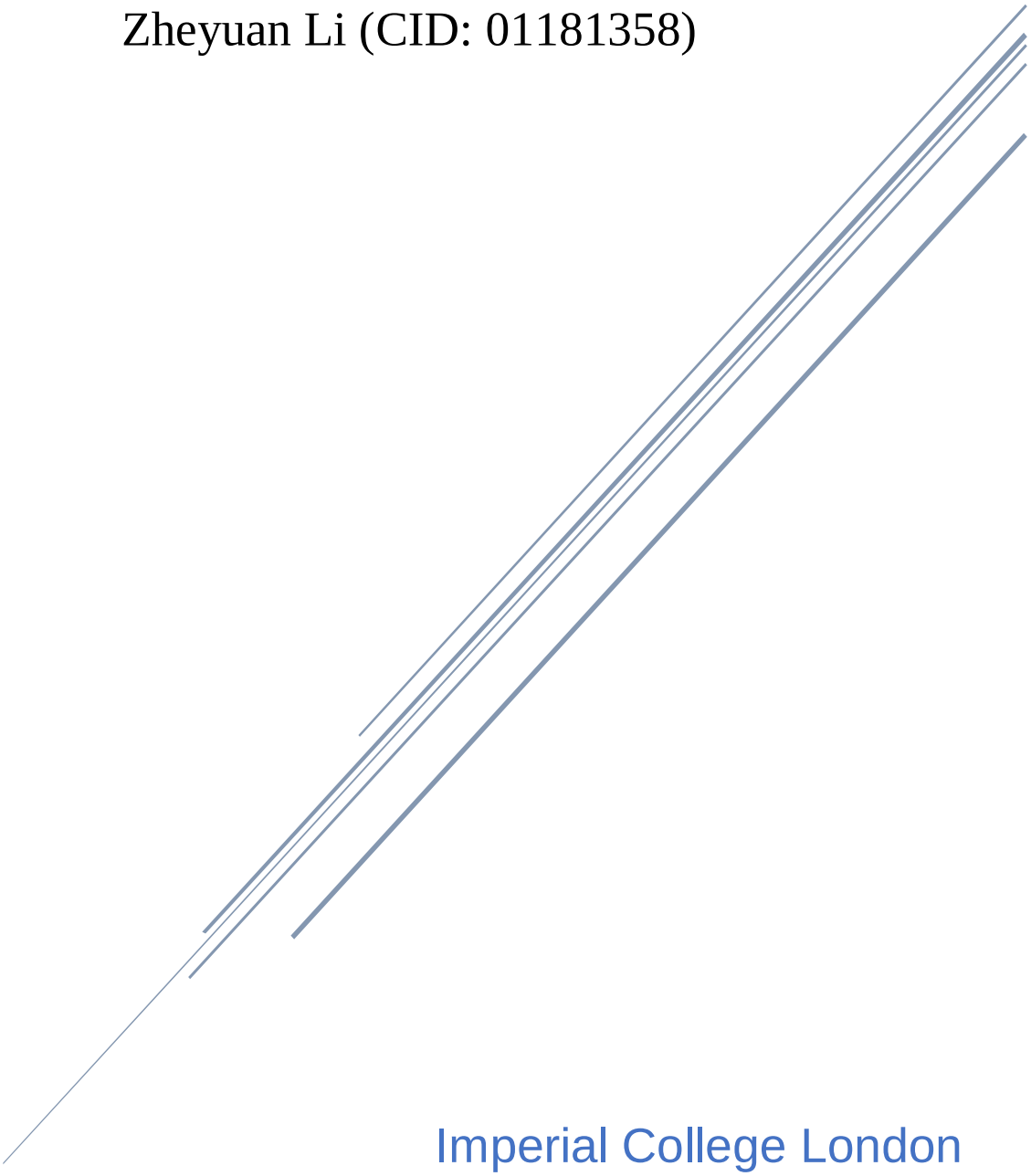
RTDSP Lab-4 Report

Calvin Chan (CID: 01048905)

Zheyuan Li (CID: 01181358)



Imperial College London

EE3-19 Real-time Digital Signal Processing (2018-2019)

Content

1. Design of the FIR filter in Matlab

In this exercise, we are going to output the frequency response of required filter by Matlab.

There are two functions which are recommended in the lab handout, which are firpm() and firpmord(). b = firpm(n+2,fo,ao,w) will give us a n+3 coefficients of order n+2 filter, and firpmord() will estimate the filter order.

The first part of the code below has presented understandable and clear comment, so no more explanation is required, and it is just simply specified the specification in lab handout.

Specification by picture has shown in beow:

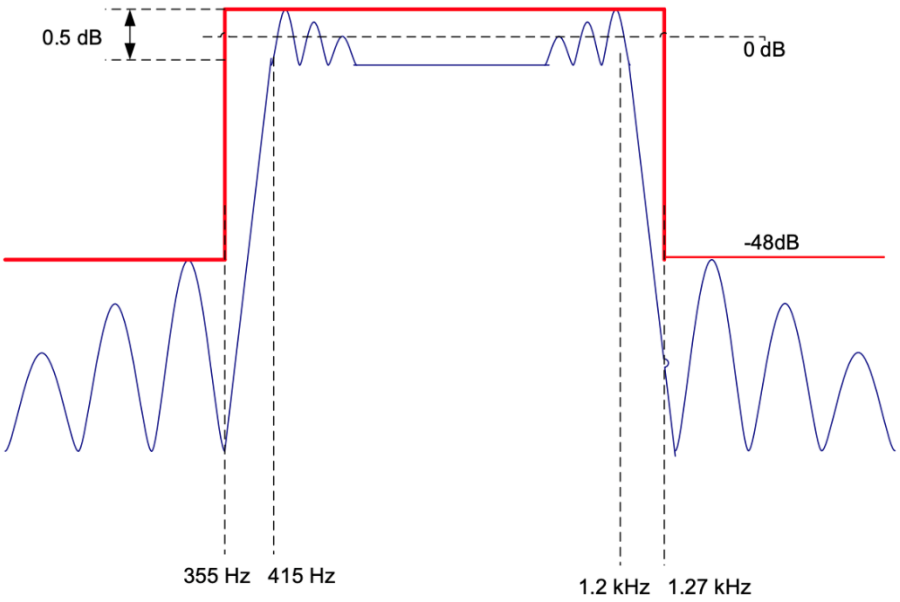


Figure 1[1]

[1] *Lab 4 – Real-time Implementation of FIR Filters. (2019). [ebook] Paul D. Mitcheson. Available at: https://bb.imperial.ac.uk/bbcswebdav/pid-1328511-dt-content-rid-4367887_1/courses/DSS-EE3_19-18_19/DSS-EE3_19-18_19_ImportedContent_20180503122456/lab4.pdf [Accessed 23 Feb. 2019].*

Writing in Matlab:

```
rp = 0.5;                       % Passband ripple (in dB)
rs = 48;                        % Stopband attenuation (in dB)
fs = 8000;                      % Sampling frequency
f = [355 415 1200 1270];         % Cut-off frequencies
a = [0 1 0];                    % Desired amplitudes

dev_pass = (10^(rp/20)-1)/(10^(rp/20)+1);    % absolute passband ripple
dev_stop = 10^(-rs/20);         % absolute stopband attenuation
dev = [dev_stop dev_pass dev_stop];

[n,fo,ao,w] = firpmord(f,a,dev,fs);
b = firpm(n+2,fo,ao,w);
% increase the order n by 2 to meet the specifications

freqz(b,1,4096,fs);
% the 4096 is the number of samples taken to plot the graphs
```

In the final part, freqz(b,1,4096,fs) is trying to output the frequency response so as to verified whether we have met the design specification. The graphs have been put in Figure 1, it shows the frequency response of the FIR filter in Matlab (actual frequency response from the APx500 audio analyzer will be presented later):

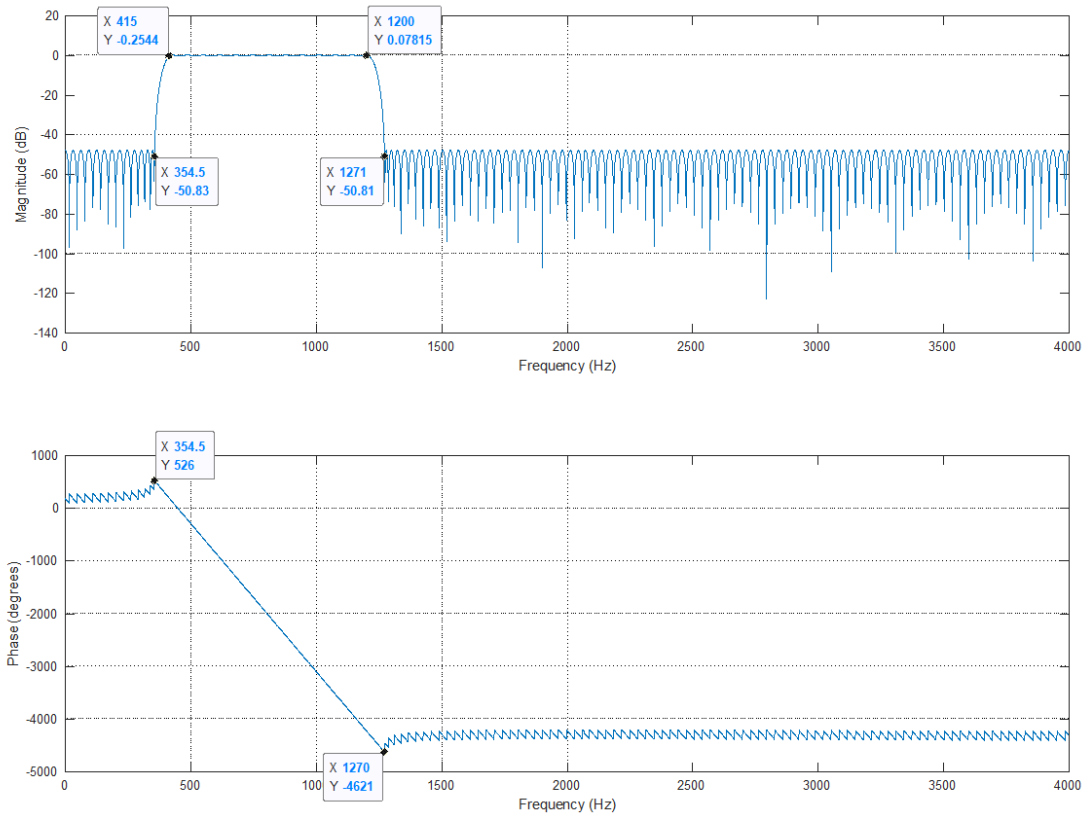


Figure 2: Frequency response of the FIR filter in Matlab

2. Non-circular FIR filter in C

In the following exercise, we are going to test the speed of non-circular buffer FIR filter algorithm. Before doing this, the way of testing needs to mention, which that we put two break points in the beginning and the end of the filter function.

