RTDSP LAB 4

Yihan Qi, CID: 00813873

Lizhang Lin, CID: 00840705

Contents

1. Filter Design Using Matlab

In this lab session, we were given a design specification of a FIR filter, which is shown in the figure 1 below.

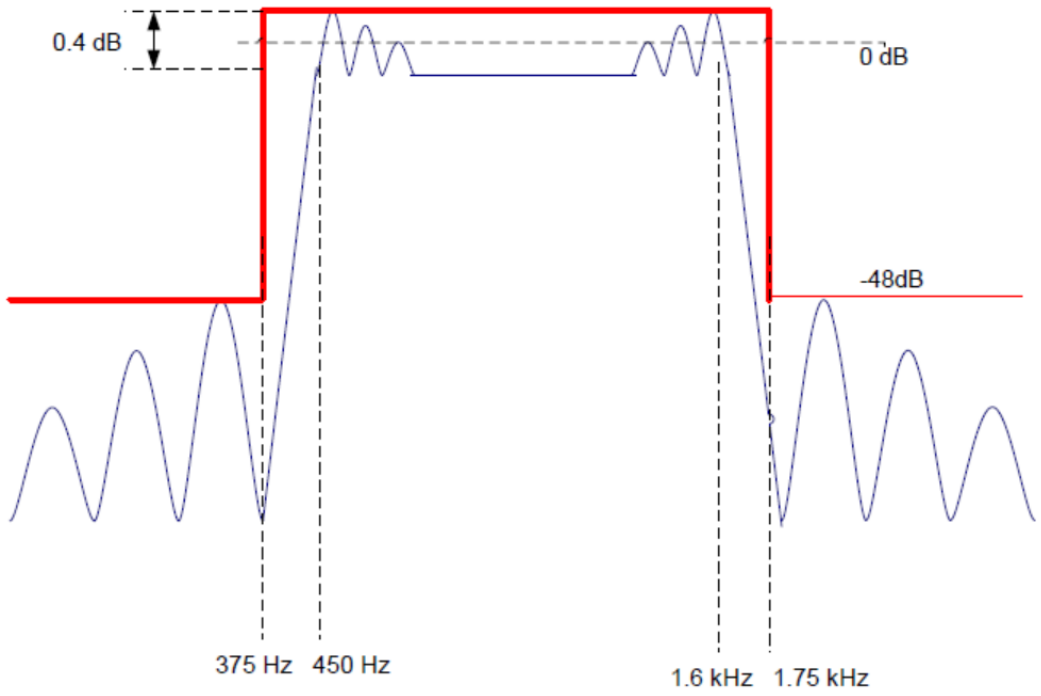


Figure 1: design specification for the FIR bandpass filter

To achieve this, we used Matlab to implement such a FIR filter that satisfies the specifications given above and saved all the coefficients in a file called ***'file_b'***. A detailed code and its explanation can be found in the following paragraph.

```
rp = 0.4;                    % Passband ripple
rs = 48;     % Minimum stop band attenuation
fs = 8000;                   % Sampling frequency
f = [375 450 1600 1740];  % passband and stopband Cutoff frequencies
a = [0 1 0];      % Desired amplitudes
dev = [10^(-rs/20) (10^(rp/20)-1)/(10^(rp/20)+1) 10^(-rs/20)];% maximum allowance ripple

[n,fo,ao,w] = firpmord(f,a,dev,fs);%calculate an approximate filter
%n is filter order
%fo stands for the normalised frequency edges
%a0 is the frequency amplitudes
%w stands for the weights

b = firpm(n+7,fo,ao,w); %b is a vector which stores the coefficients

freqz(b,1,1024,fs);%magnitude and phase plots

%create a text file called 'file_b.txt' where coefficients are stored and
%elements are defined
%define N
file = fopen('file_b.txt','wt');
fprintf(file, '#define N %d \n', n+8);
fclose(file);
```

```
%store coefficients into the file
file = fopen('file_b.txt','at');
fprintf(file,'double b[]={');
dlmwrite ('file_b.txt',b,'-append');
fprintf(file,'};');
fclose(file);
```

Firstly, we defined the parameters such as sampling frequency ***fs*** and vector ***f*** which stores passband and stopband cutoff frequencies. We calculated deviation and desired amplitudes in stopbands and passband, and stored the values in vector ***dev*** and ***a*** respectively.

Secondly, we used the parameters defined previously and applied them into function ***firpmord*** to calculate the approximate required filter order ***n***, the normalised frequency edges ***fo***, the amplitudes ***ao*** for frequency band and weights, ***W***, which would be used by the next function ***firpm*** to meet our design specification. Note that in function ***firpm***, ***(n+7)*** was chosen as the order of the filter rather than ***n***. This is because ***n*** is an approximate filter order generated by the function ***firpmord,*** which is not accurate to meet the design specification. Thus, after a few tests, we found that filter order of ***(n+7)*** brings the desired FIR filter. And the coefficients were saved in vector ***b***.

Magnitude and phase responses of the filter shown below are achieved by using command `freqz(b,1,1024,fs)`.

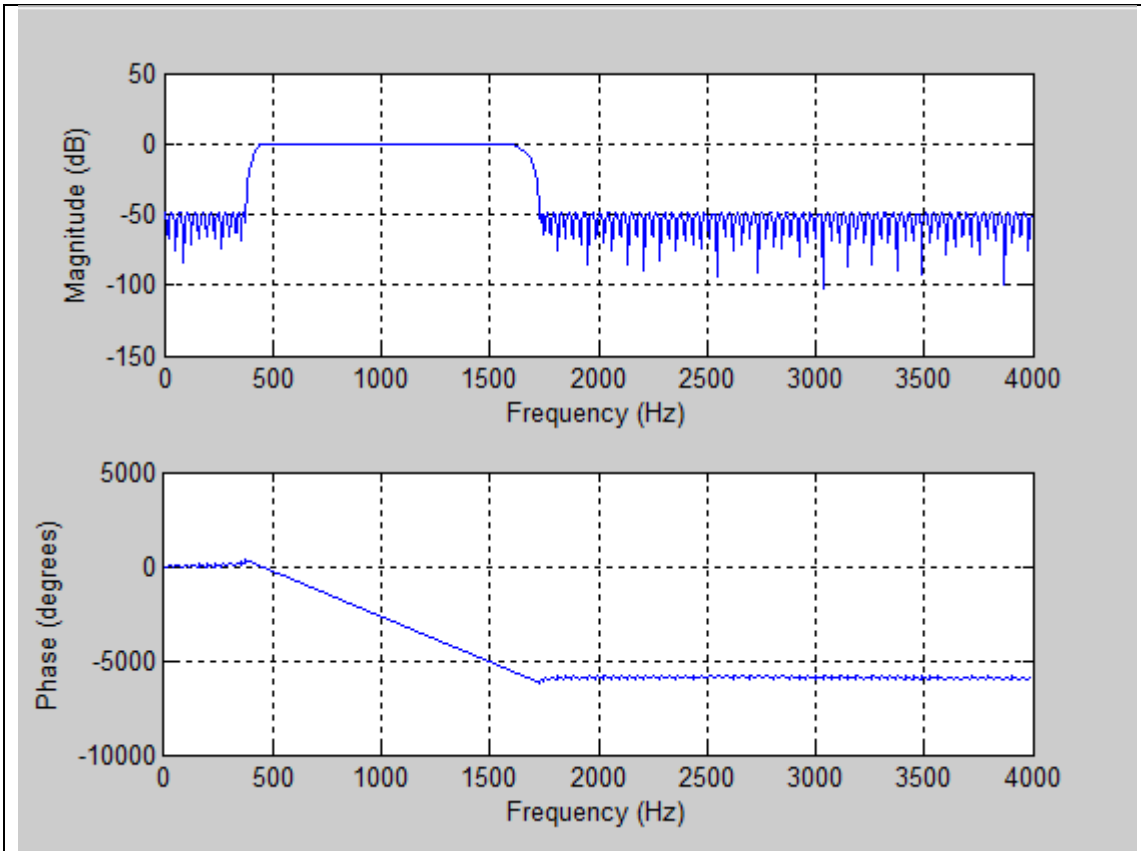


Figure 2: magnitude and phase response of the filter which meets the design specification.

