegen()` to calculate the sine wave value to be output, the loop now simply does nothing. When an interrupt occurs, the context of the while loop is saved, and the ISR code is then vectored to. After the interrupt source has been serviced, the while loop is then re-loaded from the position it was pre-empted.

The ISR code, `ISR_AIC()`, uses a specially written function `mono_read_16Bit()` to read in a sample from the Codec. It simply reads Left and Right samples from the audio port, divides the amplitude by two and then sums them together to provide a mono input. The interrupt source is set as `MCSP_1_Receive`, as for this exercise, the interrupt is triggered by the presence of an input at the audio input port.

This input is then rectified by passing the absolute value to the output,

which is done by another special function `mono_write_16Bit(samp)`.

```
67  void main(){
68
69
70      // initialize board and the audio port
71    init_hardware();
72
73    /* initialize hardware interrupts */
74    init_HWI();
75
76    /* loop indefinitely, waiting for interrupts */
77    while(1)
78    {};
79
80  }
```

Figure 19: `main()` for exercise 1

```
120  void ISR_AIC()
121  {
122      mono_write_16Bit(abs(mono_read_16Bit()));
123  }
```

Figure 20: ISR for exercise 1

The `abs()` function is used in this case as it simply performs the same function as an `if` statement along with sign inversion. The difference is that since the function is quite simple, branching to it and returning back would take more time than the actual processing performed by it. However, if the compiler optimisation level is increased to 3 or above, small functions such as `abs()` will be replaced with *inline* versions which would be identical to using an `if` statement instead.

2.3 Exercise 2: Transmitting Data

In exercise 2, the previous program is modified so that within the ISR, a sine wave is generated using the look-up table method from Lab 2.

The ISR must therefore be modified and the interrupt source must be changed such that it will be entered when the MCBSP serial port buffer is

ready to write a sample rather than when it has received one. To do this, the ISR must be associated with `MCSP_1_Transmit` rather than `MCSP_1_Receive` within the configuration file.

Although the code from Lab 2 can largely be pasted in, a few changes had to be made. Firstly, the gain in Lab 2 was set at 2100000000, a decimal number close to the highest possible 32 bit binary number (2,147,483,647), and the signals were stereo. In this lab, the outputs are sent to the left and right audio ports as 16-bit mono signals and thus the gain must be adjusted accordingly. 32767 was chosen as any large number below the maximum 16-bit positive number (32767) will do.

The `ISR_AIC()` function must be altered from the previous exercise as `mono_read16Bit()` is not required.

Another requirement of the task was to full-wave rectify the signal. This could be done simply by passing the absolute value (disadvantages of the `abs()` function explained in the section above) of `sample`, which contains the value of the lookup take at the required index, to the output. Alternatively, one could exploit the symmetry of the sine wave and modify `sine_init()` to fill the lookup table with values of only the positive half of the sine wave as the output is full-wave rectified. This would improve the resolution of the output as demonstrated in Figure 21. However, `sinegen()` must in turn be altered to skip twice as many indices each cycle in order to ensure that `sine_freq` still sets the frequency of the output. The resolution could be further improved by filling the lookup table with values of only a quarter of the sine wave.

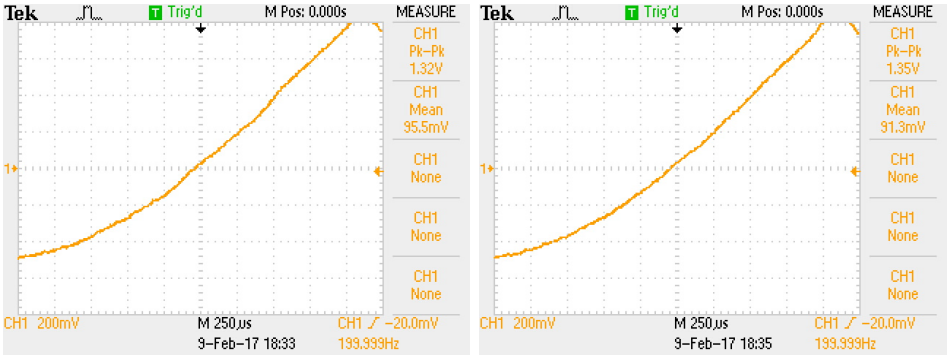

Figure 21: Side by side comparison of resolution improvement at 100Hz

An attempt was made to try and make the output independent of the sampling frequency but it did not work as the code for Lab 3 was not written to allow real-time changes in the actual sampling frequency used by the DSK6713 chip and the AIC23 audio Codec. Hence, this endeavour was dropped.

The results obtained for Exercise 2 were identical to those of Exercise 1 explained in section 1.2 above. However, the code of Exercise 2 can not produce an output of frequency larger than 4kHz, as aliasing occurs automatically while skipping through the entries in `table`, illustrated in Figure 22. Hence, an output is expected for all frequencies.