2.1. Optimizations Levels

Before writing and explaining the code, an explanation of optimization level in tool code composer studio is needed since we are going to use later. There are four levels of optimization, which are -o0, -o1, -o2 and -o3. By checking the "TMS320C6000 Optimizing Compiler User's Guide", a brief description for three levels has shown in below.

1. *-o0*
   - *Performs control-flow-graph simplification*
   - *Allocates variables to registers*
   - *Performs loop rotation Eliminates unused code*
   - *Simplifies expressions and statements*
   - *Expands calls to functions declared inline*
2. *-o1*
   - *Performs all −O0 optimizations*
   - *Performs local copy/constant propagation*
   - *Removes unused assignments*
   - *Eliminates local common expressions.*
3. *-o2*
   - *Performs all −O1 optimizations*
   - *Performs software pipelining*
   - *Performs loop optimizations*
   - *Eliminates global common subexpressions*
   - *Eliminates global unused assignments*
   - *Converts array references in loops to incremented pointer form*
   - *Performs loop unrolling*[2]

In this lab, we are going to use -o0 and -o2 but not -o3, and we expect level -o2 contributes the fewest cycles of the function, because the o2 has optimized the most theoretically.

[2] *TMS320C6000 Optimizing Compiler User's Guide. (2004). [ebook] Texas Instruments. Available at:* http://www.cs.cmu.edu/afs/cs/academic/class/15745-s05/www/c6xref/compiler_guide.pdf *[Accessed 23 Feb. 2019].*

2.2. Processing File of Coefficient

The coefficients of the FIR filter are in the variable "b" in the Matlab workspace. Before using the coefficients, we need to write them into a txt file.

```
save fir_coefs_tabs.txt b -ascii -double -tabs
```

By running this command in Matlab, the variable "b" is then written into a text file called "fir_coefs.txt" in double floating-point precision. The format of an array that is understandable by C is like "double b[] = {1, 2, 3, 4, …}" in which numbers are delimited by comma. However, the numbers in the "fir_coefs_tabs.txt" is tab-delimited rather than comma-delimited. By simply running a C++ script (in the appendix), there is a new txt file named "fir_coefs.txt" generated and will be used later.

2.3. Explanation of Code

Full version of code is provided in the appendix.

Somewhere in beginning of the code in C:

```
#include "fir_coefs.txt"

#define N 250
double x[N] = {0};
```

Including the text file "fir_coefs.txt" enables us to use the filter coefficients generated in Matlab after format conversion. There are totally 251 coefficients. So we need an array of size 251. It includes the first coefficient which would be multiplied by the current sample. And initializing all elements in the array to be zero.

Code in the ISR:

```
void ISR_AIC(void)
{
    Int16 samp_out = 0;

    samp_out = round(non_circ_FIR());
    mono_write_16Bit(samp_out);
}
```

Declaring local variables. "samp_out" is the output sample which is the input parameter of the function "mono_write_16Bit". The input of this function can only be a 16bits signed integer. So the variable "samp_out" is assigned as a 16bits integer. However the return value of the function "non_circ_FIR()" will be a double type. Rounding here to retain accuracy. Then the "samp_out" is being written to the output.

Function of non-circular FIR filter:

```
double non_circ_FIR(void)
{   //initialize output
    double sum = 0;
    int i = 0; //counter

    for(i=N;i>0;i--){
        x[i] = x[i-1];
    }
    x[0] = mono_read_16Bit();  //read the current sample

    for(i=0;i<=N;i++){
        sum += b[i] * x[i];
    }

    return sum;
}
```

Declaring the "sum" to be a double type to avoid inaccuracy building up in the multiply-accumulate procedure. The int "i" is basically an index of the sample array and the coefficient array.

The objective of this for loop is to shift all the samples one space higher. In each iteration, it copies the $(i-1)^{th}$ delayed sample to the i^{th} delayed sample. The process is illustrated in Figure 3. Then we assign x[0] to be the new sample read from H-Codec.

The multiply-accumulate (MAC) operation is performed in the for loop (see Figure 4). And the "sum" is returned, being the output sample.

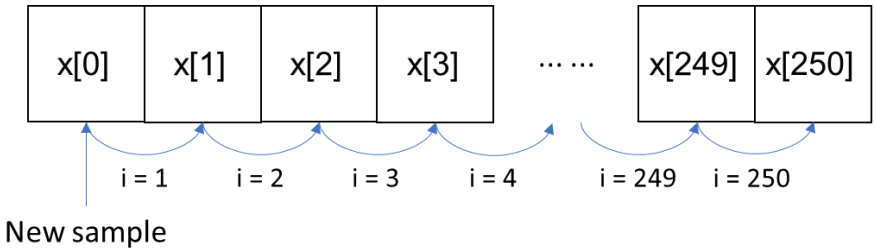


Figure 3

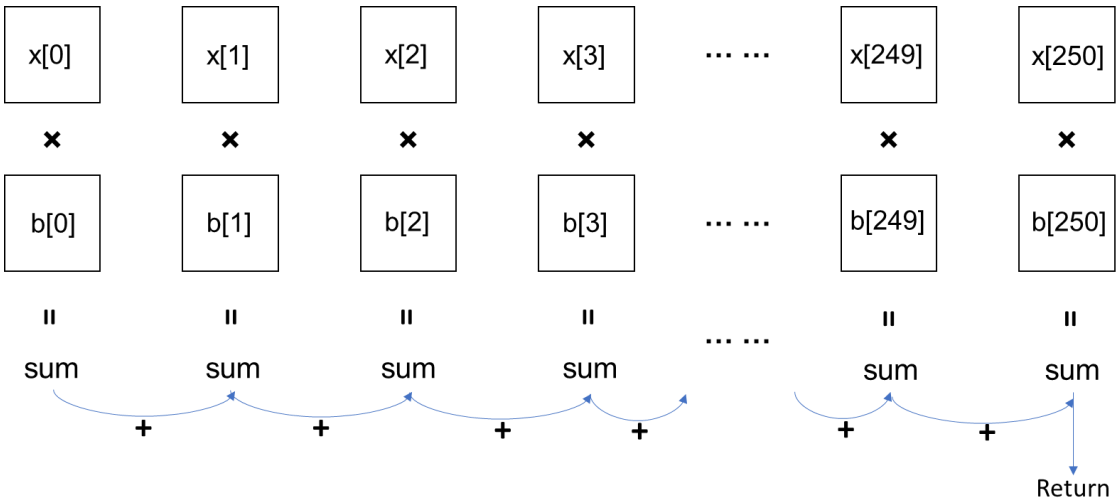


Figure 4

2.4. Benchmark

Figure 5 shows the number of cycles we measured for the DSK board to perform each interrupt service routine function "ISR_AIC".

Non-circular Filter	
Optimisation Level	**Cycles**
None	16170
-o0	13179
-o2	3261

Figure 5: Benchmark of non-circular FIR filter

From the table above, we conclude that if there is no optimization, the DSK board would take 16170 cycles to calculate each output sample. Increasing the optimization level would rapidly reduce the number of cycles. Using "-o2" level of optimization enables the DSK board run roughly 5 times faster than using no optimization.

3. Circular buffer FIR filter in C

3.1. Methodology

In the non-circular FIR filter, we used the shifting samples method which is highly inefficient. For example, it copies N samples each time, where N is the order of the filter. It would be very slow if N gets very large, i.e. very high order FIR filter.

Instead of copying data around, we can just introduce a pointer which points to the most current sample in the array and simply move the pointer around. This improvement much reduces the computational complexity when we need to update the buffer array after receiving a new sample. However, the MAC procedure stays the same.