Thirdly, we wrote a text file called '***file_b.txt***'. Inside it, C++ instructions were written to create a vector ***b*** which stores all the coefficients of the filter in ***double*** precision and this file also defines a parameter ***N***, which is the total number of coefficients stored in the vector ***b***. Note that ***N*** equals to ***(n+8)*** instead of the filter order, ***(n+7)*** for the reason that the total number of coefficients equals to the filter order plus one.

Note: All the coefficients and calculations used double precision during the process of measuring and comparing performance for different filter implementations in this lab session.

2. Filter Implementation

In this section, we copied files from lab 3, and made necessary edits to the ***intio.c*** file.

U Having designed a bandpass digital filter using Matlab and saved all the coefficients in a file called ***'file_b'***, we then started to implement filters. In the following subsections, a basic non-circular buffer FIR filter was firstly introduced and investigated. Then different versions of circular buffers with improved operation speeds were implemented in the section 2.2 which also includes detailed code explanations and comparisons of all the methods at different compiler optimisation levels.

2.1. Basic Non-Circular Buffer Implementation

The main idea of convolution using non-circular buffer can be described by the following equation:

$$output = \sum_{i=0}^{N-1} b[i] \times x[i]$$

Where ***N*** equals to the total number of coefficients of the filter, which are stored in the array ***b[]***.

x[i] stands for buffer with index ***i***. In this design, we saved the most current inputs in the buffers with lower indexes, and the past inputs in the buffers with higher indexes. E.g. ***x[0]*** stores the current input, ***x[1]*** stores the previous input, and etc.

b[i] is the ***(i+1)***th coefficient of the digital filter.

Therefore, output is the result of multiply accumulations. We would further discuss how the idea was implemented in the following subsection, and demonstrate the performance of our design in the later subsection.

We wrote two functions to implement a non-circular buffer. One is the ISR function and the other is called ***non_circ_FIR()***.

In order to realize a FIR filter using non-circular buffers, we implemented a function that fetches the current sample and stores it into the first element of a array which is defined , ***x[0]***. and N-1 previous samples are processed at the same moment, where N is the order of filter. Then, this function should be able to perform convolution after delay operation. Successively, output y(n) is stored to the audio port. In this very beginning intuitive design, we were regardless any algorithm or data structure optimizations and, compared to the improved version in later sessions, we called this non-circular buffer filter, non_circ_FIR().

2.1.1. Code Explanation

The idea of using non-circular buffer for convolution was implemented using an ISR and a function which will be called by the ISR. We kept the same configuration settings as in lab3 exercise one for the ISR but made some modifications on the code (see the code below). Thus every time the interrupt is initialized, the function ***non_circ_FIR()*** is called (the code below) by the interrupt and it returns an output which is casted into 16-bit ***int*** and written back to the DAC in the DSP kit.

```
/********************* WRITE YOUR INTERRUPT SERVICE ROUTINE HERE**********************/
void ISR_AIC(void){
   mono_write_16Bit((Int16) (non_circ_FIR()));//write sample back to memory using non-circular convolution
                                 //casting double into 16 bit int data type.
}
```

```
/********************* NON-CIRCULAR FILTERING CODE CALLED BY A ISR**********************************/
double non_circ_FIR(void){
//this function performs non-circular convolution

   x[0] = (double) mono_read_16Bit();  // reads input and convert the input into double data type and put it into the buffer
   output = 0.0; //reset output to 0
   for(i=0; i<N; i++){//convolution
      output += x[i]*b[i];//multiply accumulation
   }
   for(i=N-1; i>0; i--){
      x[i] = x[i-1]; // move data along buffer from lower element to next higher
   }
   return output;
}
```

The function ***non_circ_FIR()*** does everything except writing sample back to the memory. Basically, the code firstly reads an input from the ADC, casting it into ***double*** data type, and stores it into the buffer ***x[0]***, which is always the memory location for new inputs. Then we used two ***for*** loops. The first one implements convolution as described by the equation we just mentioned, $output = \sum_{i=0}^{N-1} b[i] \times x[i]$, and updates the global ***double*** data type variable ***output***, which is always reset to 0 before convolution. The next ***for*** loop executes after the convolution ***for*** loop, and shifts all the values stored in the buffers to a higher index, e.g. the current input ***x[0]*** replaces the previous input ***x[1]***, the previous input ***x[1]*** replaces the 2nd past input buffer ***x[2]*** and etc (see figure 3). Therefore, by time the function will be called again at next sampling, all the inputs in the buffers are delayed by one index.

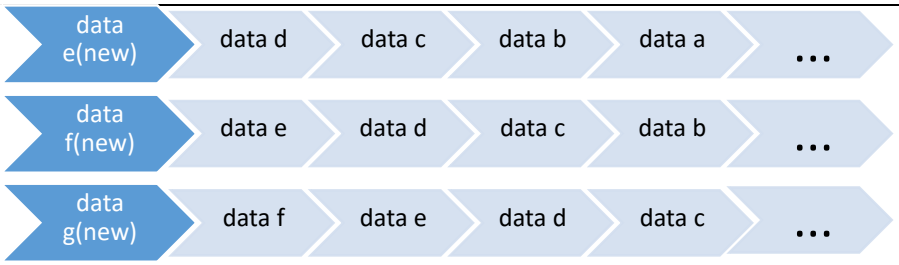


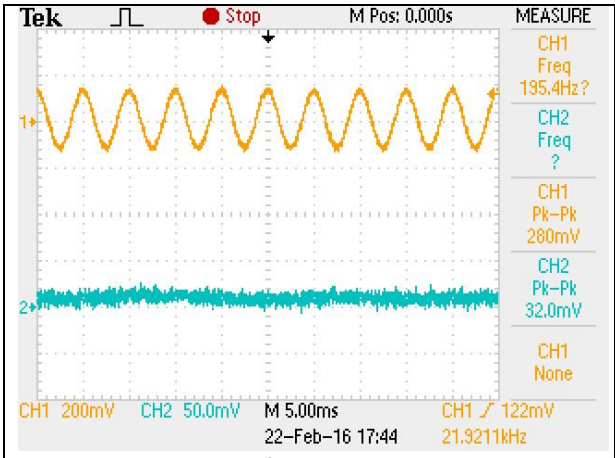
Figure 3: it illustrates how function non_circ_FIR() works. Each element here is a buffer, and each row represents x[]. Data e in the first column is a new input. By the time the function is called again, data f becomes a new input, and all the past data (data e,d,c,b,a and etc) have already been shifted to the right of the new input.

Finally, at the end of the function, ***output*** is returned to the ISR, and the ISR writes it back to the DAC in the DSP kit.

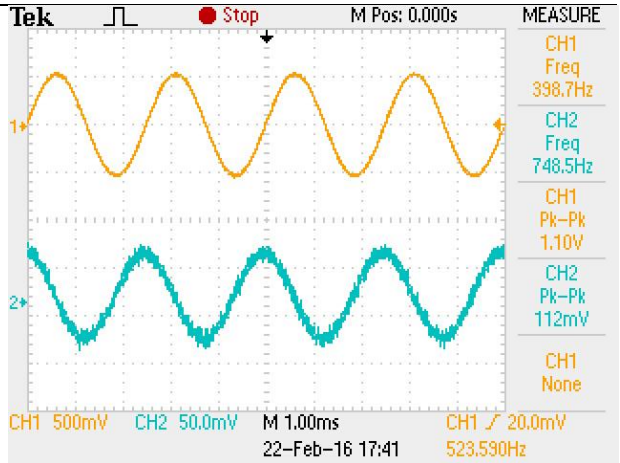
2.1.2. Testing

In this subsection of non-circular buffer implementation, we tested our implementation and provided scope traces below to demonstrate that our design matches our expectation. We expected the output to be noise when the input frequency is at the stop bands (below 375 Hz and above 1750 Hz), to be an attenuated version of the input when the input frequency is at the transition bands (375 to 450 Hz and 1600 to 1750 Hz), and to be the same as the input when the input frequency is at the passband (450 to 1600 Hz). Hence the tests were carried out with input signal frequencies at different frequency bands, such as the 1st stop band (scope trace 1), the 1st transition band (scope trace 2), the pass band (scope trace 3), the 2nd transition band (scope trace 4) and the 2nd stop band (scope trace 5).