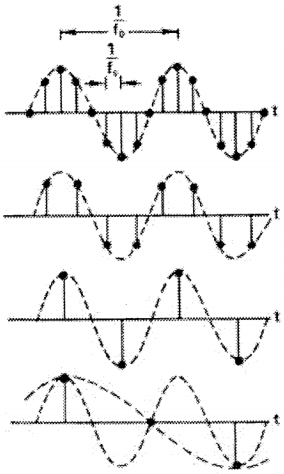

Figure 22: Demonstration of aliasing while skipping indices in `table`

However, this was not the case, on account of `index` incrementing beyond twice `SINE_TABLE_SIZE` at frequencies above 4kHz. The `if` statement alone would not suffice in bringing `index` back to within its range. This was solved by using a `while` loop instead, as seen in Figure 23. Consequently, outputs were obtained for any frequencies that were entered. The results were again the same for inputs of frequencies $4000 - f$ and f, where $0 \leq f \leq 2000$ *but* $f \equiv f_{input}(mod\ 4000)$. For instance, if f_{input} is 15kHz, $f = 15,000(mod\ 4000) = 3000 = 4000 - 1000$ and hence the results obtained will be same as those for a 1kHz input.

```
158     index += (2*SINE_TABLE_SIZE*sine_freq/sampling_freq); //offset definition
159     while(index>=SINE_TABLE_SIZE)
160     {
161         //while index exceeds the table size, table size is subtracted from index
162         index-=SINE_TABLE_SIZE ;
163     }
```

Figure 23: Altered code

References

[1] Texas Instruments. *TLV320AIC23b Stereo Audio Codec Data Manual*. 2004. [Online; accessed 2017-02-07].

[2] Robert A. Witte. Sampling theorem. `http://www.globalspec.com/reference/77961/203279/4-5-sampling-theorem`, 2001. [Online; accessed 2017-02-08].

Appendices

A Full `intio.c` code for Exercise 1

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

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

                                LAB 3: Interrupt I/O

                                ********* 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>

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

```c
// 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 ISR_AIC(void);
/********************************** Main routine ***********************************/
void main(){


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

  /* initialize hardware interrupts */
  init_HWI();

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

}

/********************************* init_hardware() *********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

    /* Function below sets the number of bits in word used by MSBSP (serial port) for
    receives from AIC23 (audio port). We are using a 32 bit packet containing two
    16 bit numbers hence 32BIT is set for  receive */
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

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

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


}

/********************************* init_HWI() ************************************/
void init_HWI(void)
{
    IRQ_globalDisable();            // Globally disables interrupts
    IRQ_nmiEnable();                // Enables the NMI interrupt (used by the 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 ISR_AIC()
{
    mono_write_16Bit(abs(mono_read_16Bit()));
}
```

Listing 2: Full `intio.c` for Exercise 1

B Full `intio.c` code) without resolution improvement for Exercise 2

```
/****************************** 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>
#include "helper_functions_polling.h"

#define SINE_TABLE_SIZE 256
#define PI 3.141592653589793


/* Left and right audio channel gain values, calculated to be less than signed 32 bit
 maximum value. */
Int32 L_Gain = 32767;

float sine_freq = 1000.0;
int sampling_freq = 8000.0;
float sample;
float index=0;
float table[SINE_TABLE_SIZE];

/********************************* 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 ISR_AIC(void);
float sinegen(void);
void sine_init();
/********************************** Main routine **************************************/
```

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


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

  /* initialize hardware interrupts */
  init_HWI();

  sine_init();

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

}

/********************************** init_hardware() ********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

    /* Function below sets the number of bits in word used by MSBSP (serial port) for
    receives from AIC23 (audio port). We are using a 32 bit packet containing two
    16 bit numbers hence 32BIT is set for  receive */
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

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

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


}

/********************************** init_HWI() *************************************/
void init_HWI(void)
{
    IRQ_globalDisable();            // Globally disables interrupts
    IRQ_nmiEnable();                // Enables the NMI interrupt (used by the 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 ISR_AIC()
{
    sample = sinegen();

    sample = (Int16) (sample * L_Gain);
    mono_write_16Bit(abs(sample));
}
```

```
150  float sinegen(void)
151  {
152      /*This code produces a fixed sine of 1KHZ (if the sampling frequency is 8KHZ)
153      using a digital filter.*/
154      // temporary variable used to output values from function
155      float wave;
156
157      //index must skip samples in order to maintain correct interpreted frequency
158      index += (SINE_TABLE_SIZE*sine_freq/sampling_freq); //offset definition
159      if(index>=SINE_TABLE_SIZE)
160      {
161          //when index exceeds the table size, table size is subtracted from index
162          index-=SINE_TABLE_SIZE ;
163      }
164
165      //set the output as the value of the sine wave, stored in table
166      wave = table[(int)floor(index)];
167
168      return(wave);
169
170  }
171
172  //fills table with values of 256 equally spaced points around the sine wave
173  void sine_init()
174  {
175      int x;
176      for(x=0; x<SINE_TABLE_SIZE; x++){
177          table[x] = sin((2*PI*x)/SINE_TABLE_SIZE);
178      }
179  }
```

Listing 2: Full `intio.c` code without resolution improvement Exercise 2

C Full `intio.c` code with half-wave resolution improvement for Exercise 2