The circular buffer algorithm could be visualized in Figure 6:

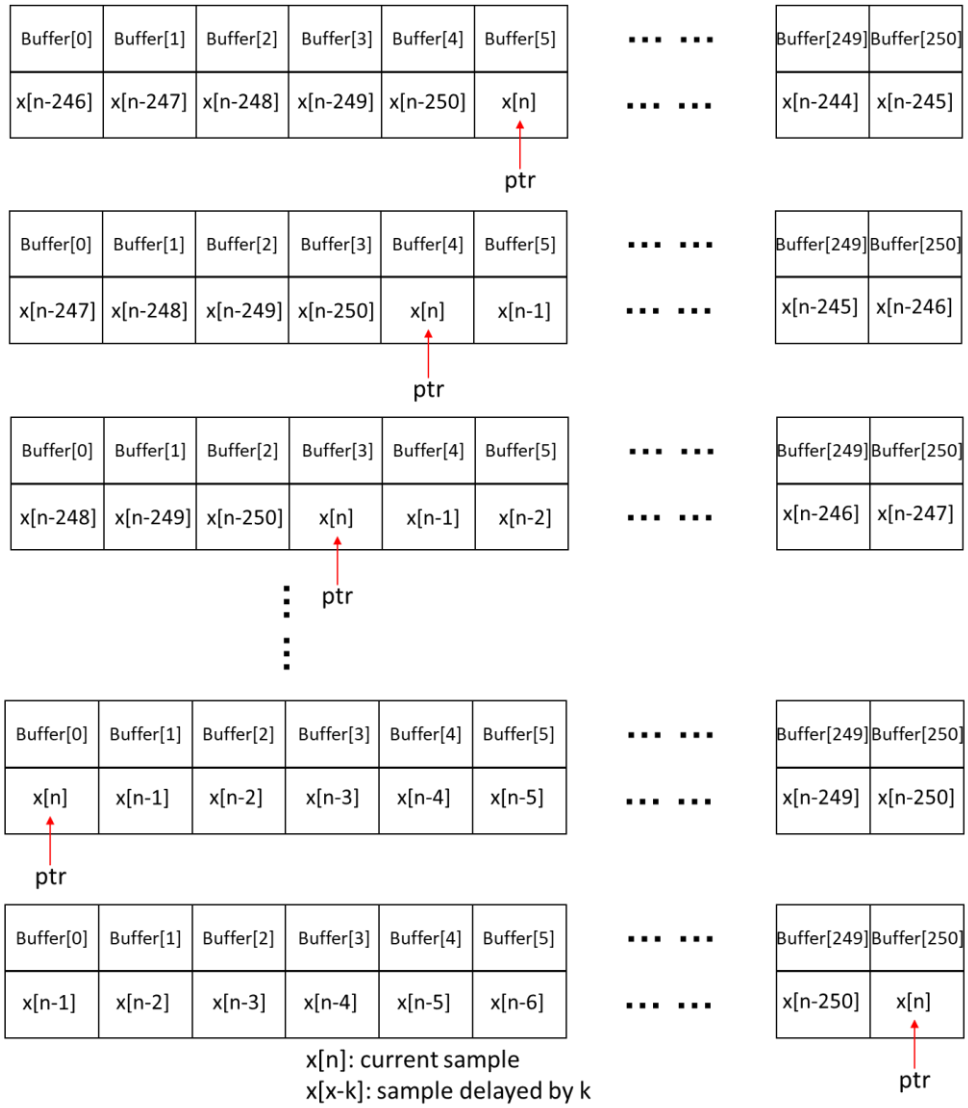


Figure 6

3.2. Explanation of Code

3.2.1. Among the global declaration:

```
#include <stdint.h>       //in order to use uint8_t
#define buffer_length 251     //equal to the number of coefficients

// manually set this to buffer_length/2 (even)
// manually set this to buffer_length/2 - 1 (odd)
#define half_buffer_length 124

Int16 buffer[buffer_length - 1] = {0};   //buffer index from 0 to buffer_length-1
int ptr = buffer_length - 1;       //ptr range from 0 to buffer_length-1
```

We include a header “stdint.h” for later usage on purpose. The coefficients are the same as using the non-circular filter, although their algorithms are quite different. Before, we treat the delayed sample and current sample separately, so the array size was 250. Here the “buffer_length” is basically the same thing per se except we include the current sample in the buffer. Thus, it is 251. We define another value called “half_buffer_length” for improvement later. This is equal to half of the “buffer_length” when “buffer_length” is an even number, or half of the “buffer_length” deducted by one when “buffer_length” is odd. Apart from that, we declare a new global variable “ptr” meaning the pointer. The pointer acts like an offset telling us where the beginning of the buffer array is each time.

3.2.2. Prime function in the ISR:

```
void ISR_AIC(void)
{
    Int16 samp_out = 0;

    buffer[ptr] = mono_read_16Bit();  // ptr points to the most current sample
    samp_out = round(circ_buff_FIR_v1());
    mono_write_16Bit(samp_out);

    ptr--;
    if(ptr < 0){
        ptr = buffer_length - 1;
    }
}
```

Like the non-circular FIR filter, the “samp_out” is declared to be a 16bits signed integer, and it is assigned to be the rounding value of the return value of our different versions of circular buffer FIR filter. There are two main differences here. Firstly, a pointer “ptr” is introduced and is always pointing the position of the most current sample in the buffer. Secondly, it decrements in each call of the ISR function. When it becomes negative, it wraps around and is assigned to be buffer_length-1, because the buffer index ranges from 0 to buffer_length-1. If we want to use different versions of our circular buffer, we can just change the version number at the end of the name of the function we call.

3.2.3. Version 1 (using a for loop and modulo operator):

```
double circ_buff_FIR_v1(void)
{
    double sum = 0;
    int i = 0;
    int buffer_index = 0;

    for(i=0;i<buffer_length;i++){
        buffer_index = (ptr + i) % buffer_length;
        sum += b[i] * buffer[buffer_index];
    }

    return sum;
}
```

The "buffer_index" is a temporary variable because we do not want the "ptr" to change during the calculation causing problems. In the for loop, initially i = 0 and the b[0] is multiplied with the most current sample "buffer[ptr]". Because the "buffer_index" may exceed the buffer length so it is calculated as the remainder of (ptr+i) divided by the buffer length. After calculating the sum, the "ptr" is decremented by one to prepare for reading the next sample in the next interrupt.

However, this method does not improve the efficiency and it even makes the situation worse. The result of using modulo is that the speed has increased a lot, because modulo is a process of division, which takes lots of cycles to finish.

By setting a breakpoint at the beginning of this function, we found that it takes minimally 17009 cycles to calculate one sample shown in the Figure 7 (full benchmark data will be presented later). It takes around 900 cycles more than using the non-circular filter. This is primarily due to the inefficiency of the modulo "%" operator. It calculates the division in each iteration in the for loop.

Optimization Level	Non-circular	Circular Version 1
None	16170	17009
-o0	13179	15705
-o2	3261	8582

Figure 7: Filter order is 250

3.2.4. Version 2 (using a for loop and if statement):

```
double circ_buff_FIR_v2(void)
{
    double sum = 0;
    int i = 0;
    int buffer_index = ptr;

    for(i=0;i<buffer_length;i++){
        if(buffer_index >= buffer_length){
            buffer_index -= buffer_length;
        }
        sum += b[i] * buffer[buffer_index++];
    }

    return sum;
}
```

This version is similar as version 1. The modulo operator is now replaced by an if statement to avoid calculating division every time. The if statement, or the comparison, takes less clock cycles than doing division. Also, instead of assigning the "buffer_index" each time in the for loop, we just increase it by one. And the increments or decrements are performed directly after the appearance of the variables in the indexes, both for the "buffer_index" and the "ptr". This saves few more cycles in each iteration.

In this version of the implementation of the circular buffer filter, if no optimization is used, it takes at least 14502 cycles to finish one interrupt routine (Figure 8). It takes approximately 2500 cycles less than the version one by replacing the modulo operator by an if statement.

It seems like there is not much room for optimization because we cannot do much to improve the MAC procedure in C. However, we can still reduce the number of unnecessary comparisons (the if statement) in the for loop. So we have to change the for loop into something similar.

Optimization Level	Non-circular	Circular Version 2
None	16170	14502
-o0	13179	11435
-o2	3261	1442

Figure 8: Filter order is 250

3.2.5. Version 3 (using two while loops):

```
double circ_buff_FIR_v3(void)
{
    double sum = 0;
    int i = 0;
    int buffer_index = ptr;

    while(buffer_index < buffer_length){
        sum += b[i++] * buffer[buffer_index++];
    }
    buffer_index = 0;
    while(buffer_index < ptr){
        sum += b[i++] * buffer[buffer_index++];
    }

    return sum;
}
```

Now the for loop before is replaced by two similar while loops. But the program is doing the exact same thing. Once the "buffer_index" is equal to the buffer length, it is assigned to zero. This is equivalent to being subtracted by the buffer length when it is equal to the buffer length.

The terminology of what we are doing here is "loop unrolling" or "loop unwinding". The goal of it is to reduce the instructions in loops in order to optimize the program. The code looks longer than before but it should take fewer cycles to run theoretically, because there is no comparison inside those loops any more.

Using no optimization, this version takes at least 12510 cycles (Figure 9) for one running of the function. This is another 2000 cycles improvement comparing to version 2. However, if the optimization level is -o2, it takes much longer than not doing a loop unrolling. Thus, the effectiveness of reducing clock cycles by just doing loop unrolling is questionable.

Optimization Level	Non-circular	Circular Version 3
None	16170	12510
-o0	13179	9702
-o2	3261	1982

Figure 9: Filter order is 250

3.2.6. Version 4.1 (using the symmetry property of the coefficients - even version):

```
double circ_buff_FIR_v4_even(void)
{
    double sum = 0;
    int i = 0;
    int buffer_index1 = ptr;
    int buffer_index2 = ptr-1;

    for(i=0;i<half_buffer_length;i++){
        if(buffer_index1 >= buffer_length){
            buffer_index1 = 0;
        }
        if(buffer_index2 < 0){
            buffer_index2 = buffer_length - 1;
        }
        sum += b[i] * (buffer[buffer_index1++] + buffer[buffer_index2--]);
    }

    return sum;
}
```

For a causal and linear-phase FIR filter, its coefficients are symmetrical around the center. If we split the coefficients of a 249th order filter by half, we'll get the coefficients from 0th to 124th exactly the same as the coefficients from 249th to 125th. Therefore, we may only use the lower half of the whole coefficient array and discard the rest half completely. The code does not require further comments since it is readable and understandably. You can see the visualization later.