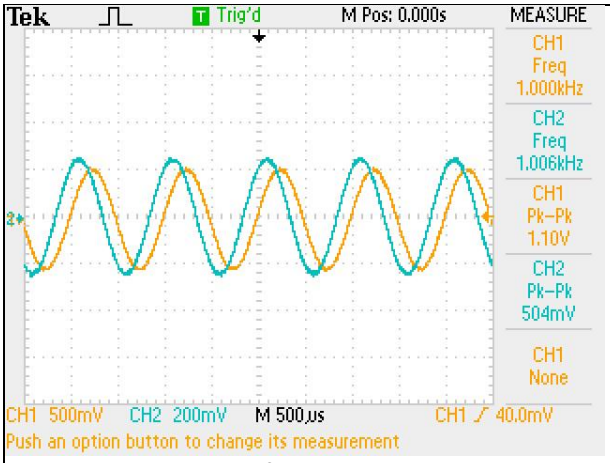
Note: Channel 1 (yellow) and channel 2 (green) display the input and output signals respectively.



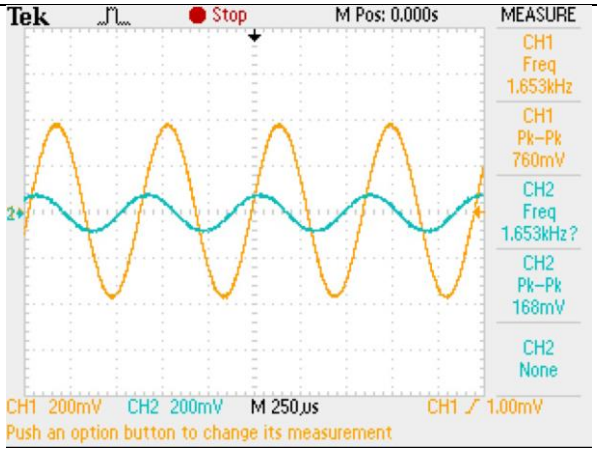
Scope trace 1: input frequency at 200 Hz



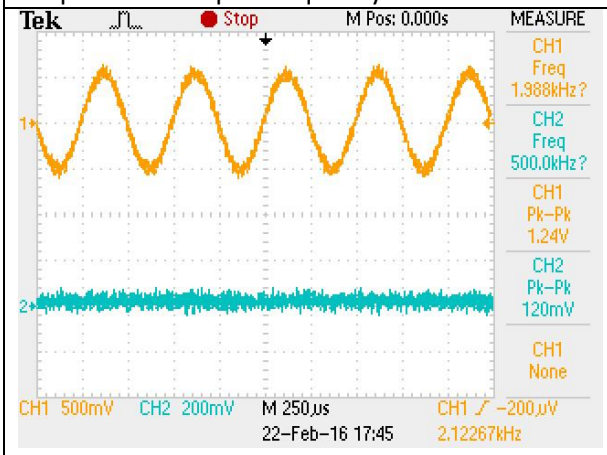
Scope trace 2: input frequency at 400 Hz



Scope trace 3: input frequency at 1000 Hz



Scope trace 4: input frequency at 1650 Hz



Scope trace 5: input frequency at 2000 Hz

From scope trace 1 and 5 (stop band), we observed only noises at the output when the input frequency is within the range of stop bands, which is exactly what we expected. Looking at scope trace 3 (pass band), we found that the pk-pk voltage of the output is about the half as that in input signal. This is because the input signal must firstly pass a potential divider (attenuated by half) before being fed into the input port of the DSP kit. Therefore, the result is what we expected. Moreover, from scope trace 2 and 4 (transition band), we found that the output still have a sinewave shape but is further attenuated comparing to that in scope trace 3 (pass band). This is reasonable since at the transition band the bandpass filter has an attenuation less than one (less than 0 dB) but better than the stop band (-48dB).

Hence, all the scope traces match what we expected from the bandpass filter. We further examined the output using spectrum analyser, which would be discussed in section 3.

Note: we also repeated the above testing process for the following implementations, such as basic circular buffer implementation, in order to make sure the MAC operations were executed correctly and consistently. However, since the testing procedures were the same, they would no longer be discussed in the following sections of the report.

2.1.3. Performance

In this subsection of non-circular buffer implementation, we discussed the performance of our design.

The table below displays the number of instruction cycles for a complete ISR of the basic non-circular buffer filter implementation at three different compiler optimisation levels. For non-optimised code, it takes the greatest number of instruction cycles which is 13299. As optimisation level upgrades, the instruction cycles that the code has to take becomes less and less. With optimisation level 2, the code requires only 1384 cycles, which is about 10% of that of non-optimised code. Therefore, we can see that optimised compiler significantly increases the efficiency of our code by reducing a great number of instruction cycles for every ISR.

We measured the code efficiency to observe its running speed in terms of instruction cycles, which was measured from the beginning of the ISR until its end. The whole process includes evoking convolution function, reading from input, manipulating data and writing the result back to the hardware.

The measurements took place under different compiler optimisation levels, and the results are displayed in the table below. For non-optimised mode, the process costs the greatest number of instruction cycles, 13295. As optimisation level upgrades, the instruction cycles that the code has to take reduces. At optimisation level 2, it requires only 1202 cycles, which is about 9% of that for non-optimised compiler. Therefore, we can see that optimised compiler significantly increases the efficiency of our code by reducing a great number of instruction cycles for every ISR.

Op-level	Number of cycles
none	13295
0	12658
2	1202

Moreover, detailed explanations about these levels of compiler optimisations can be found in section 2.2.4.

2.2. Circular Buffer Filter Implementation

In this section, we firstly realized basic circular buffer implementation. Then we improved our code and reduced the number of instruction cycles to some extent by using the symmetry property of the linear phase filter. After that, we focused on shaping a better data structure as final improvement and made comparisons of benchmark data among all the implementations.

The main idea behind the circular convolution in our design can be described in the following equation:

$$output = \sum_{i=0}^{N-1} b[i] \times x[index + i]_{mod(N)}$$

Where ***index*** is an int variable ranging from 0 to N.

N stands for the number of taps. In this case, N equals to the number of coefficients (the order of filter+1). ***N*** = 214.

x[] are the buffers which stores all the past inputs and the new input.

b[] contain all the coefficients of the filter in double precision. The sizes of ***b[]*** and ***x[]*** are the same.

output is the result of multiply accumulations, which essentially implements circular convolution.

mod(N) means that ***x[index + i]*** is duplicated when overflow occurs. Namely, when ***(index+i)*** exceeds N-1, it restarts from ***0***. We would talk about this in more detail in section 2.2.1.

2.2.1. Basic Case Implementation

Again, the circular convolution function is called in the ISR (shown in code below). The output is returned to the ISR, casted into ***Int16*** and written back to the memory using the command ***`mono_write_16Bit((Int16) (base_circ_FIR()))`***. The code for the ISR function is shown below.

```
/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE**********************/
void ISR_AIC(void){
//write sample back to memory using base circular convolution (no optimisation).
//casting double into 16 bit int data type.
        mono_write_16Bit((Int16) (base_circ_FIR()));
}
```

The function below implements the method using circular buffer, as described by the equation,

$$output = \sum_{i=0}^{N-1} b[i] \times x[index + i]_{mod(N)}$$

.

```
/******************** BASE-CIRCULAR FILTERING CODE CALLED BY A ISR**********************************/
double base_circ_FIR(void){
//this function performs basic circular convolution

  //the two for loops below avoids overflow/underflow of the index of x (index of x cell must lie in the range between 0 and N-1).
  //In this case, index is a global variable which is initially defined as 0. Thus avoids a problem of index being out of range.
  x[index] = (double) mono_read_16Bit();  //read input and store it in a buffer
  output = 0.0; //reset output
  //1st for loop loops from i=0 to i=(N-index-1) which is the point just before the overflow of the index of x. (index+i=N-1)
  for(i=0;i<N-index;i++){
    output += x[index+i]*b[i];
  }
  //This loop handles the overflow of the index of x and continues looping from i=(N-index+1) to i=N-1,
  //which is the end of the coefficient vector.
  for(i=N-index;i<N;i++){
    output += x[index-N+i]*b[i];
  }
  index--;//decrease index, therefore new input will be stored in decending order.
      // this instruction performs circular shift
  if(index == -1){index = N-1;}//note that N stands for the no. of elements in vector b[].
                  // Therefore, the index for the last element is N-1.
  return output;
}
```

Basically, the function does the same things as the previous implementation. It reads input from the DAC using command *mono_read_16Bit()*, stores the casted reading (***double*** precision) into the buffer, and reset ***output*** to 0 at the beginning of the function.

