`

RTDSP Lab 3 Report

Zheyuan Li (CID: 01181358)
Guilin Huang (CID: 01237662)

Contents:

Answers for the 2 Questions

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

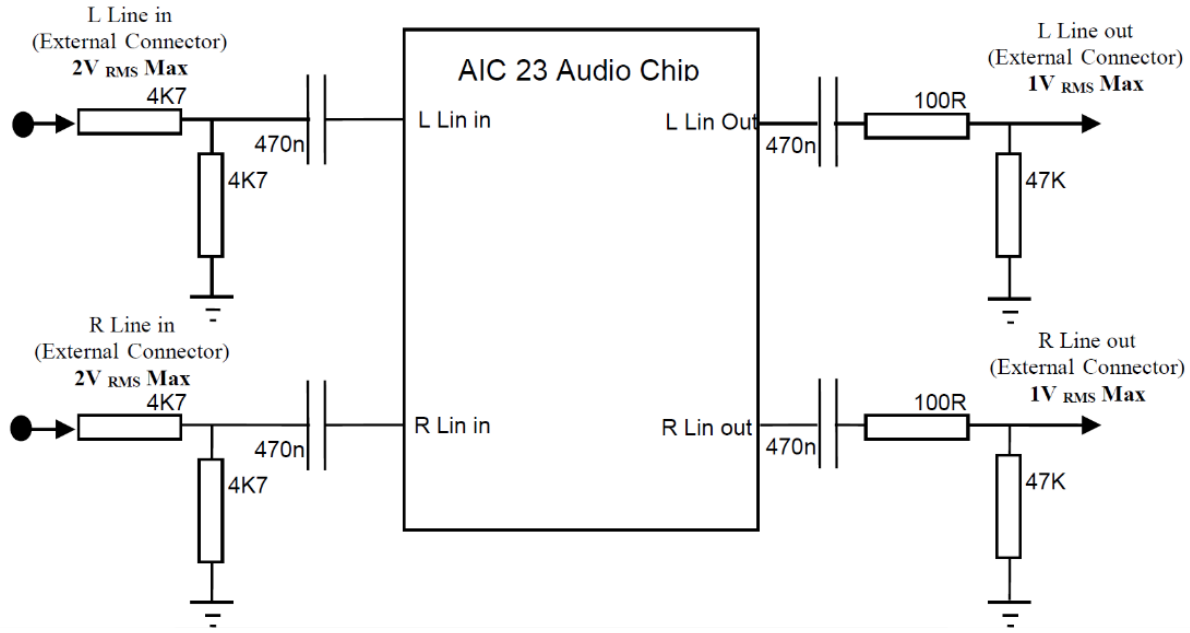


Figure 1: AIC23 Audio chip external components

In the AIC23 audio chip, there is a capacitor in each of the L and R "Lin out" as the red circles marked in the Figure 1. These **capacitors** are in series with the output signal of left and right channel. Each of them basically **blocks the DC components** of the analogue signals generated from the AIC23 audio chip and acts like an RC high pass filter, hence, the graph we observed from scope is the **AC variation.**

Q: 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 3.8 kHz? Can you explain what is going on?

To know the reason why there is only a full-wave rectified sine wave below a certain frequency, we need to understand the frequency components of the rectified sine wave. The mathematical representation of a rectified sine wave is simply $|\sin(\omega t)|$, where ω is the angular frequency which is equivalent to $2\pi f$, and f is the frequency of input signal. Here are some simple derivations of the frequency decomposition of a rectified sine wave.

The Fourier Series:

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

and Euler formulae:

$$a_0 = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\,dx$$
$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(nx)\,dx$$
$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(nx)\,dx$$

where a_0 is the DC component, a_n and b_n are the Fourier coefficient. Basically, the rectified sine wave is an even function because it is symmetrical with respect to y-axis (Figure 2). In another word, the rectified sine wave could be seen as an infinite even extension of a half sine wave on [0, π]. This also means the Fourier coefficient b_n of the rectified wave is zero.

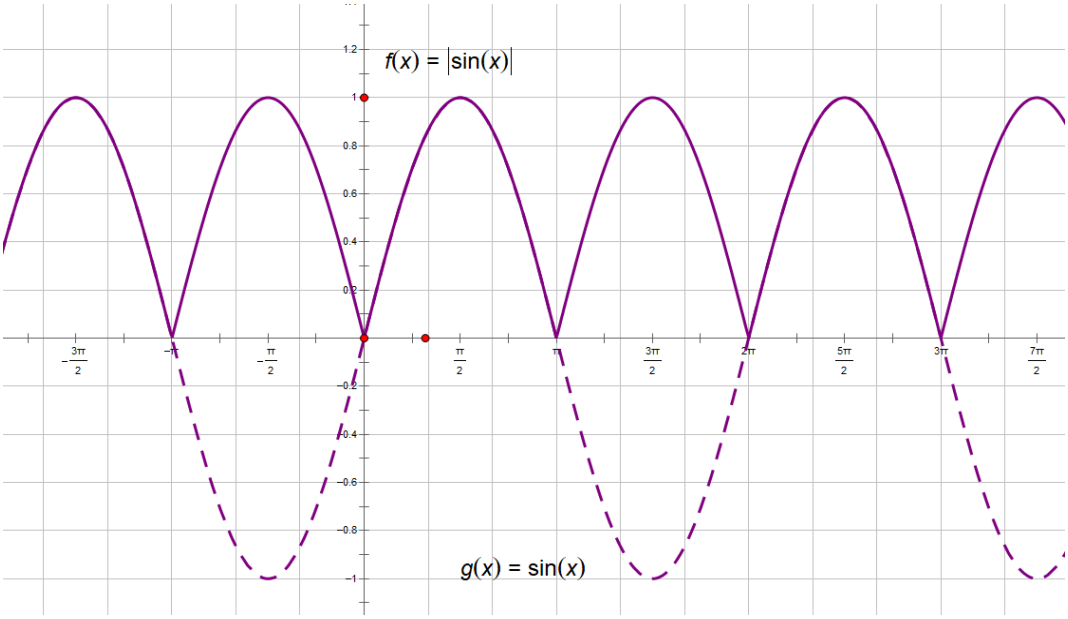


Figure 2

Firstly, we calculate a_0:

$$\begin{aligned} a_0 &= \frac{1}{\pi}\int_{-\pi}^{\pi} |\sin(\omega t)|\,d(\omega t) \\ &= \frac{1}{\pi}\int_{-\pi}^{0} -\sin(\omega t)\,d(\omega t) + \frac{1}{\pi}\int_{0}^{\pi} \sin(\omega t)\,d(\omega t) \\ &= \frac{2}{\pi}\int_{0}^{\pi} \sin(\omega t)\;\,d(\omega t) \\ &= \frac{2}{\pi}[-\cos(\omega t)]_0^{\pi} \\ &= \frac{\pi}{4}. \end{aligned}$$

then a_n:

$$\begin{aligned}
a_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} |\sin(\omega t)|\, d(\omega t) \\
&= \frac{1}{\pi}\int_{-\pi}^{0} -\sin(\omega t)\cos(n\omega t)\, d(\omega t) + \frac{1}{\pi}\int_{0}^{\pi} \sin(\omega t)\cos(n\omega t)\, d(\omega t) \\
&= \frac{2}{\pi}\int_{0}^{\pi} \cos(\mathrm{n}\omega\mathrm{t})\cdot\sin(\omega t)\, d(\omega t) \\
&= \frac{1}{\pi}\int_{0}^{\pi} \{\sin[(n+1)\omega t] - \sin[(n-1)\omega t]\}\, d(\omega t) \\
&= \frac{1}{\pi}\int_{0}^{\pi} \sin[(n+1)\omega t]\, d(\omega t) - \frac{1}{\pi}\int_{0}^{\pi} \sin[(n-1)\omega t]\, d(\omega t) \\
&= \frac{1}{\pi}\left[\frac{-\cos[(n+1)\omega t]}{n+1}\right]_0^{\pi} - \frac{1}{\pi}\left[\frac{-\cos[(n-1)\omega t]}{n-1}\right]_0^{\pi} \\
&= \frac{1}{\pi}\left\{\frac{-\cos[(n+1)\pi]-1}{n+1}\right\} + \frac{1}{\pi}\left\{\frac{\cos[(n+1)\pi]-1}{n-1}\right\} \\
&= \frac{1}{\pi}\left[\frac{-(-1)^{n+1}-1}{n+1} + \frac{(-1)^{n-1}-1}{n-1}\right] \\
&= \begin{cases} 0, & n\ odd \\ -\frac{4}{\pi}\frac{1}{n^2-1}, & n\ even. \end{cases}
\end{aligned}$$

Therefore, we can rewrite:

$$|\sin(\omega t)| = \frac{2}{\pi} - \frac{4}{\pi}\sum_{n\ even}^{\infty} \frac{\cos(n\omega t)}{n^2-1}.$$

Because the DC component is blocked by the capacitor, the actual output would be:

$$\begin{aligned}
f(\omega t) &= -\frac{4}{\pi}\sum_{n\ even}^{\infty} \frac{\cos(n\omega t)}{n^2-1} \\
&= -\frac{4}{\pi}\left[\frac{\cos(2\omega t)}{3} + \frac{\cos(4\omega t)}{15} + \frac{\cos(6\omega t)}{35} + \frac{\cos(8\omega t)}{63} + \cdots\right]. \quad (*)
\end{aligned}$$

We could deduce from the result that if there is an input signal generated from the computer and send to the DSK board through 3.5mm audio jack, after rectification, the output signal would be a rectified sine wave without DC component (shown in Figure 3 and Figure 4).