This algorithm will reduce the convolution target load by half, and it basically adds up the sample values which will be multiplied by the same coefficient, and multiplies the sum with the corresponding coefficient. The algorithm can be visualized in Figure 11.

This version requires less than 10k number of cycles using no optimization (Figure 10). It runs roughly twice the speed of the non-circular buffer. If using the -o2 level of optimization, it takes 1123 cycles per iteration, which is approximately 1/3 of using non-circular buffer.

Optimization Level	Non-circular	Circular Version 4(even)
None	16170	9815
-o0	13179	6924
-o2	3261	1123

Figure 10: Filter order is 249, No. of coefficients is 250 (even)

The visualization of our algorithm of version 4.1 (Even):

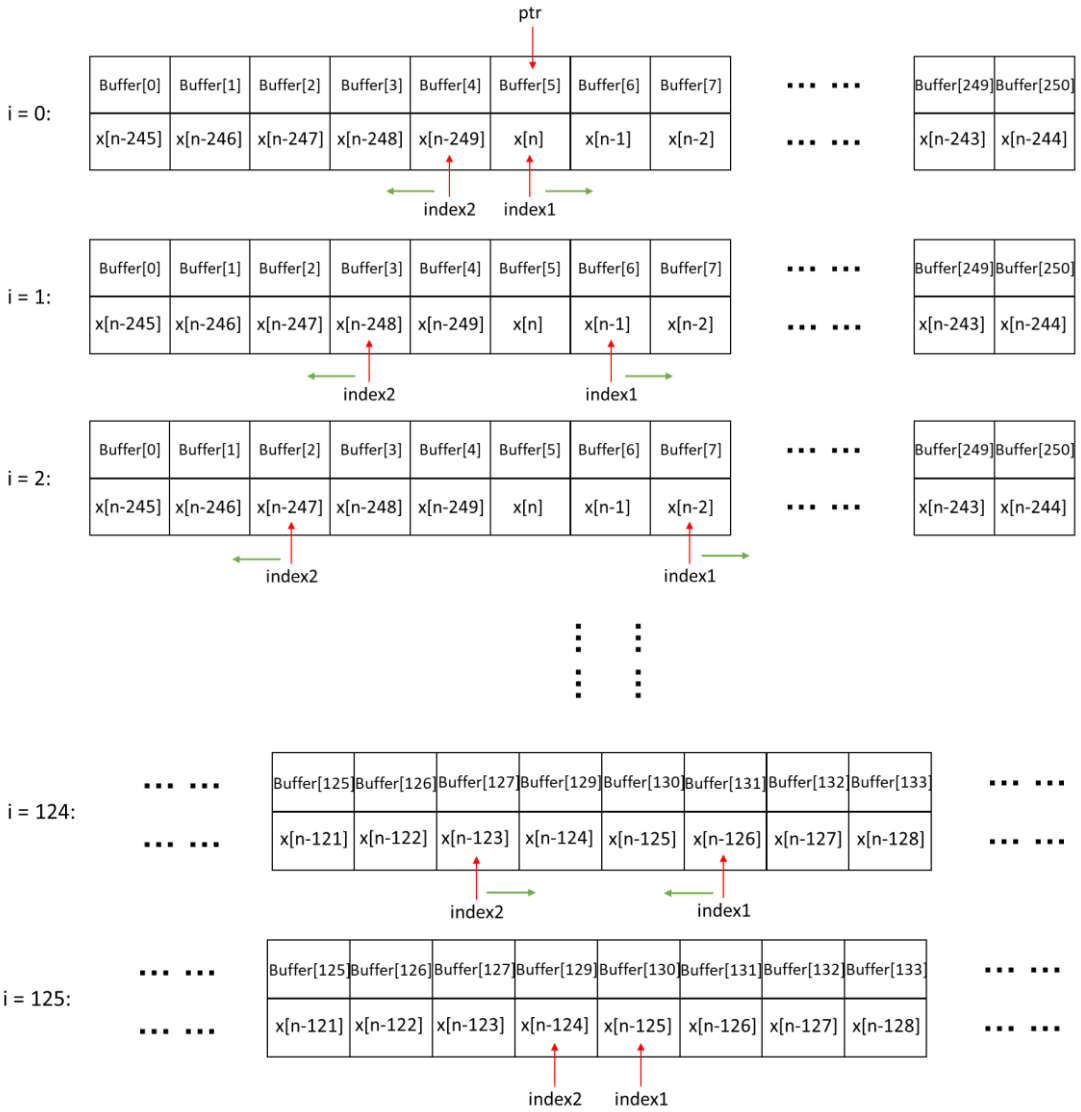


Figure 11: No. of coefficients is 250 (even)

3.2.7. Version 4.2 (using the symmetry property of the coefficients - odd version):

```
double circ_buff_FIR_v4_odd(void)
{
    double sum = 0;
    int i = 0;
    int buffer_index1 = ptr;
    int buffer_index2 = ptr-1;

    for(i=0;i<half_buffer_length;i++){
        if(buffer_index1 >= buffer_length){
            buffer_index1 = 0;
        }
        if(buffer_index2 < 0){
            buffer_index2 = buffer_length - 1;
        }
        sum += b[i] * (buffer[buffer_index1++] + buffer[buffer_index2--]);
    }

    // special case for odd No. of coefficients. When ptr=buffer_length/2,
    // buffer_index would equal buffer_length exceeding the buffer[] array.
    // buffer[] starts from zero ends at buffer_length-1 (see line 54)
    if(buffer_index1 == buffer_length){
        buffer_index1 = 0;
    }

    sum += b[i] * buffer[buffer_index1];

    return sum;
}
```

It is very unexpected to see that this version with odd filter coefficient is much faster than the even version in above. More importantly, this is our fastest version ever, which accounts for 643 cycles (see Figure 12). The methodology is similar to the previous one, so no further explanation needed. The visualization explanation can be seen in below.

The reason of saying unexpected outcome is that this reduces almost a half cycles time than the even version 4.1, even we choose a filter order larger than 249 (used in even version). Figure 13 visualizes the shifts of indexes for a circular buffer FIR filter of order 250.

We have checked in the Spectrum Analyzer is correct, which will be presented in later section.

Optimization Level	Non-circular	Circular Version 4(even)
None	16170	9862
-o0	13179	6970
-o2	3261	654

Figure 12: Filter order is 250, No. of coefficients is 251 (odd)

The visualization of our algorithm of version 4.2 (Odd):

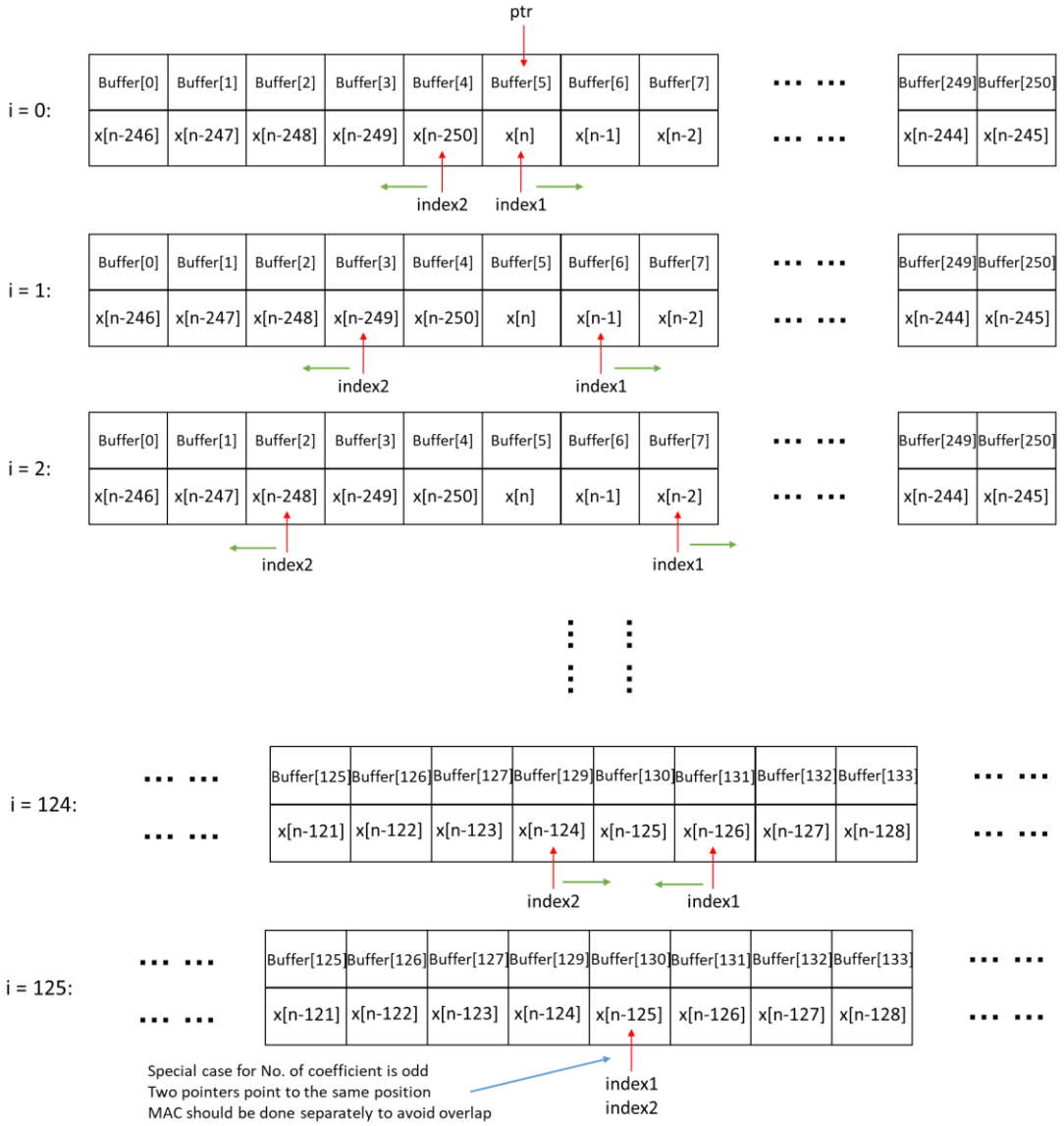


Figure 13: No. of coefficients is 251 (odd)