However, there are some changes. You can find the figure 4 below providing an intuitive explanation of how the code works.

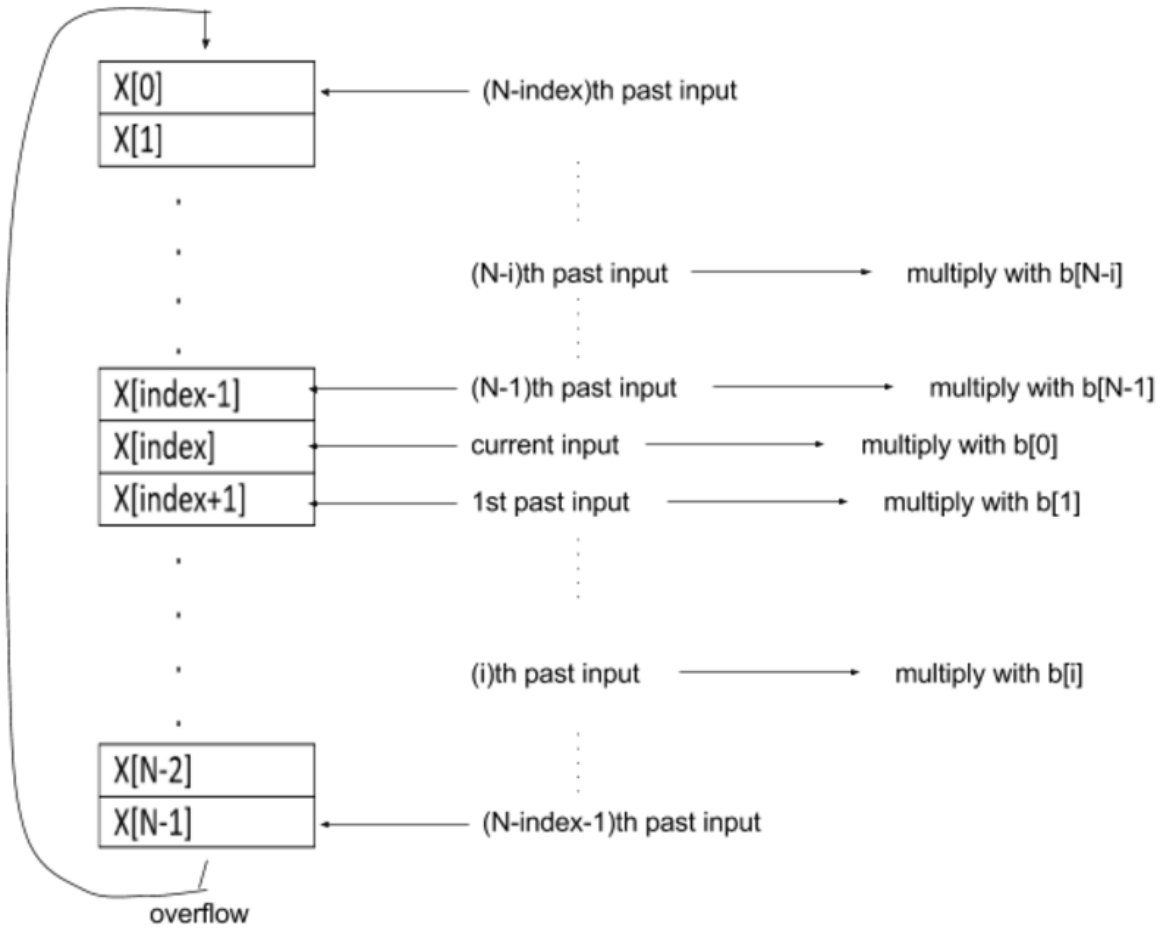


Figure 4: it illustrates how function base_circ_FIR() works. The blocks on the left forms an array of buffers x[]. Arrows pointing to the buffers indicate the types of inputs that are stored in the buffers. E.g. X[index] stores the current input, and X[index+1] stores the 1st past input which means the input before the current input. Therefore, 2nd past input is the input before the 1st past input, and has two sample times delay from the current input. In addition, arrows pointing to the right directions indicate the mapping of inputs and coefficients.

The globally defined ***int*** variable ***Index*** inside ***x*** cell stands for the current index of the buffer which stores the new input. Instead of using method of shifting buffers in non-circular implementation, we varied the value of the globally defined integer ***index*** which was initialised to 0 in descending order (As long as the value of ***index*** is in the range of 0 to N-1, this function always works). In this way, we stored past inputs in the buffers with higher index (if no overflow occurs), and recent inputs in the buffers with smaller index. For example, if ***x[2]*** stores the previous input, then current input is stored in ***x[1]***, and the next input will be placed in ***x[0]***. Thus we avoided complex and inefficient non-circular implementation which shifts all the values stored in the buffers to the larger index by 1. This was implemented by the first ***for*** loop.

In addition, we implemented another ***for*** loop to handle the case when the index of the buffer that stores current input becomes less than 0 (underflow). For example, if ***x[1]*** stores the previous input,

then current input is stored in ***x[0]***, and the next input will be placed in ***x[N-1]*** instead of ***x[-1]***. Namely, when underflow occurs, e.g. ***index*** becomes -1, it will be reset to N-1.

Inside the two ***for*** loops, MAC algorithm is applied. The current and past inputs stored in the buffers ***x[]*** are mapped with corresponding coefficients ***b[].*** For example, if ***index=N-2***, the multiplying accumulation starts from ***x[N-2]*b[0]***. This is because ***x[N-2]*** is the current input and ***b[0]*** is the first coefficient of the filter. Since the previous input was stored in buffer ***x[N-1]***, the next multiply accumulation should be ***x[N-1]*b[1]***. Hence we concluded that the index of buffer should increase by 1 in each for loop until its value exceeds ***N-1*** (overflow). After that, the second ***for*** loop continues the MAC, and stops until the last coefficient of the filter ***b[N-1]*** is reached. Therefore, in the example we just mentioned, the last multiply accumulation should be ***x[N-3]*b[N-1]***.

At the end of the function, it returns the ***output*** to the ISR. Note that ***output*** is reset back to zero every time the function is called.

2.2.2. Improvement 1 - Using Symmetry Property

From Matlab plots (figure 2), we obtained a constant phase change in the pass band (450 Hz to 1600 Hz), thus the filter is a linear phase filter. Every linear phase FIR filter has symmetric coefficients. In this case, the coefficients we obtained from Matlab, which is stored in the vector ***b[]***, are symmetric (see the Matlab plot below, figure 5). Namely, ***b[0]*** has the same value as ***b[N-1]***, ***b[1]*** is the same as ***b[N-2]***, and etc.