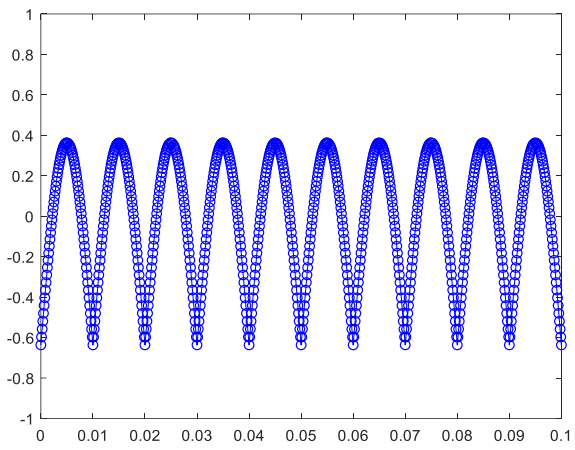


Figure 3: Rectified 50Hz wave in MATLAB

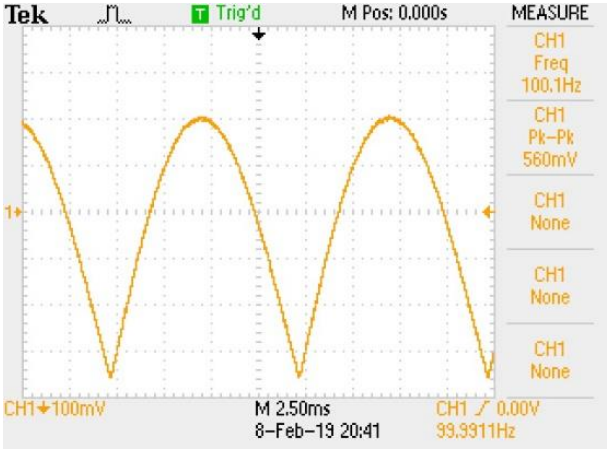


Figure 4: Rectified 50Hz wave on oscilloscope

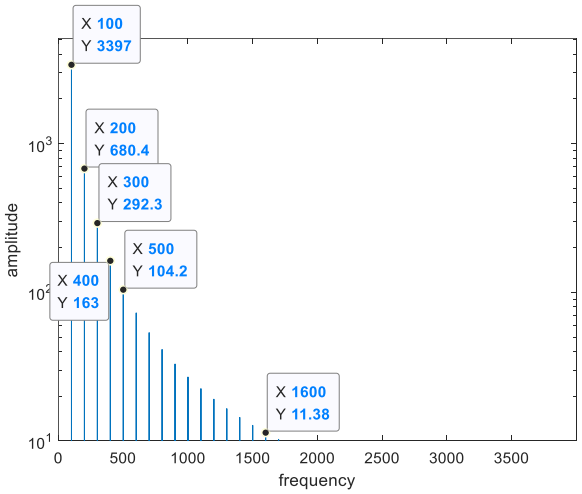


Figure 5: FFT in MATLAB

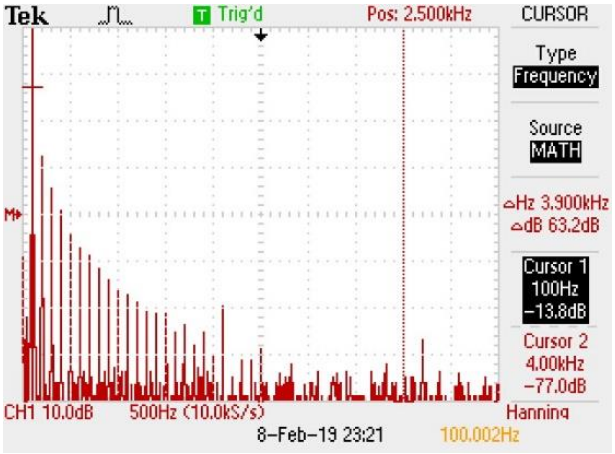


Figure 6: FFT in oscilloscope

From our calculation (*), the rectified wave should contain frequencies of 2, 4, 6, 8 etc. times the original sine frequency. For example, the original sine wave frequency is 50Hz and the sampling frequency is 8kHz. After rectification, if we plot the FFT of the output wave in MATLAB (Figure 5), there are several peaks observed at 100Hz, 200Hz, 300Hz etc. In the oscilloscope (Figure 6) we can see peaks at 100Hz, 200Hz, 300Hz etc. as well which match the theoretical thought.

Note: For simplicity, from now on in the report, the frequency which is twice the original sine wave frequency is denoted as the **fundamental frequency**. The frequency at 4 times the original frequency is denoted 2nd **harmonic**, frequency at 6 times the original frequency is 3rd harmonic and so on. The cursor 1 in the FFT graph in the screenshots of the oscilloscope always indicates the fundamental frequency. Cursor 2 always indicates the Nyquist frequency which is always 4kHz in this lab session.

When the input signal frequency is very low (50Hz), the output wave contains frequencies up to 16th harmonic. They are separated by 100Hz each other. However, the higher order of the harmonic is, the lower amplitude that harmonic has, the smaller effect of that harmonic contribute to the waveform.