3.2.8. Version 5 (special number of coefficients 256 - traverse):

```
double circ_buff_FIR_v5(void)
{
    double sum = 0;
    int i = 0;
    uint8_t buffer_index = ptr;  //8bits unsigned int range 0-255

    for(i=0;i<buffer_length;i++){
        sum += b[i] * buffer[buffer_index++];
    }

    return sum;
}
```

However, instead of using filter 248, we can switch to order 256 to increase the speed. Because 256 is 2 to the power of 8, and in the sense of assembly code, this is just cost arithmetic right shift of eight bits. Order 248 is not the result of power of two, so this will cause more cycles than order 256.

In version 5, our "buffer_index" has defined in uint8_t, which means instead of using an int pointer type, we are going to improve based on the data type. The trick part in here is that when the "buffer_index" overflows, uint8_t will wrap around by itself. This method can only work for the situation where the number of coefficients is 256 or the order of the filter is 255. This eliminates the if statement in the for loop, like in version 2, effectively reduces a lot of repeating instructions.

This version does not include any algorithms using the symmetry property, so it is slower compare to the previous version. It takes 3663 cycles using -o2 level of optimization, even slower than the non-circular version (Figure 14). So traversing the whole buffer array is a very inefficient way of designing an FIR filter. We still need to use the symmetry property of the linear-phase FIR filter to boost the efficiency.

Optimization Level	Non-circular	Circular Version 5
None	16170	11849
-o0	13179	9527
-o2	3261	3663

Figure 14: Filter order is 250

3.2.9. Version 6 (special number of coefficients 256 - using symmetry):

```
double circ_buff_FIR_v6(void)
{
    double sum = 0;
    int i = 0;
    uint8_t buffer_index1 = ptr;  //8bits unsigned int range 0-255
    uint8_t buffer_index2 = ptr-1;  //8bits unsigned int range 0-255

    for(i=0;i<half_buffer_length;i++){
        sum += b[i] * (buffer[buffer_index1++] + buffer[buffer_index2--]);
    }

    return sum;
}
```

By using symmetry property of a linear-phase FIR filter, we have observed that the speed of the algorithm has improved dramatically. We would now apply the same methodology on the special number filter order 256 case.

Firstly, instead of just using one index, we are going to use two indexes with type uint8_t. The reason of using this type is same as version 5 in above, and only half of the sample buffer is used.

By measuring the clock cycles, the function only takes 776 cycles (Figure 15), which is the second fastest. However, this version again only works when the number of coefficients is exactly 256.

Optimization Level	Non-circular	Circular Version 6
None	16170	6985
-o0	13179	5309
-o2	3261	776

Figure 15: Filter order is 250

3.3. Full Benchmark

Number of cycles		No. of coefs	Filter order	Optimization Level		
				None	-o0	-o2
Non-Circular FIR Filter		251	250	16170	13179	3261
Circular Buffer FIR Filter	Version 1	251	250	17009	15705	8582
	Version 2	251	250	14502	11435	1442
	Version 3	251	250	12510	9702	1982
	Version 4(Even)	250	249	9815	6924	1123
	Version 4(Odd)	251	250	9862	6970	654
	Version 5	256	255	11849	9527	3663
	Version 6	256	255	6985	5309	776

Figure 16: Full Benchmark – Table

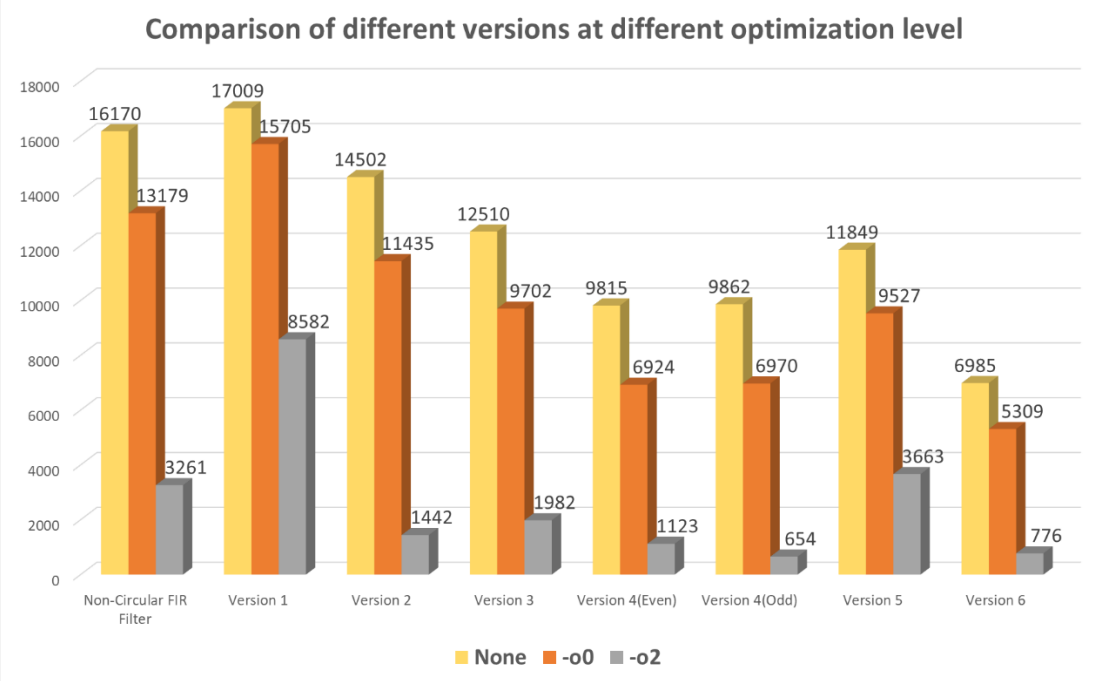


Figure 17: Full benchmark – Chart

The version 1 of the circular buffer is much slower than others, regardless of what the optimization level is. This is purely due to the modulo operator doing a division in each iteration in the for loop.

From Figure 16, we conclude that if we use no optimization or "-o0" level of optimization, the version 4 of the circular buffer FIR filter is the fastest. If we use

optimization level "-o2", the version 2, 3, 4 have not much difference. This is because we used C language when implementing the FIR filter, which is a high-level programming language. The C language requires minimal effort from programmers comparing to linear assembly language and hand optimized assembly. However, the higher level of the language we use, the less control we have, consequently the inefficient or slower the code is. Although we tried our best optimizing the code in C, there is still not much difference when the compiler converts the C script into machine code using "-o2" level of optimization.

3.4. Summary

In summary, without optimization, the fastest version is Version 4(odd), which clearly shows in Figure 17. With o0 optimization, Version 6 is the fastest, and with o2 optimization, Version 4(odd) and Version 6(256) are very close.

4. Actual frequency response using APx500

In this exercise, we are going to check whether the frequency response output by us has any difference between the theoretically response output by Matlab.

In the following exercise, we are just using one filter order 250 to discuss the gain and the phase behavior in the actual frequency response.

Set up APX500 Audio Analyzer, with setting:

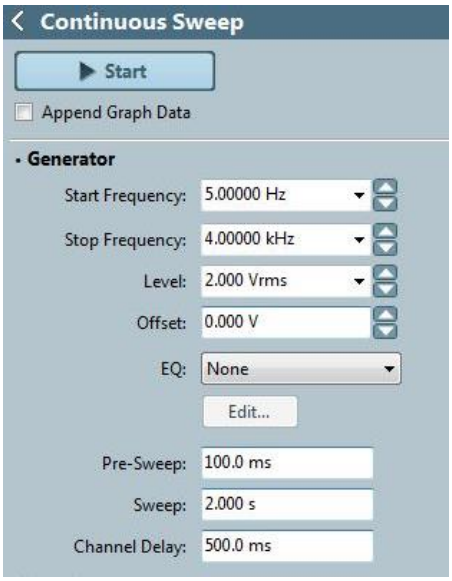


Figure 18: APX500 Audio Analyzer settings

4.1. Gain Response

Plot graphs in Matlab (ideal output):