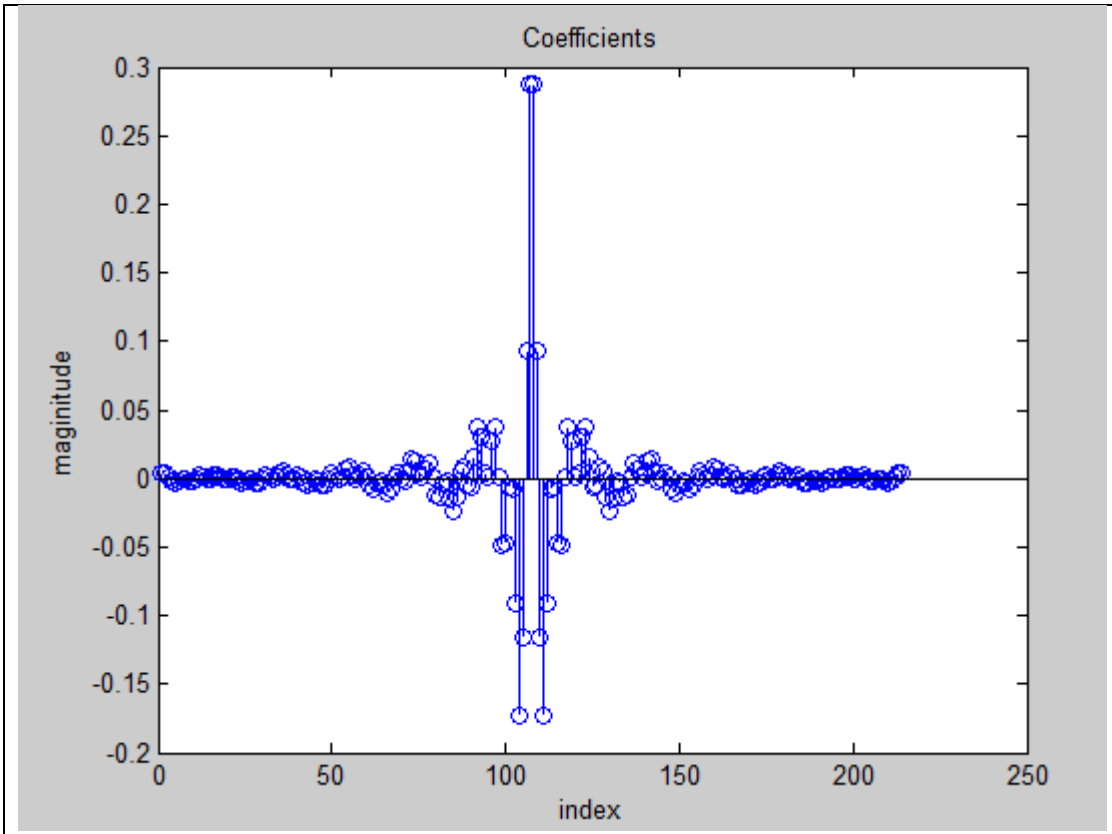


Figure 5: the Matlab plot of the coefficients b[index] where x-axis represents index and the magnitude of b[index].

From the graph we could see that the coefficients are symmetric. Therefore, the equation for convolution can be factorised into the following equation:

$$output = \sum_{i=0}^{\frac{N}{2}-1} b[i] \times \left(x[index + i]_{mod(N)} + x[index - i - 1]_{mod(N)}\right)$$

(N/2)-1 is the index for the last symmetric coefficient. ***N/2*** means the total number of multiply accumulations.

Hence, we halved the number of multiplications (only ***N/2***) comparing to the previous design ***(N)*** by factorisation since the left half coefficients of ***b[]*** is the same as the right half due to symmetry property of the filter. We implemented this method in a function called ***symmetry_cir_FIR(void)***, the code of which is shown below.

```
/******************* IMPROVEMENT ONE**********************************/
double symmetry_circ_FIR(void){
  x[index] = (double) mono_read_16Bit();//read input and store it into the buffer
  output = 0.0; //reset output
  if(index<N/2){//when underflow occurs in the right half of the buffers.
    for(i=0;i<index;i++){ //avoid the attempt that right half of the buffer underflows.
                  //when i=index-1, x[index-i-1]=x[0] is at the margin of being below its range.
      //MAC factorisation
      output += (x[index+i] + x[index-i-1])*b[i];
    }
    for(i=index;i<N/2;i++){//N is even (214), for loop stops when i=(N/2)-1, and convolution completes.
      //MAC factorisation
      output += (x[index+i] + x[index-i+N-1])*b[i];
    }
  }
  else{//when overflow occurs in the left half of the buffers.
    for(i=0;i<N-index;i++){//avoid the attempt that left half of the buffer overflows.
                  //when i=N-index-1, x[index+i]=x[N-1] is at the margin of being below its range.
      //MAC factorisation
      output += (x[index+i] + x[index-i-1])*b[i];
    }
    for(i=N-index;i<N/2;i++){//N is even (214), for loop stops when i=N/2-1, and convolution completes.
          //MAC factorisation
      output += (x[index+i-N] + x[index-i-1])*b[i];
    }
  }
  index--; //index decreases by 1 every time this function is called therefore the past inputs are stored in ascending order.
  if(index==-1){index=N-1;} //hanle underflow
  return output;
}
```

This improved version of code for circular convolution reads and writes input and output in the same manner as in the previous one. However, there are two special cases of overflow/underflow we need to handle. One is that the ***index*** of the buffer which stores current input is less than ***(N-1)/2*** (underflow), and the other is that the ***index*** is greater than ***(N-1)/2*** (overflow). We employed two ***for*** loops as what we did in the previous code to handle overflow or underflow of the index of vector ***x***, e.g. as ***i*** increases***,*** the index of vector ***x***, ***(index-i-1),*** becomes less than 0, which leads to underflow.

However, changes were made in both ***for*** loops. Firstly, the total number loops drops from ***N*** to ***(N-1)/2*** because we factorised our previous algorithm by using symmetry property of the coefficients. A detailed demonstration can be found in the following paragraph. Secondly, we should note we used four ***for*** loops, the first two are implemented to handle underflow of index, and the latter two are to handle overflow. Therefore, inside these ***for*** loops, we made some adjustments of the variables in

the index box, ***x[]*** and ***b[]***. The details of these changes can be found in our code above, and an intuitive explanation of how our code works is displayed in the figure below (figure 6).

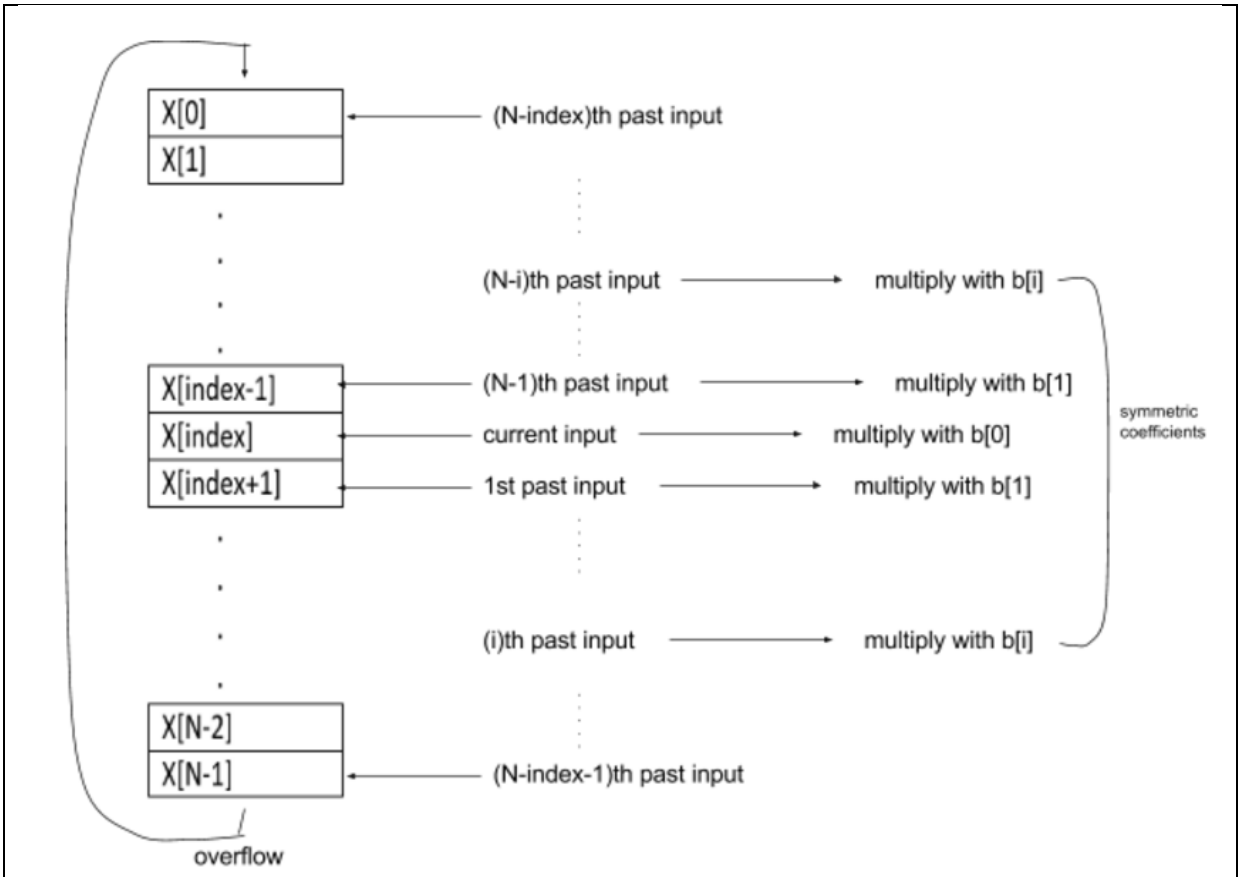


Figure 6: it illustrates how function symmetry_circ_FIR() works and handles overflow when index>N/2. The blocks on the left forms an array of buffers x[]. Arrows pointing to the buffers indicate the types of inputs that are stored in the buffers. E.g. X[index] stores the current input, and X[index+1] stores the 1st past input which is the previous input of the current one. Therefore, 2nd past input is the input before the 1st past input, and has two sample times delay from the current input. In addition, arrows pointing to the right directions indicate the mapping of inputs and coefficients which are symmetric.
Note: the function handles underflow in the same principle when index<N/2.