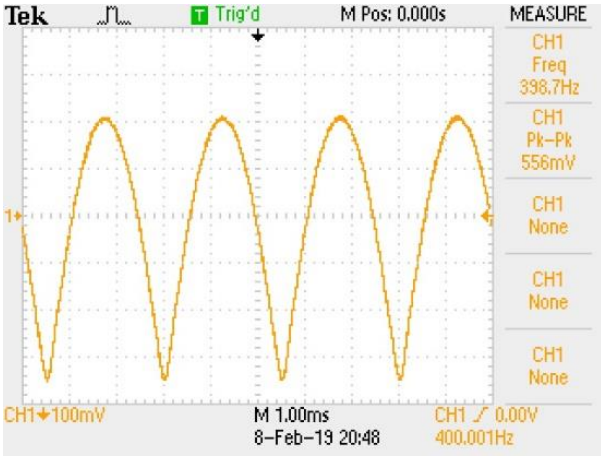


Figure 7: 200Hz

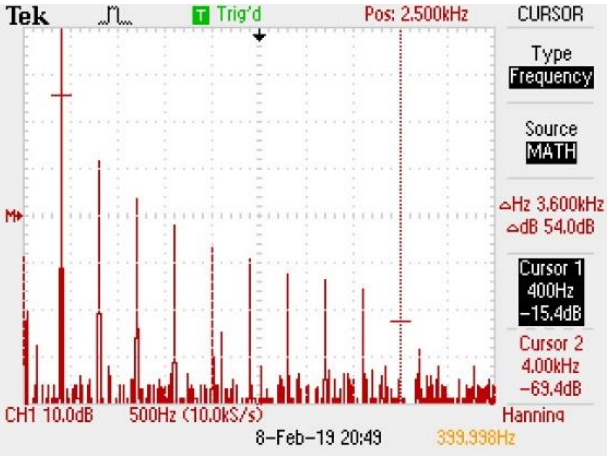


Figure 8: 200Hz

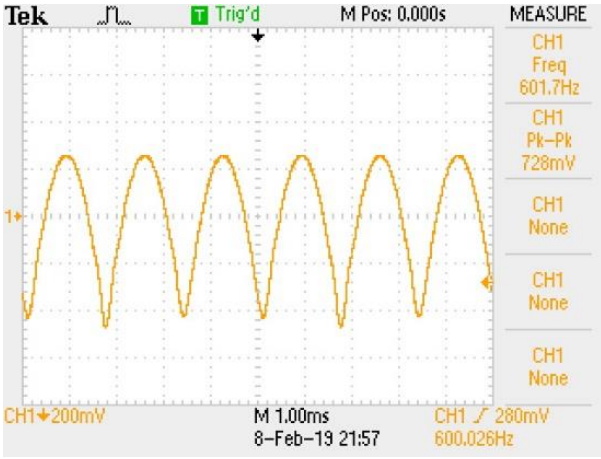


Figure 9: 300Hz

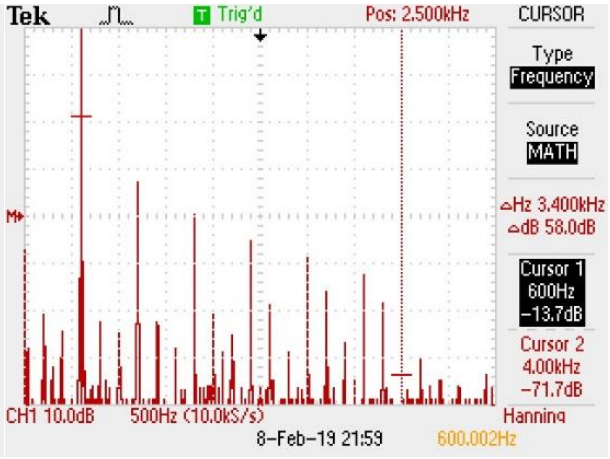


Figure 10: 300Hz

Figure 7 and Figure 8 show the waveform and FFT of 200Hz input frequency. In the FFT graph there are 10 visible peaks. The 11th harmonic, which has frequency 4400Hz, theoretically, should contribute a peak at 4400Hz in FFT if the output wave is a continuous analog signal. Since the DSK board is a digital device and we manually set the sampling frequency to be 8kHz, due to the nature of sampling, the 4400Hz frequency is beyond the Nyquist frequency and is folded or reflected back to 3600Hz. As a result, the 11th harmonic (4400Hz) is aliased and mixed up with the 9th harmonic (3600Hz), the 12th harmonic (4800Hz) is also aliased and mixed up with the 8th harmonic (3200) and so on. This causes that there are only 10 visible peaks. The waveform looks fine because the aliasing harmonics coincide with the lower harmonics. This means the frequency component above the Nyquist frequency is cut out but those within the Nyquist frequency is well retained, and losing higher order harmonics has infinitesimal effect on the waveform.

In the contrary, if the input frequency is 300Hz, the output wave form is slightly distorted (Figure 9) and the FFT graph is more disordered. The harmonics up to 6th are retained but the 7th harmonic (7x600Hz=4200Hz) is now folded back to 3800Hz which does not coincide with any other harmonics. This happens to 8th, 9th etc. harmonics too. All these mixing up causes the slight distortion of the output waveform. However, the output is still acceptable, and the details of the waveform are preserved well enough.

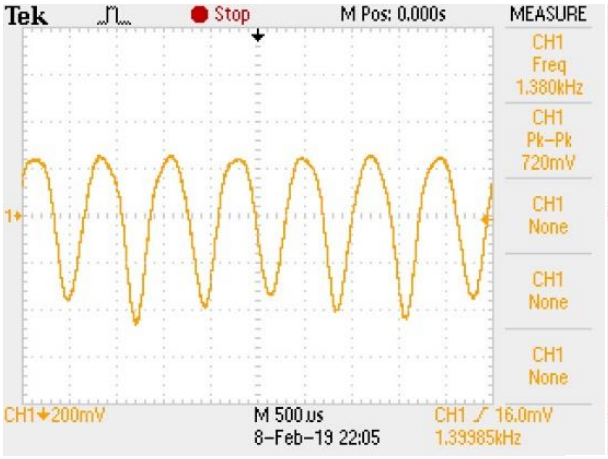


Figure 11: 700Hz

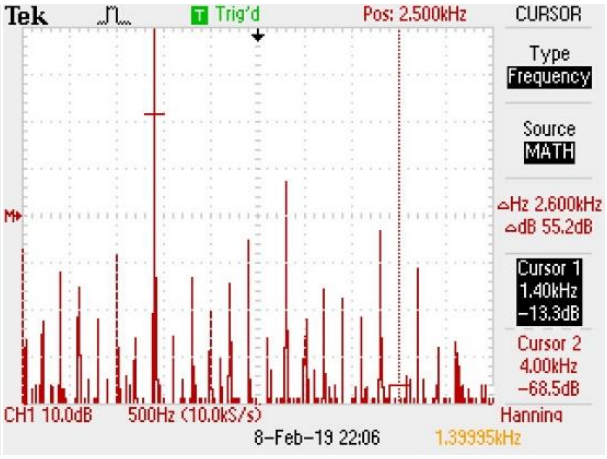


Figure 12: 700Hz

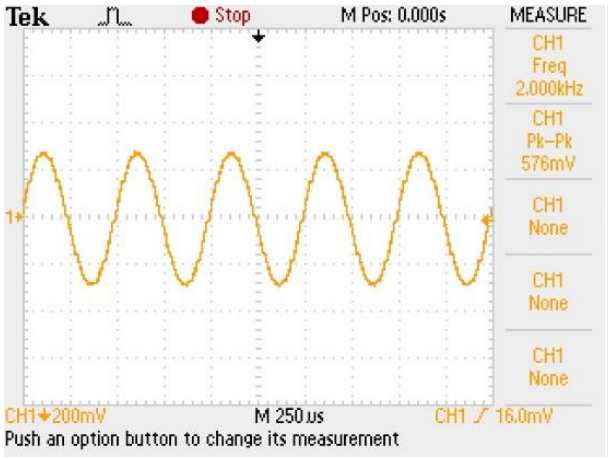


Figure 13: 1kHz

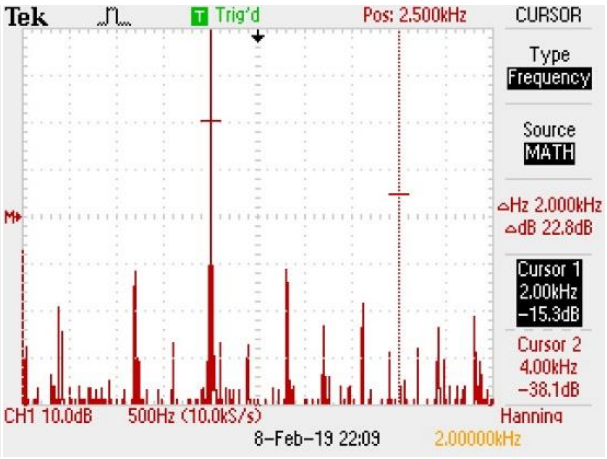


Figure 14: 1kHz

When the input frequency increases to 700Hz, the distortion of the output waveform can be clearly seen in Figure 11. And in Figure 12, the FFT shows there are only 2 correct peaks within the Nyquist frequency, which are at 1.4kHz (fundamental) and 2.8kHz (2^{nd} harmonic). Other peaks the aliased frequencies beyond the Nyquist frequency shifted to the range 0Hz ~ 4kHz. The details of the rectified sine wave are no longer preserved.

Figure 13 and Figure 14 show what happens if the input signal is 1kHz. The output wave is just a pure sine wave with frequency 2kHz which is the fundamental frequency. But the 2^{nd} harmonic is 4kHz which is exactly the Nyquist frequency which is unpredictable. The 3^{rd} harmonic folded back and added up to the 2kHz frequency. The 4^{th} harmonic folded back to 0Hz which is again unpredictable. But we could see some very slow variations of the waveform on the oscilloscope due to systematic errors, e.g. inaccuracy of the signal generator.