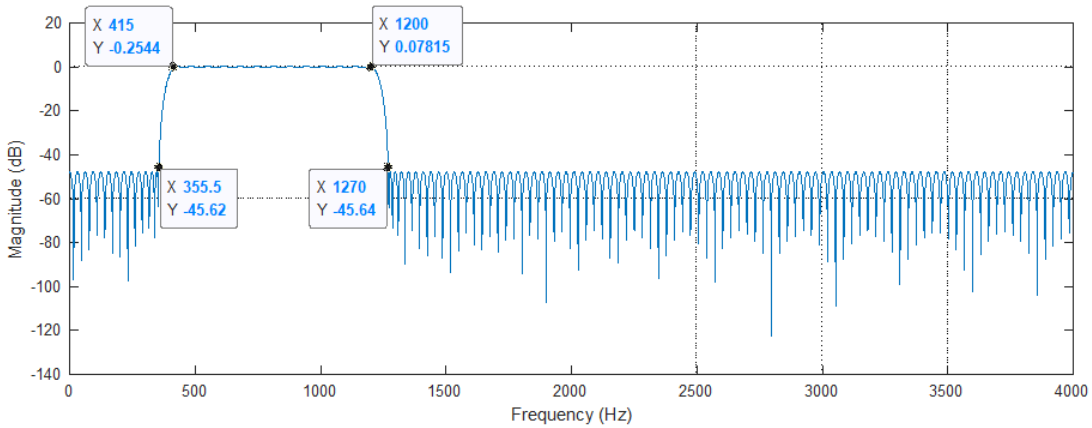


Figure 19: Frequency response (Gain) in Matlab

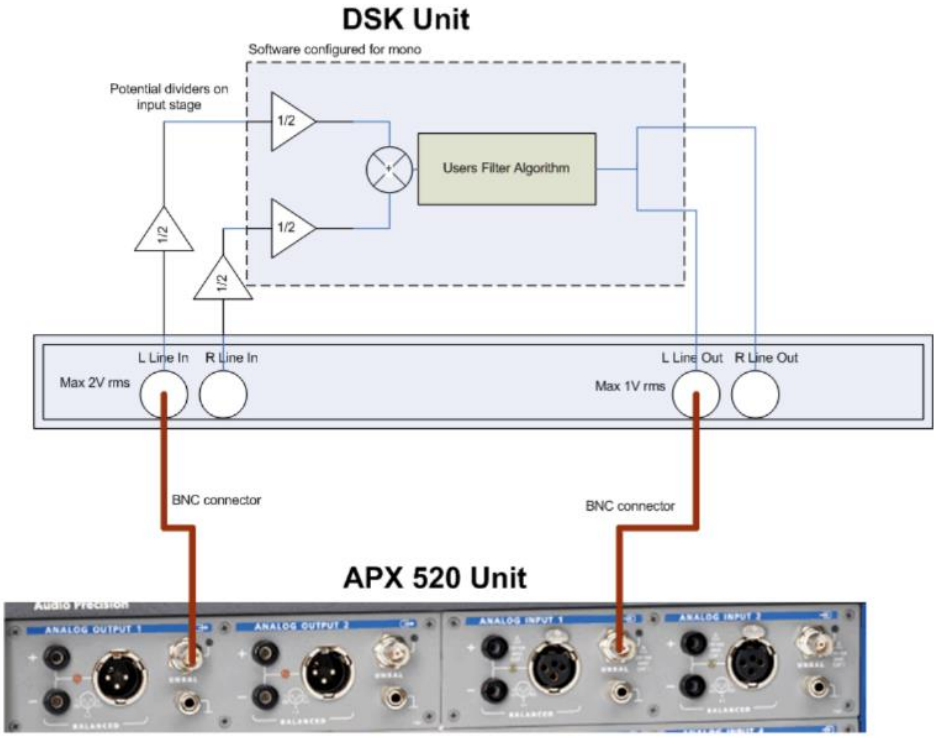


Figure 20: DSK unit

By plotting spectrum in Spectrum Analyzer, we can see that the approximate mean of the ripple gain is at -12.84dB (see Figure 21). The reason of this is going to explain.

By asking GTA, in theory, the gain should be at –12dB. The reason is that from Figure 20, it is easy to see that there is only one connection between APX 520 Unit and DSK input in each side, so this will rise output in the device four times larger. When the

signal gets into the DSK unit, it will encounter a potential divider, has shown in above. This will attenuate the signal by 1/2, and moreover, there will also some attenuation will take place due to the C code mono_read_16Bit() (told by GTA). These two factors will contribute the -12dB gain. However, instead of showing -12dB of what we expect, it has shown about -12.84dB. I have discussed with my lab partner, we all think this is because of the noise and non-ideal resistor value.

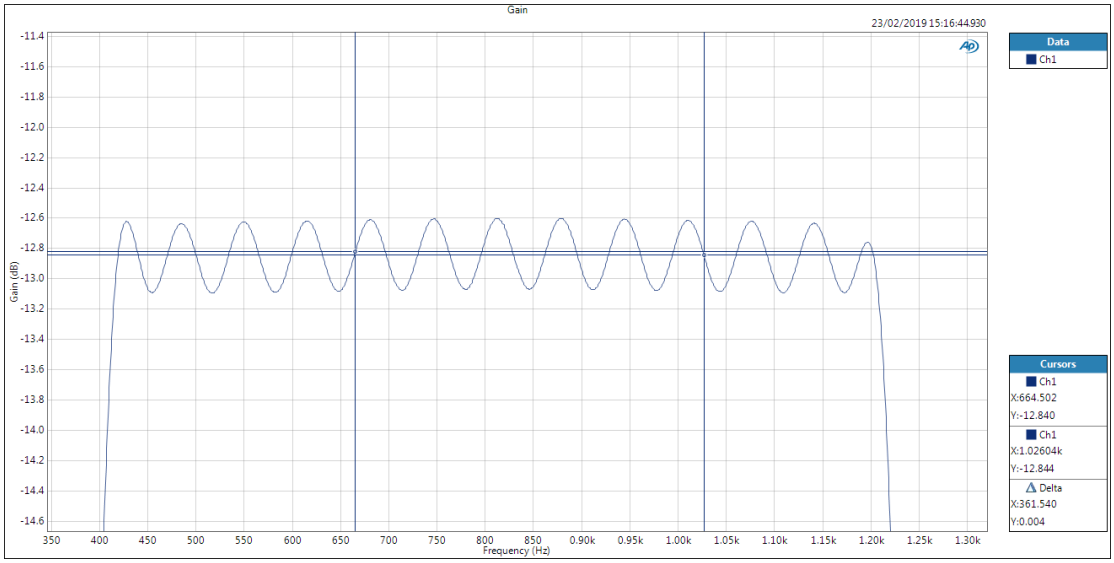


Figure 21: Ripple mean

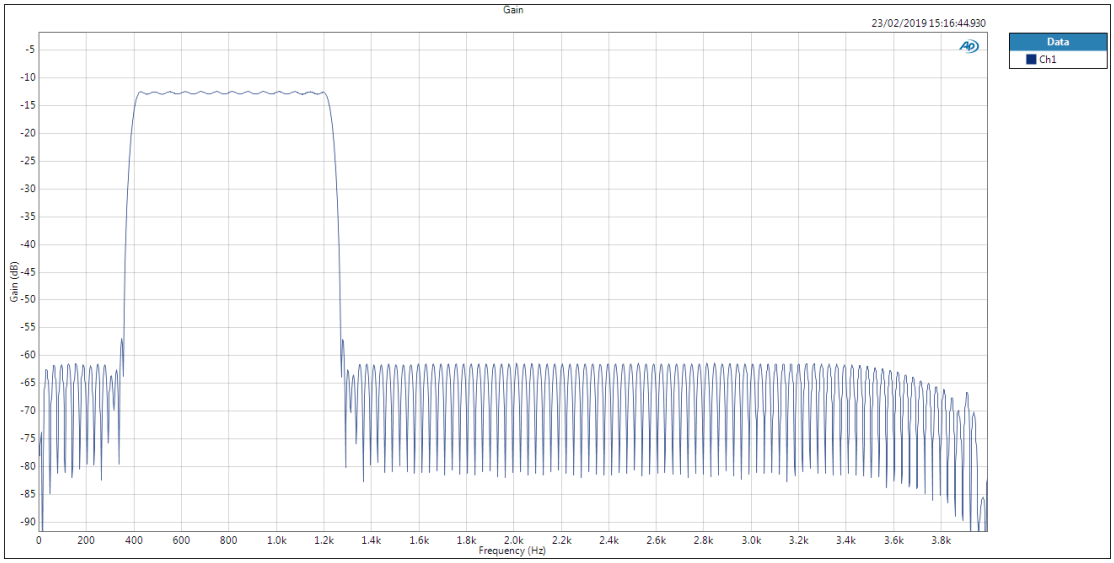


Figure 22: Actual frequency response (Gain overview)

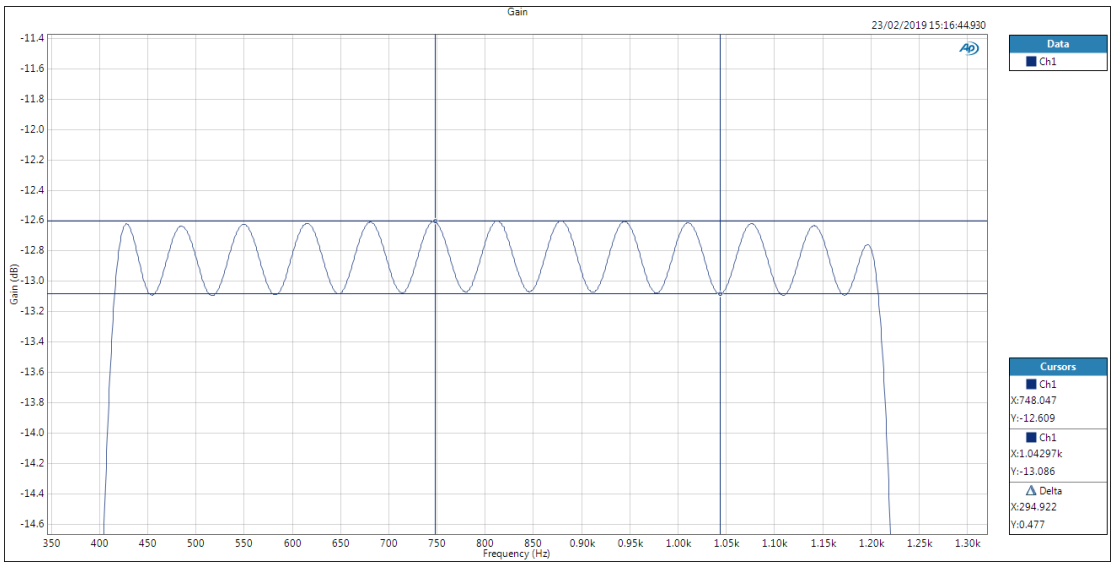


Figure 23: : Actual frequency response (Ripple)