Moreover, we provided a table (figure 8) to assist our demonstration, which gave an example showing how using symmetry property effectively factorise and halve the number of multiply accumulations.

index=0\i	0	1	2	3
x[index+i](mod N)	x[0]	x[1]	x[2]	x[3]
x[index-i-1](mod N)	x[7]	x[6]	x[5]	x[4]
multiply	(x[0]+x[7])*b[0]	(x[1]+x[6])*b[1]	(x[2]+x[5])*b[2]	(x[3]+x[4])*b[3]
output	(x[0]+x[7])*b[0] + (x[1]+x[6])*b[1] + (x[2]+x[5])*b[2] + (x[3]+x[4])*b[3]			

Figure 8: an example of how symmetry property of coefficients effectively factorises and halves the number of loops for MAC. The number of coefficients N for this linear phase digital filter is chosen to be 8 in this example, which only requires 4 loops of multiplication instead of 8 to complete the circular convolution.

2.2.3. Improvement 2 - Doublesize the Buffer While Using Symmetry

The code below is our updated fastest implementation.

```
/******************** IMPROVEMENT TWO ***********************************/
double X_doublesize_symmetry_circ_FIR(void){
  //this function uses x[2*N] instead of x[N]

  //read input and store it into a buffer
  x[index] = (double)mono_read_16Bit(); //store input in its first index
  x[index2] = x[index]; //store the input in its second index
                //this is more efficient than reading input twice
  output = 0.0; //reset output
  //loop from first coefficient to the (N/2-1)th coefficient
  for(i=0;i<N/2;i++){
    //factorised convolution using symmetry properties
    output += b[i]*(x[index+i]+x[index2-i-1]);
  }
  index--; //decrease index
  if (index==-1){index=N-1;} //handle underflow
  index2 = index+N; //make index2 follow index
  return output; //return output value.
}
```

In the previous version, we had to handle overflow and underflow with certain amount of branching and operations. Two ***for*** loops were used to avoid overflow/underflow of the index of cell ***x***. In order to increase efficiency, we implemented an extended data structure for data fetching to avoid overflow or underflow. In addition, the code no longer requires to determine whether where will be an overflow or an underflow case. Thus, only one ***for*** loop will be needed, and no ***if*** statement will be required.

The idea here was to have two reference index by doubling the ***x[]*** array memory. We designed the first reference called ***index*** within the range from 0 to ***N-1*** and the other one called ***index2*** which is a matched pair with ***N*** offset, such that ***index2=index+N***, and ***x[index]=x[index2]***. In this way, our buffer memory structure was doubled and we were only interested in the data set between ***index*** and ***index2***, which stores data from the current input to the ***212^{th}*** past inputs since ***N=214***. Our code fetches these data and does MAC using a single ***for*** loop while taking advantage from symmetry property of the coefficients. An intuitive explanation of how our code works is shown in the figure below (figure 9).

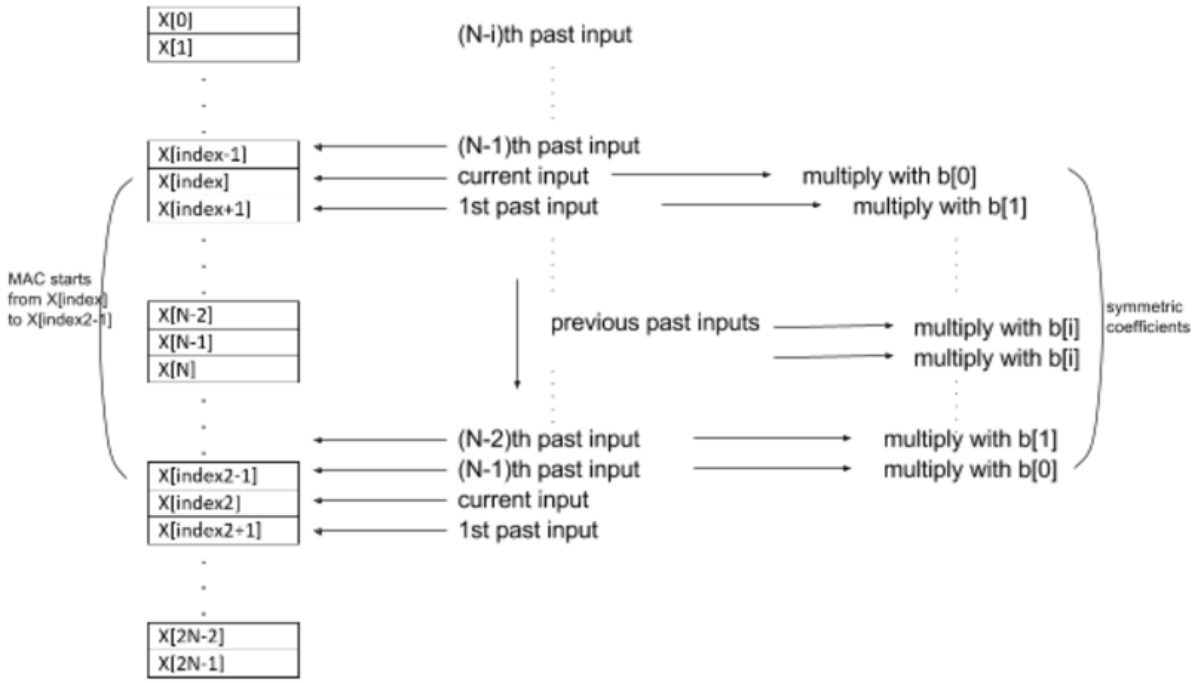


Figure 9: it illustrates how function doublesize_symmetry_circ_FIR() works and handles overflow. The blocks on the left forms an array of buffers x[] of size 2N. Basically, the buffer array starting from x[index] to x[index2-1] is the same as the combined buffer array of the two subarrays, which are formed by buffers from x[index2] to x[2N-1] and from x[0] to x[index-1]. Arrows pointing to the buffers indicate the types of inputs that are stored in the buffers. E.g. X[index] stores the current input, and X[index+1] stores the 1st past input which is the previous input of the current one. Therefore, 2nd past input is the input before the 1st past input, and has two sample times delay from the current input. In addition, arrows pointing to the right directions indicate the mapping of inputs and coefficients which are symmetric.

2.2.4. Benchmark Data and Comparisons among different methods

In this subsection, we ran all the implementations at different compiler optimisation levels and recorded the minimum time taken in terms of instruction cycles for an entire ISR function (shown in the table below).

Op-level	Non-Circular Buffer	Basic Circular Buffer	Circular Buffer-Improvement one	Circular Buffer-Improvement Two
none	13295	10366	5830	5710
0	12658	9727	6050	5403
2	1202	1798	1063	705

Hence we plotted a graph according to our findings (figure 10).