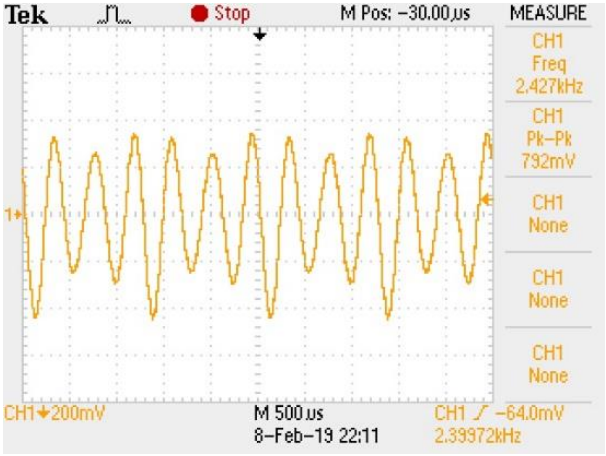


Figure 15: 1.2kHz

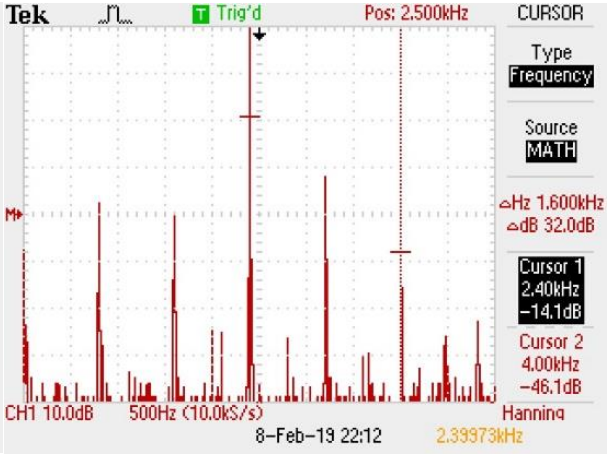


Figure 16: 1.2kHz

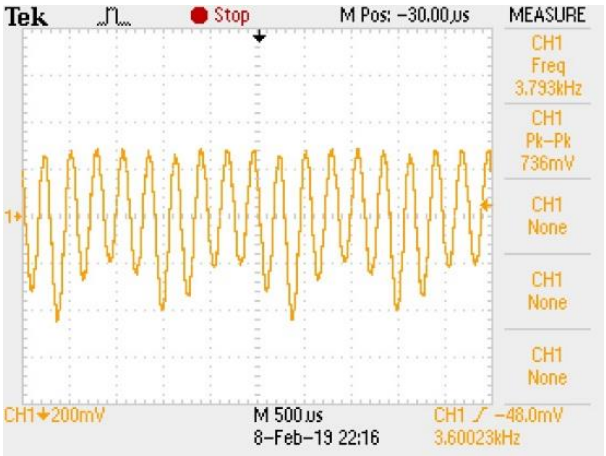


Figure 17: 1.8kHz

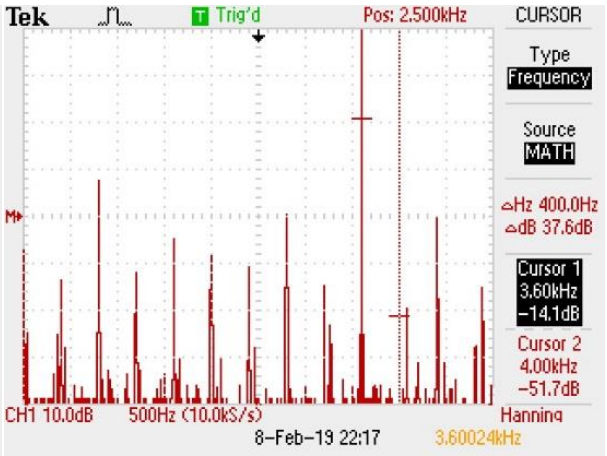


Figure 18: 1.8kHz

As the frequency continue to increase, we observed that there is only fundamental frequency (twice the input frequency) within the Nyquist frequency when the input signal has the frequency range between 1kHz and 2kHz. All other frequency components are the reflected frequencies. They add up and highly distort the fundamental frequency.

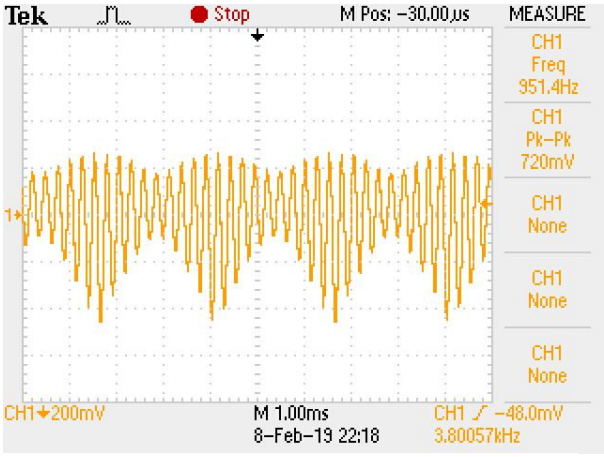


Figure 19: 1.9kHz

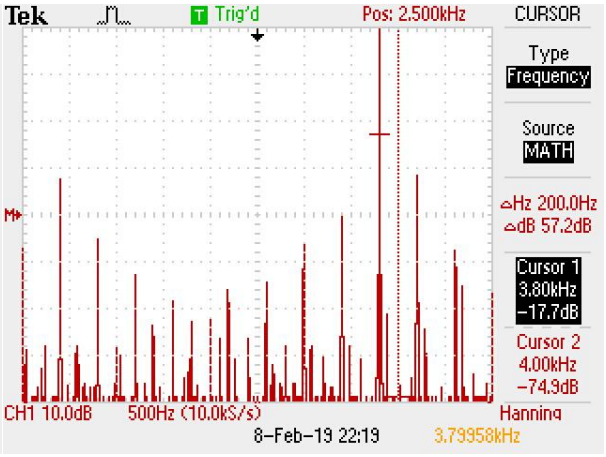


Figure 20: 1.9kHz

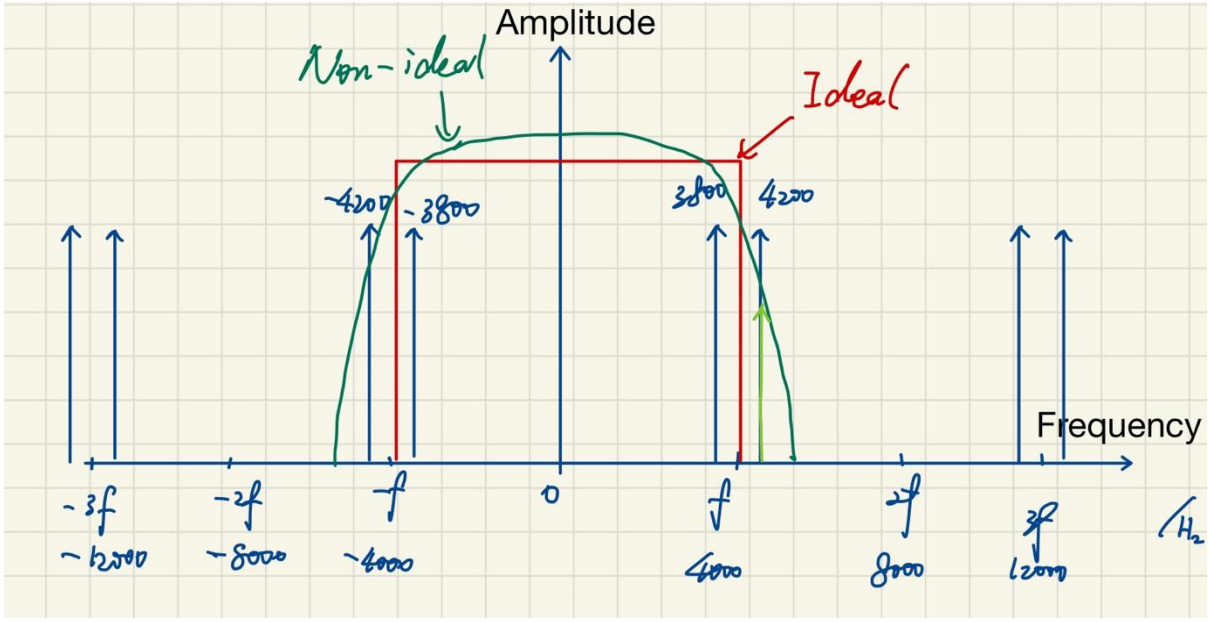


Figure 21

If the input signal is 1.9kHz, the output fundamental frequency would be 3.8kHz (Figure 20). When the fundamental frequency is close to the Nyquist frequency, there is a clear beating pattern in the waveform (Figure 19). This is due to non-ideal reconstruction filters in the DAC of the DSK board. We can plot the situation in frequency domain (Figure 21). An ideal lowpass regeneration filter should have a rectangular shape in the frequency domain and an infinitely long sinc function in time domain. A non-ideal lowpass filter does not have a perfect rectangular shape in frequency domain. There is always a finite slope between the cutoff frequency and the stopband frequency.

There is a 4.2kHz frequency component because of sampling. The non-ideal filter could only filter out part of the 4.2kHz frequency, and the left part of it mixed up with the desired 3.8kHz frequency. When two frequencies are close together, they would generate a beating pattern or an amplitude envelope pattern. Along with other harmonics folding back to 0 ~ 4kHz, the output would become quite chaotic.