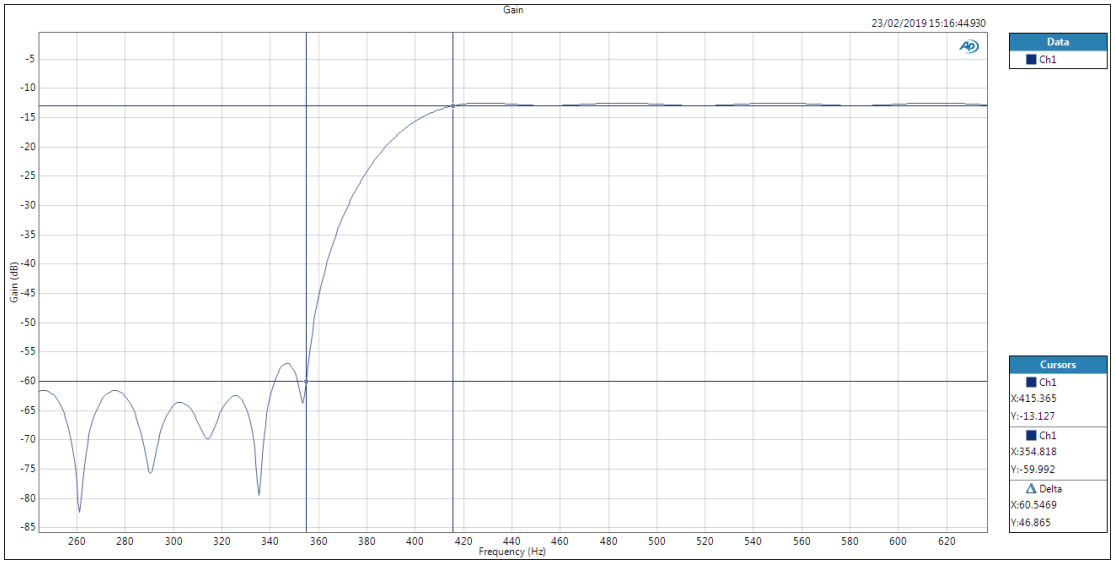


Figure 24: : Actual frequency response (Lower part)

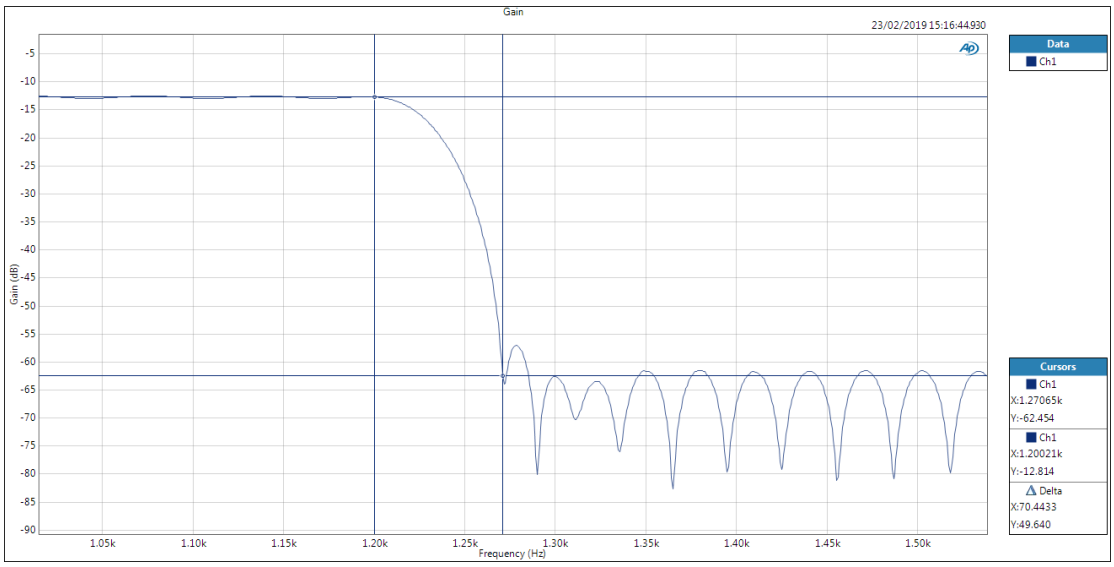


Figure 25: : Actual frequency response (Upper part)

At last, by comparing to the Matlab output:

Comparison of 250th Order Circular Buffer FIR Filter				
Aspects	**Matlab** /dB	**Spectrum Analyzer** /dB	**Spectrum Analyzer** (corrected) /dB	**Delta** /dB
Shape	original	similar	similar	ignorable
Overall Gain	0	-12.863	-0.023	0.023
Gain at 355Hz	-45.62	-59.992	-47.152	1.532
Gain at 415Hz	-0.254	-13.127	-0.287	0.033
Gain at 1.2kHz	0.78	-12.814	0.026	0.754
Gain at 1.27kHz	-45.64	-62.454	-49.614	3.974

Figure 26

From Figure 26, we agree the output of our algorithm works as expected, because the error is within the tolerance range.

4.2. Phase Response

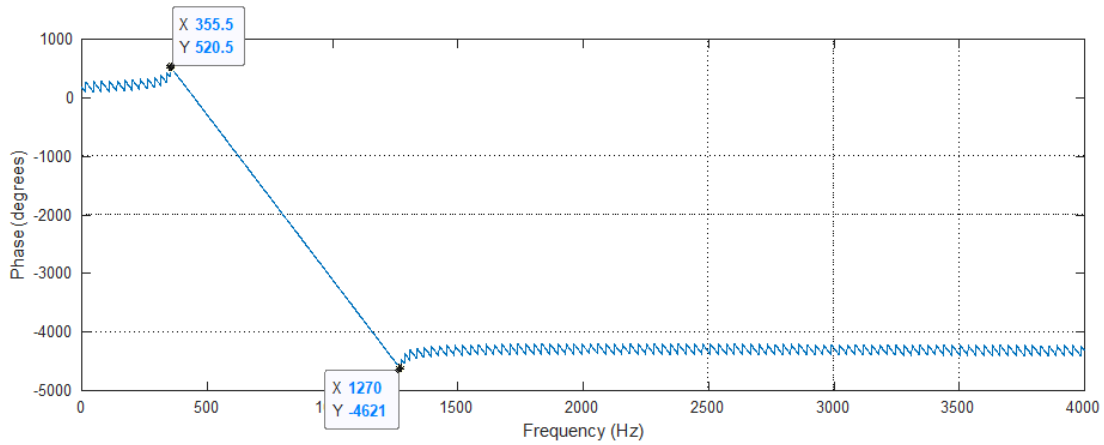


Figure 27: Phase plot in Matlab (ideal output):

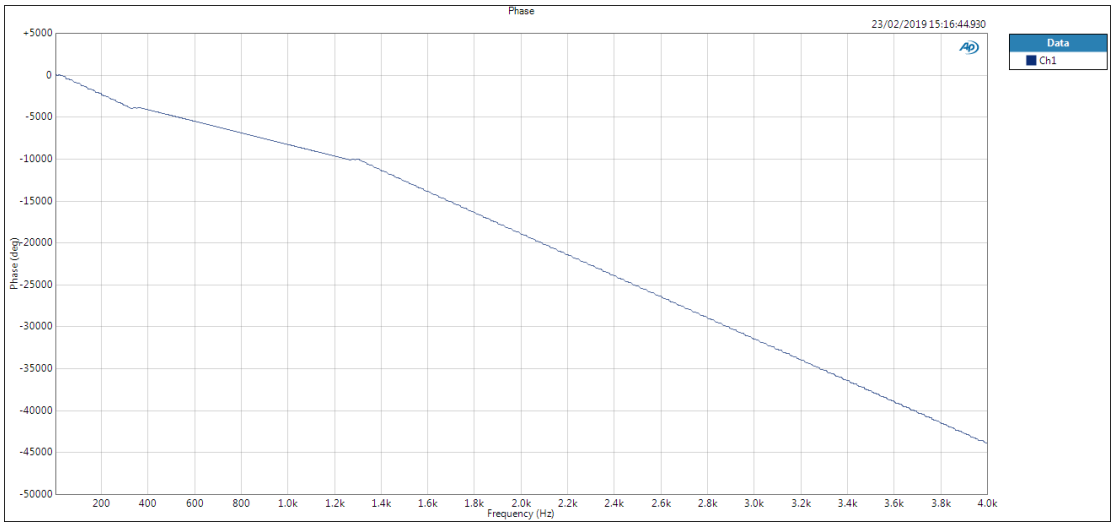


Figure 28: Frequency response (Phase)