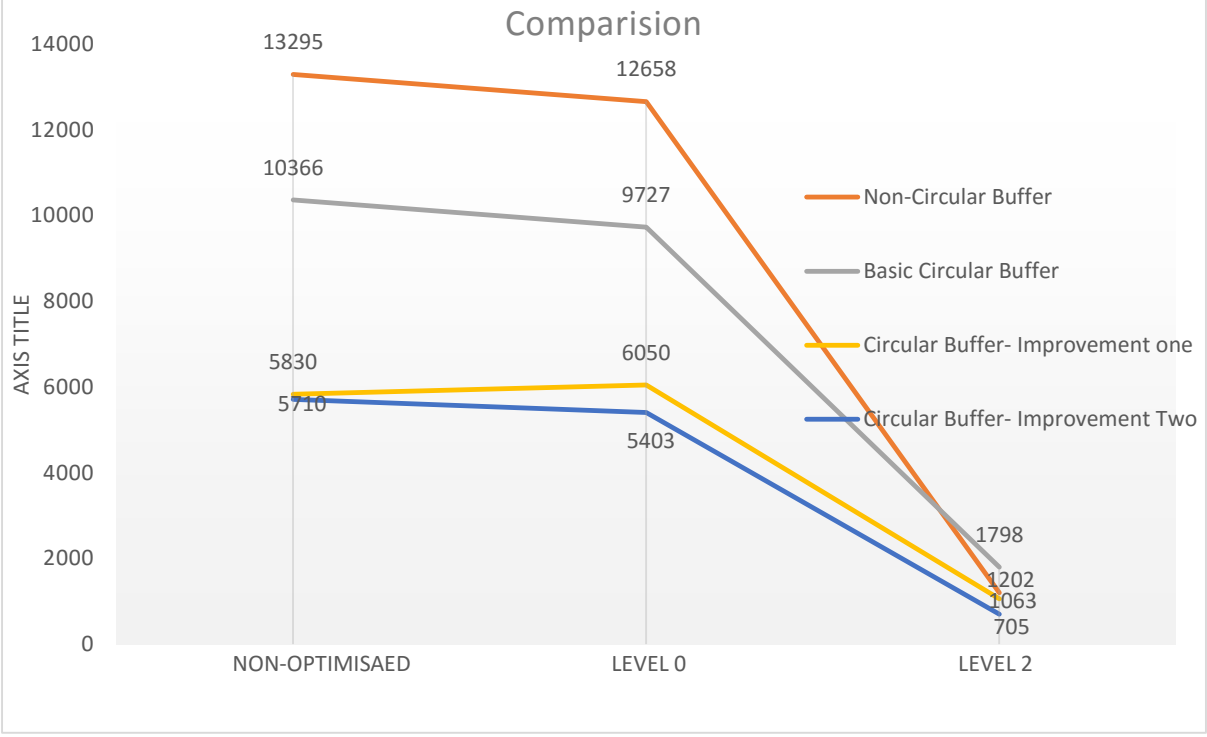


Figure 10: comparison among different implementations at different compiler optimisation levels. Clock cycles are counted for the entire ISR function.

Level 0 includes only register level optimisation. Level 1 does all level 0 optimisations and adds local optimisation. Based on the features of level 1, level 2 adds global optimisation and applies software pipelining, loop optimisation and unrolling loops.

These characteristics of different optimisation levels explain why there are only a few reductions in cycles from non-optimised to optimisation level 0 but significant from non-optimised to level 2.

In RTDSP clock efficiency is the main concern, thus our best design is the second improvement version of circular buffer, which applies doubling buffer size and symmetry property at the same time.

3. Analysis of the Frequency Response

In this lab, all the methods were tested using spectrum analyser (515 audio analyser) and their frequency responses were exactly the same, which implies that all the MAC operations were done correctly and there was no mismatch or leakage of input buffers and coefficients.

In this subsection, we provided both gain and phase plots of one of our method (all of them produced the same responses), and discussed the results with a comparison to our original FIR filter response generated by Matlab.

3.1. Gain

In this subsection, we aimed to investigate the gain plot of our design from the spectrum analyser and compare it with the ideal one generated by Matlab.

We investigated the gain plot of our design at different frequency bands, such as stop bands and pass band, and checked their correctness. From figure 11 and 12, we found that the stop band attenuations are over 48 dB (49.812 and 49.593 in this case) as required by the design specification.

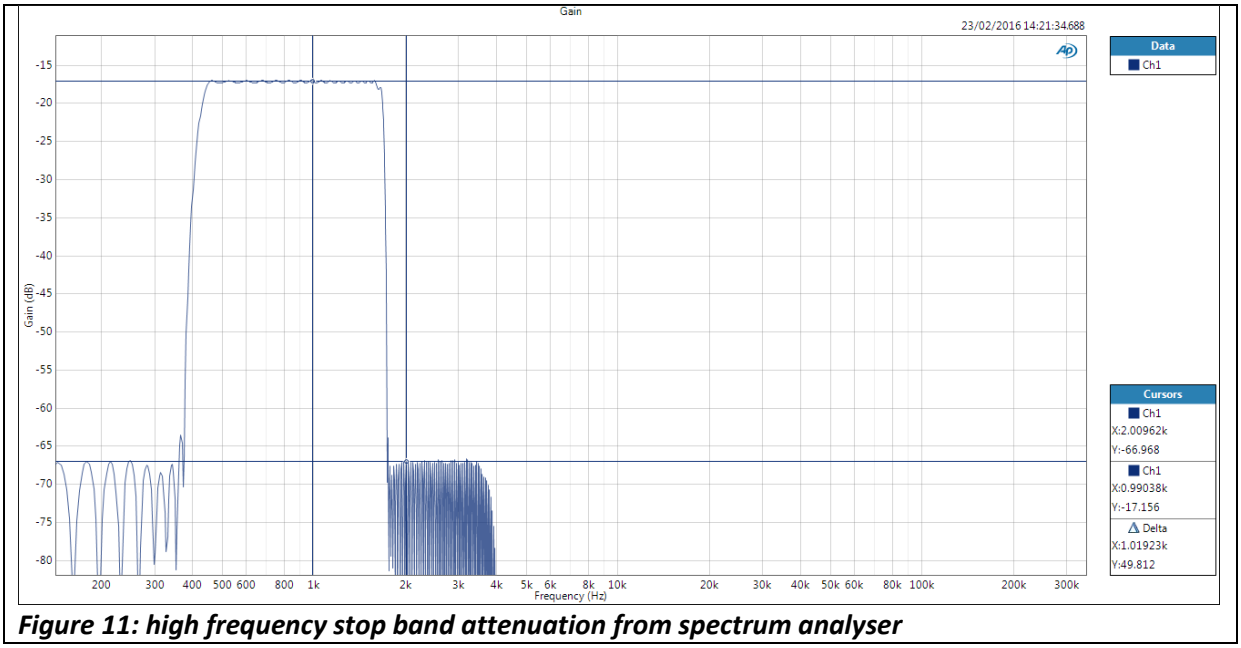


Figure 11: high frequency stop band attenuation from spectrum analyser

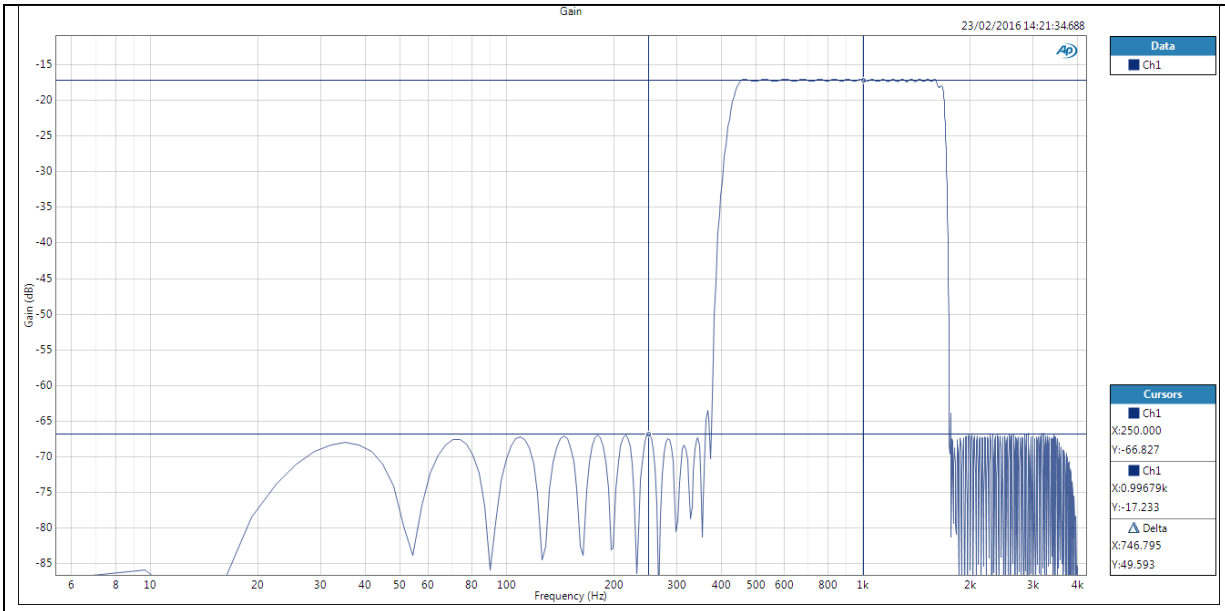


Figure 12: low frequency stop band attenuation from spectrum analyser

Secondly, we checked the passband deviation. From figure 13, we found that the passband deviation matches the design specification of the ideal filter.