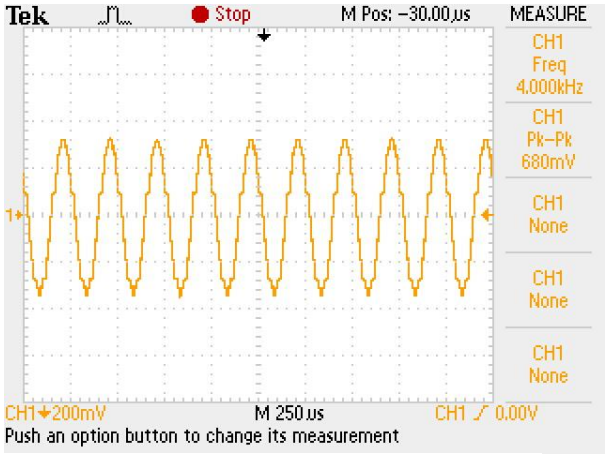


Figure 21: 2kHz

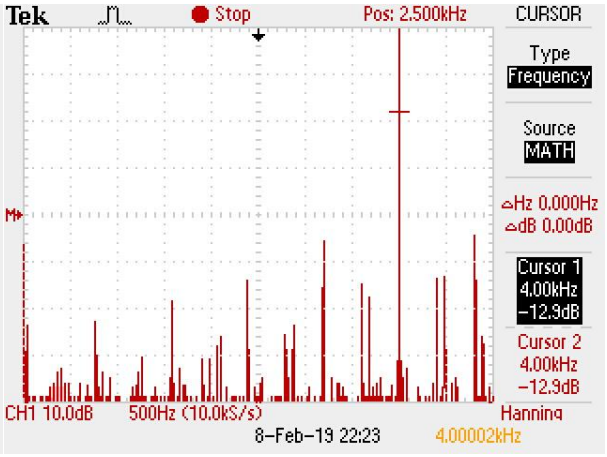


Figure 22: 2kHz

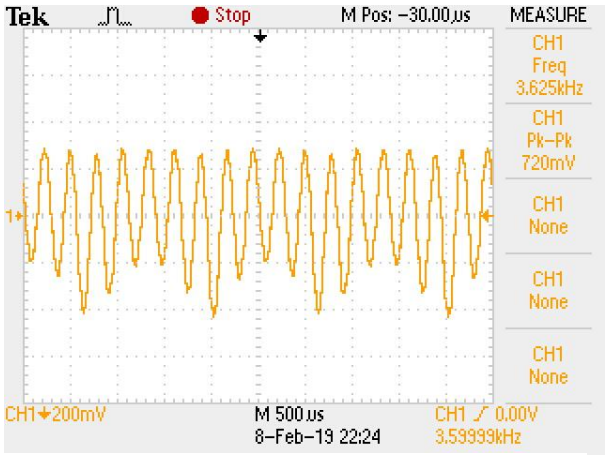


Figure 23: 2.2kHz

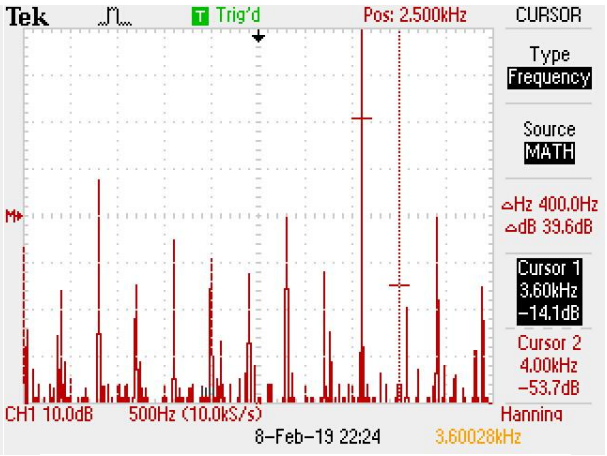


Figure 24: 2.2kHz

If the input signal frequency happens to be 2kHz (Figure 21 and Figure 22), which means the fundamental frequency of the output wave is 4kHz. This happens to be exactly the Nyquist frequency. The situation becomes unpredictable as well. Theoretically, if the signal frequency is exactly the Nyquist frequency, the regenerated waveform would be either zeros or a pure sine wave but with unknow amplitude. The amplitude would be entirely determined by the first sample. If the first sample taken is zero, the follow-up samples would be all zeros. If the first sample is taken somewhere between maxima or minima, the output would be a pure sine wave with amplitude equal to the value of the first sample.

However, we observed on the oscilloscope and found that there is a pure sine wave shown on the oscilloscope, but its amplitude is continuously increasing and decreasing with a very long period. This is again due to systematic errors. The signal generators we used on PC may generate a sine wave at frequency not exactly 2kHz.

Further increasing the input frequency, the fundamental frequency of the output wave is beyond the Nyquist frequency and is folded back. Its harmonics did as well. We discovered that the output waveform of the input frequency from 0Hz to 2kHz is the same as that of the input frequency from 4kHz to 2kHz.

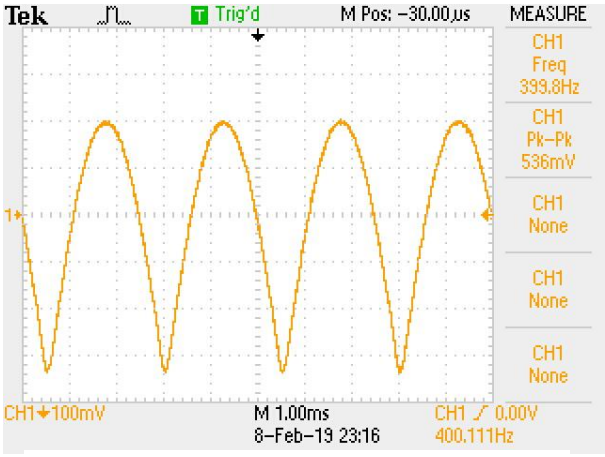


Figure 25: 3.8kHz

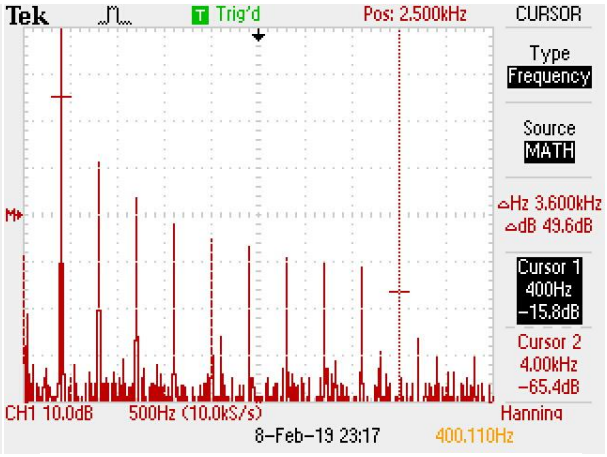


Figure 26: 3.8kHz

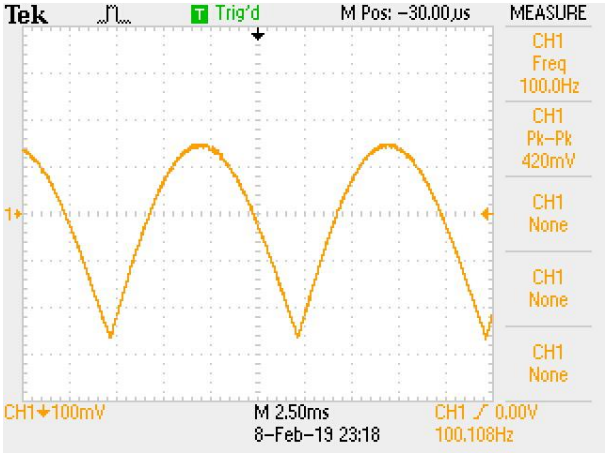


Figure 27: 3.95kHz

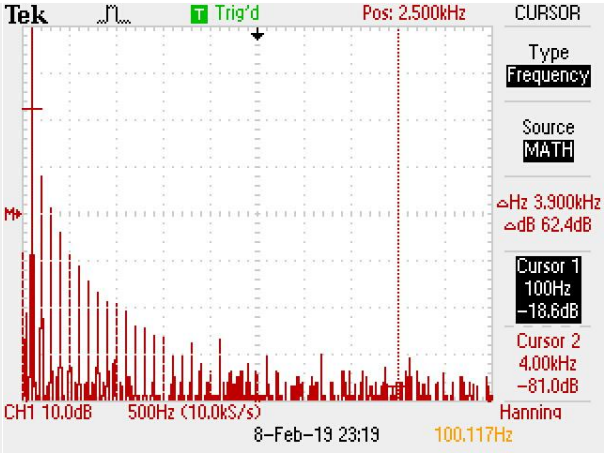


Figure 28: 3.95kHz

Figure 25 and Figure 26 show the waveform and FFT of the output signal if the input signal is 3.8kHz. Both the output waveform and FFT are indistinguishable from the output waveform and FFT of input frequency 200Hz (Figure 7 and Figure 8). The fundamental frequency of output is aliased to 400Hz, 2nd harmonic is aliased to 800Hz, 3rd to 1200Hz, 4th to 1600Hz and so on. All the aliased frequencies form the exact pattern as the 200Hz input signal does.

Then the output waveform of 3.95kHz input (Figure 27 and Figure 28) is the same as that of 50Hz, which confirms our discussion.

In conclusion, the output waveform looks fine when the input sine wave has frequencies less than 700Hz and it is distorted above 700Hz under 2kHz. When the input frequency is from 2kHz to 4kHz, the output wave becomes the same as when the input frequency is between 2kHz to 0Hz.

Explanation of Code

First exercise:

The function “ISR_AIC” (Full version of the code is in the appendix):

```
void ISR_AIC(void)
{
	int samp = 0;
	samp = abs(mono_read_16Bit());
	mono_write_16Bit(samp);
}
```

Firstly, declaring a local variable “samp”, and assign “samp” to be the absolute value of the function “mono_read_16Bit” which is stated in the lab instruction: “It reads a Left and Right sample from the audio port, divides the amplitude of them by two and then sums the samples together to provide a mono input.”

After the calculation, the “samp” is passed to the function “mono_write_16Bit”. As the lab instruction said: “This function simply takes the argument samp and sends copies of it to the left and right channels to achieve a mono output.”