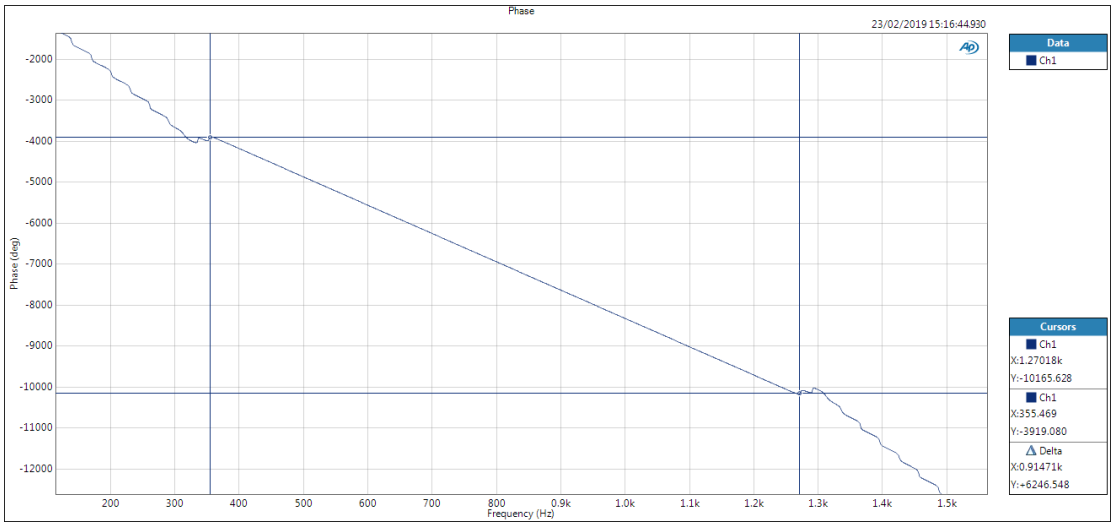


Figure 29: Frequency response (Phase) - zoomed

In Figure 29, we can see that the phase has three sections, and the desired one is a straight line with our expectation. The actual frequency response shows that the fastest version of our circular buffer FIR filter is a linear-phase FIR filter.

4.3. Group Delay

In this part, we have to mention a concept called group delay.

In signal processing, group delay is the time delay of the amplitude envelopes of the various sinusoidal components of a signal through a device under test, and is a function of frequency for each component, and it is defined by below:

Additionally, it can be shown that the group delay τ_g, and phase delay τ_φ are frequency-dependent, and they can be computed from the phase shift φ by[3]*:*

$$\tau_g = -\frac{d\varphi(\omega)}{d(\omega)}$$

$$\tau_\varphi = -\frac{\varphi(\omega)}{\omega} \quad .$$

By seeing the output of spectrum analyzer, the group delay exists, and moreover, the linear phase property has both shown in both Matlab and Spectrum Analyzer. Discussing with my lab partner, we both agree two outputs are consistent.

[3] *En.wikipedia.org. (2019). Group delay and phase delay. [online] Available at: https://en.wikipedia.org/wiki/Group_delay_and_phase_delay [Accessed 23 Feb. 2019].*

5. Appendix

5.1. C++ script to replace tabs by commas

```cpp
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

int main() {
    ifstream infile("fir_coefs_tabs.txt");
    ofstream outfile("fir_coefs.txt");

    string tmp;

    outfile << "double b[] = {";
    infile >> tmp;
    outfile << tmp;
    while (infile >> tmp) {
        outfile << ", " << tmp;
    }
    outfile << "};" << endl;

    infile.close();
    outfile.close();
}
```

5.2. Full version of C code for non-circular FIR filter

```
/*************************************************************************************
                    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>

// include the coefficient b[] calculated in Matlab
#include "fir_coefs.txt"

// define an N elements delay buffer
// an Nth order filter requires N+1 input samples and thus N coefficients
#define N 250
Int16 x[N] = {0}; //array size is N+1 which is 251

/****************************** 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);
double non_circ_FIR(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)
{
     Int16 samp_out = 0;

     samp_out = round(non_circ_FIR());
     mono_write_16Bit(samp_out);
}

double non_circ_FIR(void)
{
     double sum = 0;
     int i = 0;

     for(i=N;i>0;i--){
          x[i] = x[i-1];
     }
     x[0] = mono_read_16Bit();  //read the current sample

     for(i=0;i<=N;i++){
          sum += b[i] * x[i]; //MAC procedure
     }

     return sum;
}
```

5.3. Full version of C code for circular buffer FIR filter

```
/*************************************************************************************
                    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>

// include the coefficient b[] calculated in Matlab
#include "fir_coefs.txt"

// define an N elements delay buffer
// an Nth order filter requires N+1 input samples and thus N coefficients
// the fir_coefs.txt contains 251 coefficient so the buffer length should be 251
#include <stdint.h>     //in order to use uint8_t
#define buffer_length 251    //equal to the number of coefficients

// manually set this to buffer_length/2 (even)
// manually set this to (buffer_length-1)/2 (odd)
#define half_buffer_length 125

Int16 buffer[buffer_length - 1] = {0};   //buffer index from 0 to buffer_length-1
int ptr = buffer_length - 1;      //ptr range from 0 to buffer_length-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);
```

```
double circ_buff_FIR_v1(void);             //% modulo operator
double circ_buff_FIR_v2(void);             //if instead of %
double circ_buff_FIR_v3(void);             //loop unrolling
double circ_buff_FIR_v4_even(void);        //coefficient(even) symmetry
double circ_buff_FIR_v4_odd(void);         //coefficient(odd) symmetry (fastest)
double circ_buff_FIR_v5(void);             //256 coefficients and 8bits buffer_index
double circ_buff_FIR_v6(void);             //8bits buffer_index and symmetry

/********************************* 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)
{
     Int16 samp_out = 0;

     buffer[ptr] = mono_read_16Bit();      // ptr points to the most current sample
     samp_out = round(circ_buff_FIR_v4_odd()); // writing output to variable
     mono_write_16Bit(samp_out);

     ptr--;
     if(ptr < 0){
          ptr = buffer_length - 1; //buffer_index ranges from 0 to buffer_length - 1
     }
```

```
}


/****************** Version 6 (256 half)  ************************/
double circ_buff_FIR_v6(void)
{
     double sum = 0;
     int i = 0;
     uint8_t buffer_index1 = ptr;  //8bits unsigned int range 0-255
     uint8_t buffer_index2 = ptr-1;  //8bits unsigned int range 0-255

     for(i=0;i<half_buffer_length;i++){
          sum += b[i] * (buffer[buffer_index1++] + buffer[buffer_index2--]);
     }

     return sum;
}


/****************** Version 5 (256)  ************************/
double circ_buff_FIR_v5(void)
{
     double sum = 0;
     int i = 0;
     uint8_t buffer_index = ptr;  //8bits unsigned int range 0-255

     for(i=0;i<buffer_length;i++){
          sum += b[i] * buffer[buffer_index++];
     }

     return sum;
}


/****************** Version 4.2 (Odd)  ************************/
double circ_buff_FIR_v4_odd(void)
{
     double sum = 0;
     int i = 0;
     int buffer_index1 = ptr;
     int buffer_index2 = ptr-1;

     for(i=0;i<half_buffer_length;i++){
          if(buffer_index1 >= buffer_length){
               buffer_index1 = 0;
          }
          if(buffer_index2 < 0){
               buffer_index2 = buffer_length - 1;
          }
          sum += b[i] * (buffer[buffer_index1++] + buffer[buffer_index2--]);
     }

     // special case for odd No. of coefficients. When ptr=buffer_length/2,
     // buffer_index would equal buffer_length exceeding the buffer[] array.
     // buffer[] starts from zero ends at buffer_length-1 (see line 54)
     if(buffer_index1 == buffer_length){
          buffer_index1 = 0;
     }

     sum += b[i] * buffer[buffer_index1];

     return sum;
}


/****************** Version 4.1 (Even)  ************************/
double circ_buff_FIR_v4_even(void)
{
     double sum = 0;
     int i = 0;
     int buffer_index1 = ptr;
     int buffer_index2 = ptr-1;

     for(i=0;i<half_buffer_length;i++){
```

```
          if(buffer_index1 >= buffer_length){
               buffer_index1 = 0;
          }
          if(buffer_index2 < 0){
               buffer_index2 = buffer_length - 1;
          }
          sum += b[i] * (buffer[buffer_index1++] + buffer[buffer_index2--]);
     }

     return sum;
}


/****************** Version 3 (loop unrolling)  ************************/
double circ_buff_FIR_v3(void)
{
     double sum = 0;
     int i = 0;
     int buffer_index = ptr;

     while(buffer_index < buffer_length){
          sum += b[i++] * buffer[buffer_index++];
     }
     buffer_index = 0;
     while(buffer_index < ptr){
          sum += b[i++] * buffer[buffer_index++];
     }

     return sum;
}


/************************** Version 2 (using if) ******************************/
double circ_buff_FIR_v2(void)
{
     double sum = 0;
     int i = 0;
     int buffer_index = ptr;

     for(i=0;i<buffer_length;i++){
          if(buffer_index >= buffer_length){
               buffer_index -= buffer_length;
          }
          sum += b[i] * buffer[buffer_index++];
     }

     return sum;
}


/************************** Version 1 (using %) ******************************/
double circ_buff_FIR_v1(void)
{
     double sum = 0;
     int i = 0;
     int buffer_index = 0;

     for(i=0;i<buffer_length;i++){
          buffer_index = (ptr + i) % buffer_length;
          sum += b[i] * buffer[buffer_index];
     }

     return sum;
}
```