```
/****************************** 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>
#include "helper_functions_polling.h"

#define SINE_TABLE_SIZE 256
#define PI 3.141592653589793


/* Left and right audio channel gain values, calculated to be less than signed 32 bit
 maximum value. */
Int32 L_Gain = 32767;

float sine_freq = 1000.0;
int sampling_freq = 8000.0;
float sample;
float index=0;
float table[SINE_TABLE_SIZE];

/******************************* 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 ISR_AIC(void);
float sinegen(void);
void sine_init();
```

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


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

  /* initialize hardware interrupts */
  init_HWI();

  sine_init();

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

}

/********************************** init_hardware() ********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

    /* Function below sets the number of bits in word used by MSBSP (serial port) for
    receives from AIC23 (audio port). We are using a 32 bit packet containing two
    16 bit numbers hence 32BIT is set for  receive */
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

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

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


}

/********************************** init_HWI() *************************************/
void init_HWI(void)
{
    IRQ_globalDisable();            // Globally disables interrupts
    IRQ_nmiEnable();                // Enables the NMI interrupt (used by the 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 ISR_AIC()
{
    sample = sinegen();

    sample = (Int16) (sample * L_Gain);
    mono_write_16Bit(abs(sample));
}
```

```
150  float sinegen(void)
151  {
152      /*This code produces a fixed sine of 1KHZ (if the sampling frequency is 8KHZ)
153      using a digital filter.*/
154      // temporary variable used to output values from function
155      float wave;
156
157      //index must skip samples in order to maintain correct interpreted frequency
158      index += (2*SINE_TABLE_SIZE*sine_freq/sampling_freq); //offset definition
159      if(index>=SINE_TABLE_SIZE)
160      {
161          //when index exceeds the table size, table size is subtracted from index
162          index-=SINE_TABLE_SIZE ;
163      }
164
165      //set the output as the value of the sine wave, stored in table
166      wave = table[(int)floor(index)];
167
168      return(wave);
169
170  }
171
172  //fills table with values of 256 equally spaced points around half a sine wave
173  void sine_init()
174  {
175      int x;
176      for(x=0; x<SINE_TABLE_SIZE; x++){
177          table[x] = sin((PI*x)/SINE_TABLE_SIZE);
178      }
179  }
```

Listing 3: Full `intio.c` code with half-wave resolution improvement for Exercise 2

D Full `intio.c` code with quarter-wave resolution improvement for Exercise 2

```
/****************************** 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>
#include "helper_functions_polling.h"

#define SINE_TABLE_SIZE 256
#define PI 3.141592653589793

/* Left and right audio channel gain values, calculated to be less than signed 32 bit
 maximum value. */
Int32 L_Gain = 32767;

float sine_freq = 1000.0;
int sampling_freq = 8000.0;
float sample;
float index=0;
float table[SINE_TABLE_SIZE];
//sign dictating the direction in which index increments
int inc_sign = 1;

/********************************* 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 ISR_AIC(void);
float sinegen(void);
void sine_init();
```

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


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

  /* initialize hardware interrupts */
  init_HWI();

  sine_init();

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

}

/********************************** init_hardware() ********************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the AIC23 codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

    /* Function below sets the number of bits in word used by MSBSP (serial port) for
    receives from AIC23 (audio port). We are using a 32 bit packet containing two
    16 bit numbers hence 32BIT is set for  receive */
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);

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

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


}

/********************************** init_HWI() *************************************/
void init_HWI(void)
{
    IRQ_globalDisable();            // Globally disables interrupts
    IRQ_nmiEnable();                // Enables the NMI interrupt (used by the 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 ISR_AIC()
{
    sample = sinegen();

    sample = (Int16) (sample * L_Gain);
    mono_write_16Bit(abs(sample));
}
```

```
150  float sinegen(void)
151  {
152      /*This code produces a fixed sine of 1KHZ (if the sampling frequency is 8KHZ)
153      using a digital filter.*/
154      // temporary variable used to output values from function
155      float wave;
156
157      //index must skip samples in order to maintain correct interpreted frequency
158      index += (4*SINE_TABLE_SIZE*sine_freq/sampling_freq); //offset definition
159      if(index>=SINE_TABLE_SIZE)
160      {
161          //when index exceeds the table size, subtract table size from index to continue the wave
162          index = 2*SINE_TABLE_SIZE - index -1;
163          inc_sign = -1;
164      }
165      if(index<=0)
166      {
167         index = -index;
168         inc_sign = 1;
169      }
170
171      return(wave);
172
173  }
174
175  //fills table with values of 256 equally spaced points around the sine wave
176  void sine_init()
177  {
178      int x;
179      for(x=0; x<SINE_TABLE_SIZE; x++){
180          table[x] = sin((PI*x/2)/SINE_TABLE_SIZE);
181      }
182  }
```

Listing 4: Full `intio.c` code with quarter-wave resolution improvement for Exercise 2