Second exercise:

Overview of the function “ISR_AIC” (Full version of the code is available in the appendix):

```
void ISR_AIC(void)
{
	// temporary variable used to output values from function
	float sampling_period=0;
	float result=0;
	int remainder = 0;
	Int16 result_16=0;
	int calcu=0;

	sampling_period = SINE_TABLE_SIZE*sine_freq/sampling_freq;
	calcu = round(sampling_period*counter);
	remainder = calcu%SINE_TABLE_SIZE;
	result = table[remainder];

	result_16 = round(result * 32767);
	mono_write_16Bit(abs(result_16));

	if(remainder == 0 ){
		counter = 0;
	}
	counter = counter+1;

	// Set the sampling frequency. This function updates the frequency only if it
	// has changed. Frequency set must be one of the supported sampling freq.
	set_samp_freq(&sampling_freq, Config, &H_Codec);
}
```

Detailed explanation:

```
float sampling_period=0;
float result=0;
int remainder = 0;
Int16 result_16=0;
int calcu=0;
```

Declaring local variables. Initially, the “sampling_period”, “calcu”, “result”, “remainder” and the “result_16” are set to zero. The “result_16” is forced to be a 16bits integer because later in the function “mono_write_16Bit”, the input argument of this function must be a 16 bits signed number.

```
sampling_period = SINE_TABLE_SIZE*sine_freq/sampling_freq;
calcu = round(sampling_period*counter);
remainder = calcu%SINE_TABLE_SIZE;
result = table[remainder];
```

The variable “sampling_period” indicates how many values in the sine table we have to skip to get the next value. This is calculated by the “SINE_TABLE_SIZE” times the ratio of “sine_freq” to “sampling_freq”. For example, if the sine frequency is 1kHz and the sampling frequency is 8kHz, the ratio of them is 1/8. This means there are 8 samples in every cycle of a sine wave. If we know the first sample should take the value of index zero in the lookup table, the next sample would be the value of index 32 because 256/8=32 where 256 is the table size. And the following samples would be the index 64, 96, 128 etc.

The variable “calcu” basically calculates the index of the current samples in the lookup table. Rounding is necessary because the index of the lookup table must be an integer. However, the variable “calcu” may go beyond the size of the lookup table resulting errors. Instead of using a “while loop” or an “if statement”, this could be easily avoided by finding the remainder of “calcu” divided by the size of the lookup table. Here, the “%” is a modulo operator which would return the remainder of two numbers. Because if the index is 256, the corresponding table value would be the same as the table value of index 0. The table values repeat every 256 consecutive values.

The “result” would be a floating-point number between -1 and 1.

```
result_16 = round(result * 32000);
mono_write_16Bit(abs(result_16));
```

The input argument of the function “moni_write_16Bit” could only be a 16bits integer. So there are two steps here.

1. We need to multiply “result” by an appropriate number. The 16bits integer is actually a 16bits signed number. The minimum of a 16bits signed number is -2^{15} which is -32768, and the maximum is 2^{15}-1 which is 32767. Thus, if a number whose absolute value is larger than 32767 is chosen, the output signal may contain an overflow. Therefore, 32767 is an appropriate number here.

2. The "result" would not be an integer in most cases even multiplied by 32000. So another rounding is required.

Then, to rectify the signal, we take the absolute value of the calculated "result_16". Every negative number is taken its opposite value.

```
if(remainder == 0 ){
        counter = 0;
}
```

However, in the lab experiment, we found that if the "counter" we used exceeds certain number, the output waveform would become quite chaotic and highly distorted. This may because of the modulo operator may take the CPU more time to calculate the remainder.

However, forcing the "counter" to zero may cause some abrupt changes in output wave periodically. By forcing the "counter" to 0 if the "remainder" calculated before is equal to 0, there would be no transients or abrupt changes of the output wave. Because the "remainder" is actually the index of the table. If "remainder" is zero, "calcu" is divisible by the table size which is 256. And "calcu" is basically the "counter" times the "sampling_period". The "sampling_period" does not change in each iteration. So setting the "counter" to zero if "remainder" is zero works perfectly.

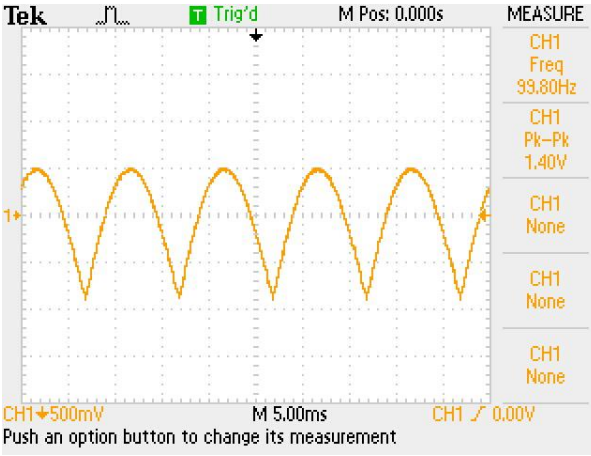
```
counter = counter+1;
```

The "counter" is then incremented by one in order to calculate the next sample.

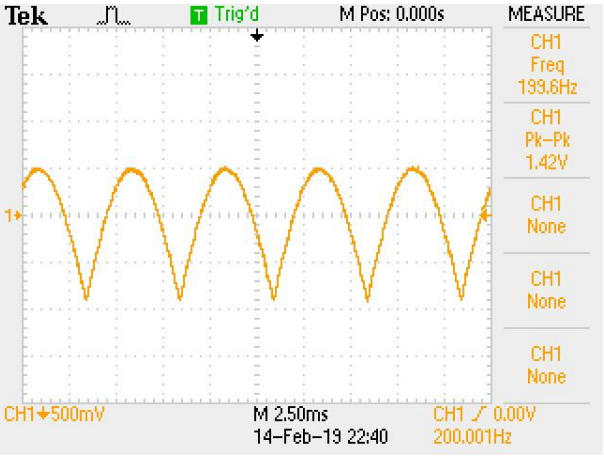
```
set_samp_freq(&sampling_freq, Config, &H_Codec);
```

Lastly, we added this line of code, in case we want to change the sampling frequency when the DSK board is running.

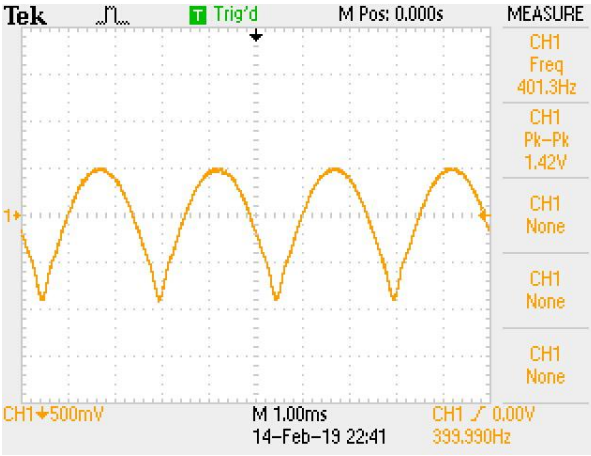
Scope traces in exercise 2:



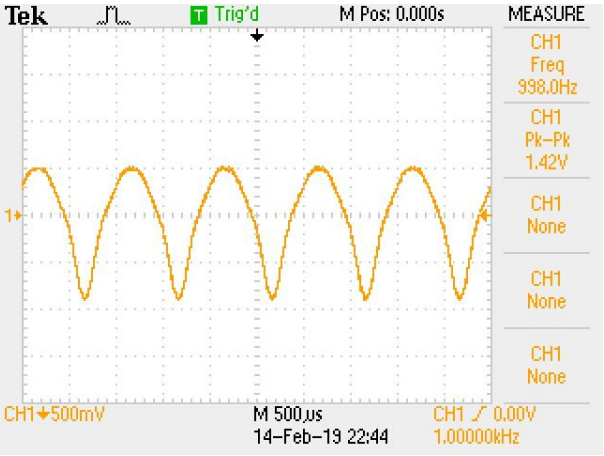
Input: 50Hz



Input: 100Hz

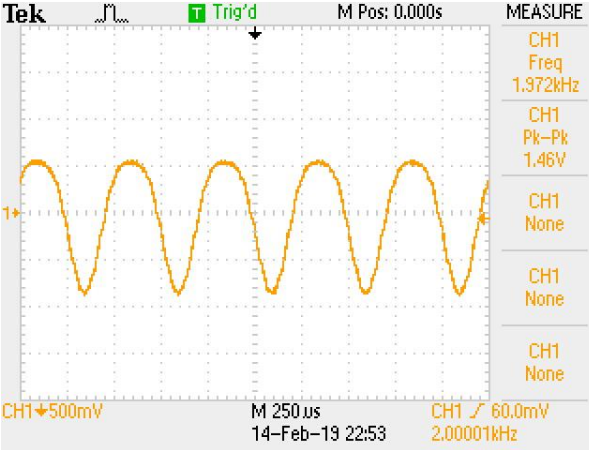


Input: 200Hz

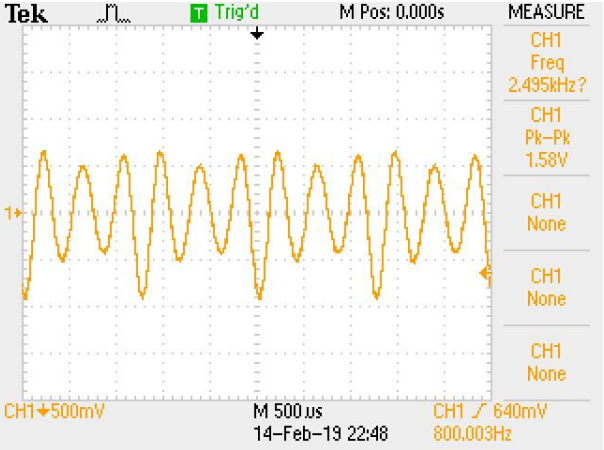


Input: 500Hz

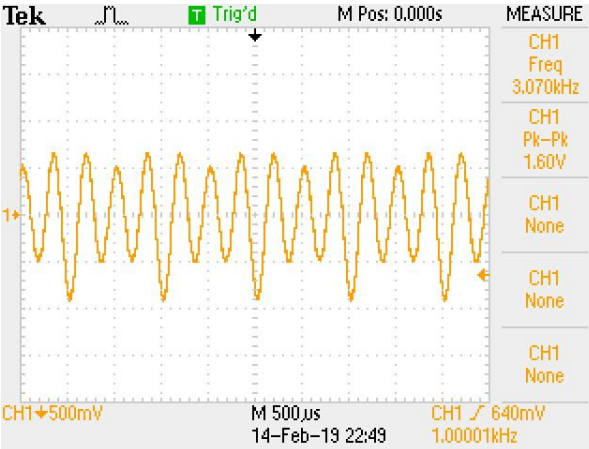
Scope traces in exercise 2:



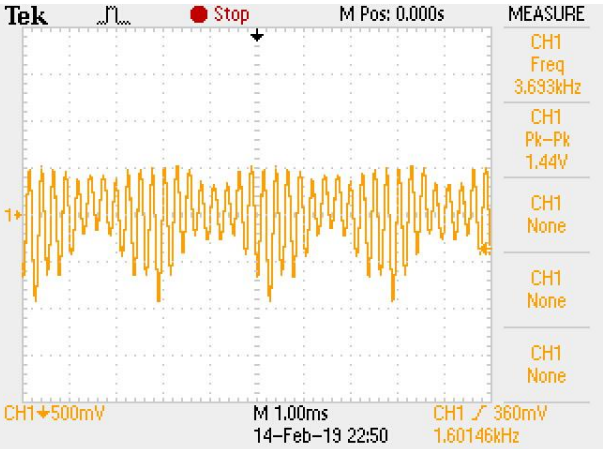
Input: 1kHz



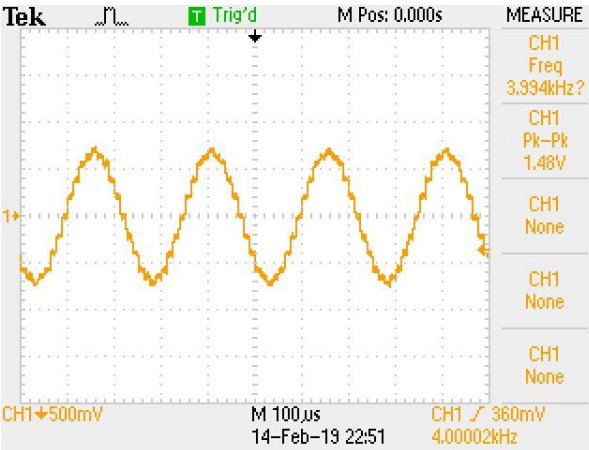
Input: 1.2kHz



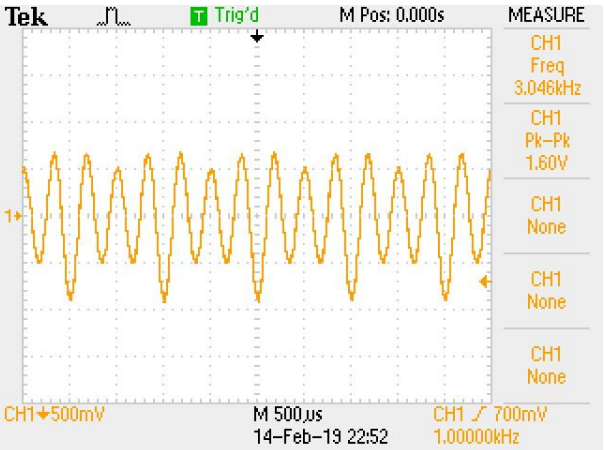
Input: 1.5kHz



Input: 1.9kHz

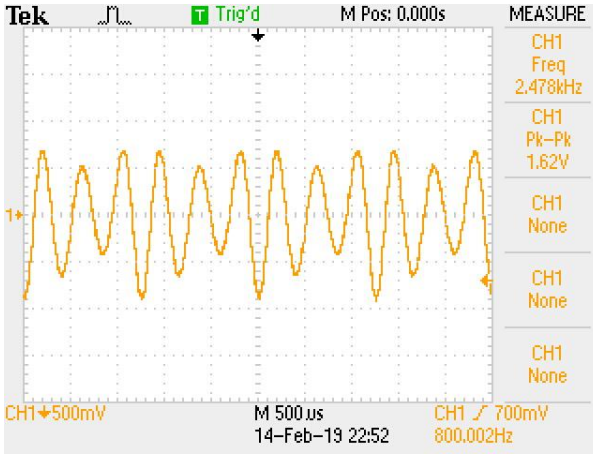


Input: 2kHz

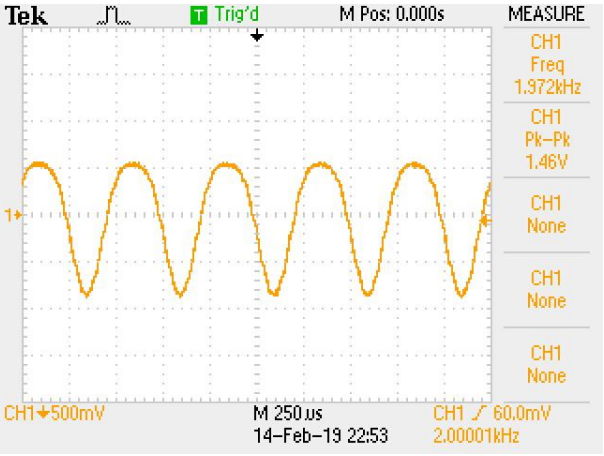


Input: 2.5kHz

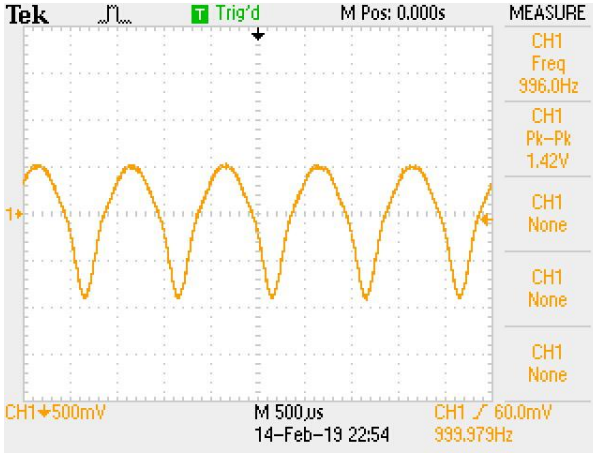
Scope traces in exercise 2:



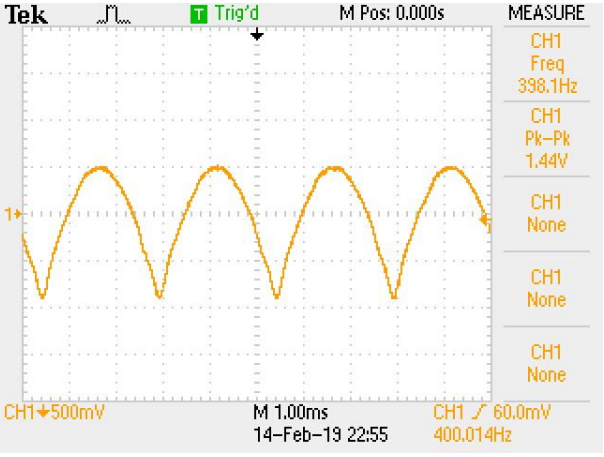
Input: 2.8kHz



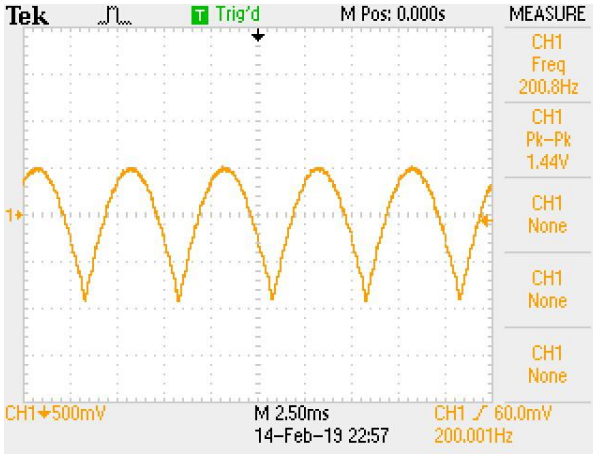
Input: 3kHz



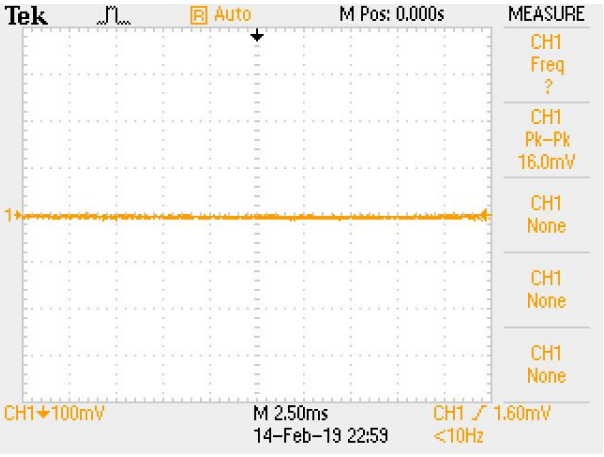
Input: 3.5kHz



Input: 3.8kHz

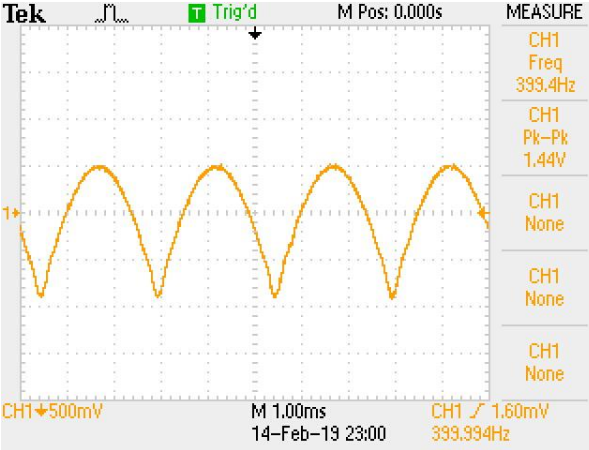


Input: 3.9kHz

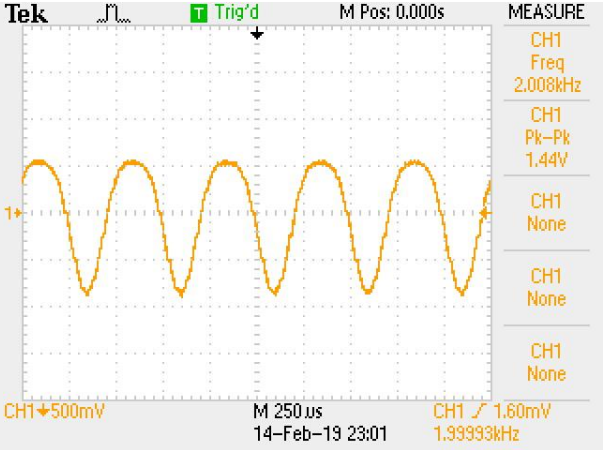


Input: 4kHz

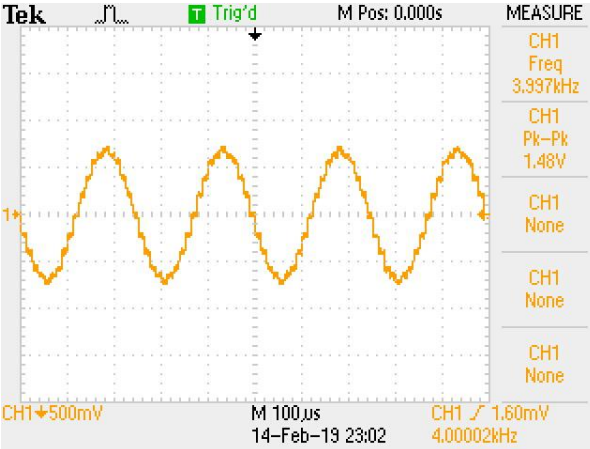
Scope traces in exercise 2:



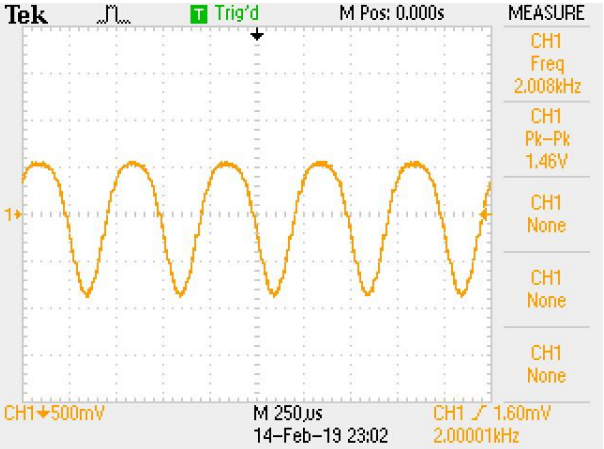
Input: 4.2kHz



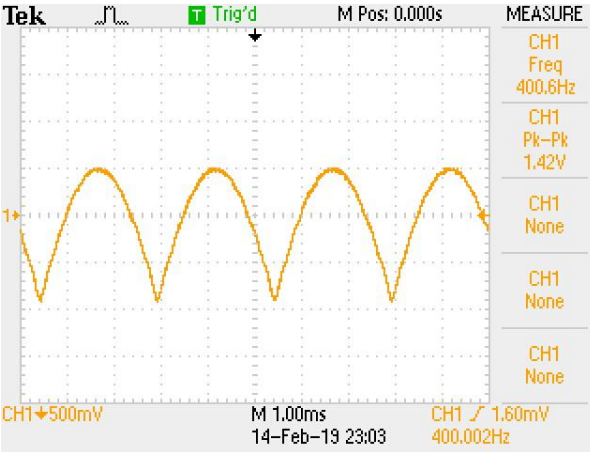
Input: 5kHz



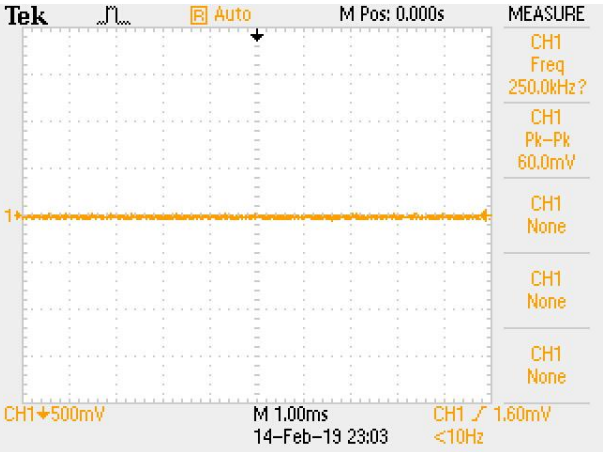
Input: 6kHz



Input: 7kHz



Input: 7.8kHz



Input: 8kHz

The scope traces proved that our code works as expected. And edge frequencies are shown as well.

Appendix (Full version of code)

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

/*************************** Pre-processor statements ****************************/

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

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

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

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

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

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

/*****************************************************************/
/*   REGISTER              FUNCTION                      SETTINGS         */
/*****************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                   */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                   */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                   */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                   */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off       */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on       */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit                */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ                 */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                    */\

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

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

```
 /****************************** Function prototypes ******************************/
void init_hardware(void);
void init_HWI(void);
void 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(void)
{
       int samp = 0;
       samp = abs(mono_read_16Bit());
       mono_write_16Bit(samp);
}
```

Exercise 2:

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

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

                                                        LAB 3: Interrupt I/O

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

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

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

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

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

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

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

/******************************* 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);

/*******************************add on*********************************************/

// Some functions to help with configuring hardware
#include "helper_functions_polling.h"
#define PI 3.141592653589793
#define SINE_TABLE_SIZE 256
float table [SINE_TABLE_SIZE];
int counter =0;
int sampling_freq = 8000;
float sine_freq = 1000.0;

void sine_init();
/*********************************************************************************/


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

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

       /* initialize hardware interrupts */
       init_HWI();

       //intialize the lookup table and global variable
       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);

       /* Set the sampling frequency of the audio port. Must only be set to a supported
       frequency (8000/16000/24000/32000/44100/48000/96000) */
       DSK6713_AIC23_setFreq(H_Codec, get_sampling_handle(&sampling_freq));
```

```
	/* 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

}

void sine_init()
{
	int n=0;
	//define local variable
	for(n=0;n<SINE_TABLE_SIZE;n++){
		table[n] = sin(2*PI*n/SINE_TABLE_SIZE);
	}
}

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

void ISR_AIC(void)
{
	// temporary variable used to output values from function
	float sampling_period=0;
	float result=0;
	int remainder = 0;
	Int16 result_16=0;
	int calcu=0;

	sampling_period = SINE_TABLE_SIZE*sine_freq/sampling_freq;
	calcu = round(sampling_period*counter);
	remainder = calcu%SINE_TABLE_SIZE;
	result = table[remainder];

	result_16 = round(result * 32767);
	mono_write_16Bit(abs(result_16));

	if(remainder == 0 ){
		counter = 0;
	}
	counter = counter+1;

	// Set the sampling frequency. This function updates the frequency only if it
	// has changed. Frequency set must be one of the supported sampling freq.
	set_samp_freq(&sampling_freq, Config, &H_Codec);
}
```