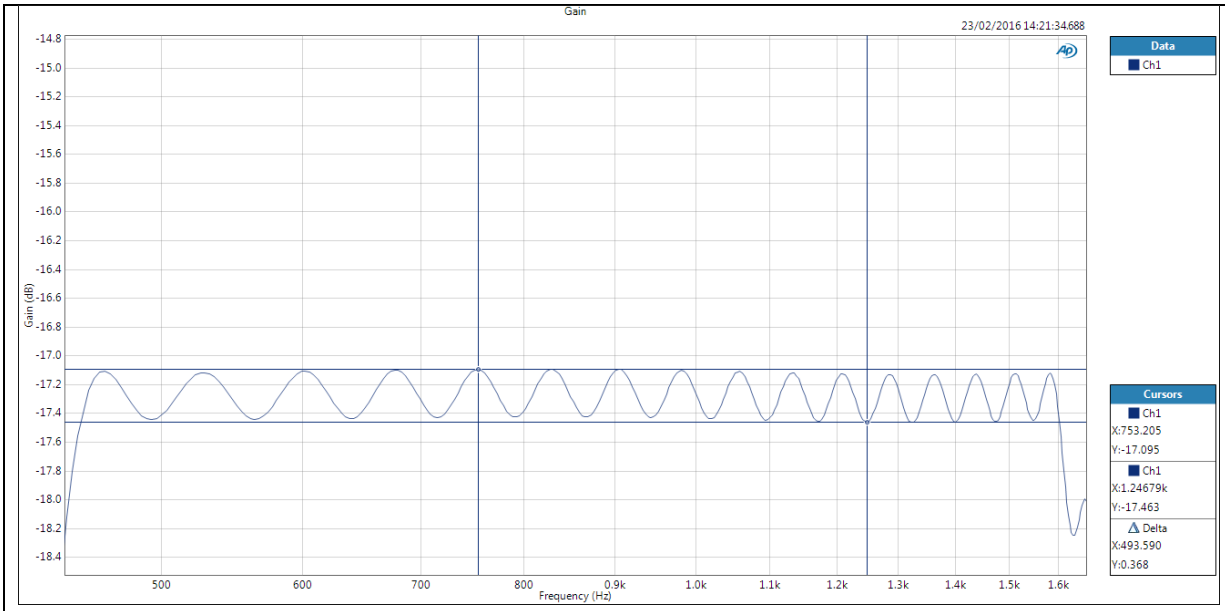


Figure 13: pass band deviation from spectrum analyser

We noticed that the gain plot is attenuated by approximately 17dB. This is due to the fact that the audio analyser we used (515 audio analyser) has ¼ attenuation and internal offsets around -5dB. Therefore, in order to overcome the offset and attenuation from the device to make a fair comparison, we imported the data of gain plot relative to 1000 Hz (inside pass band). Thus, the comparison was fair, which is shown below (see figure 14).

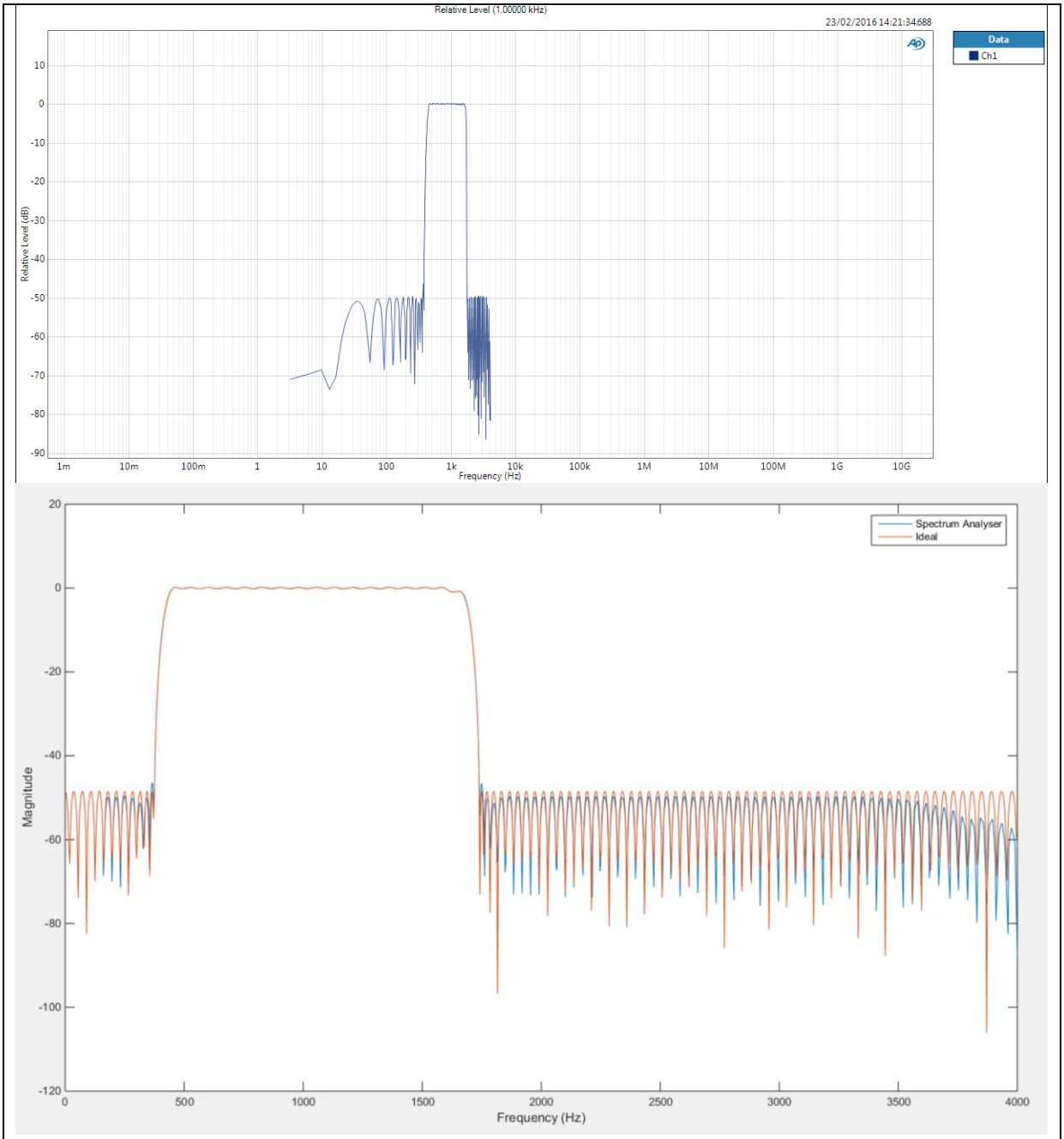


Figure 14: the upper graph is gain response relative to 1000 Hz and the lower graph is a comparison between what we got from spectrum analyser (blue) and the ideal response (orange).

From the comparison, we found that the resulting gain plot nearly perfectly overlaps with the ideal one except at high frequencies above 3500 Hz. This is due to the fact that all the input signals must pass an anti-aliasing filter before entering the input port of the DSP kit. The anti-aliasing filter equals to a non-ideal low pass filter at Nyquist frequency, 4000 Hz, since the sampling frequency is 8000 Hz. However, it is not ideal, namely it has a relatively wide transition band, which starts from 3500 Hz and ends around 4200 Hz. Therefore, it's what we expected that the resulting gain plot is further attenuated at frequencies above 3500 Hz.

However, we observed a little mismatch of the measurement around the stop band edge of the 1st and the 2nd transition band where one of the lobes due to a zero shows less gain attenuation in

response comparing to Matlab. The reason is that the numbers and voltage levels involved at such attenuation levels are small, so we can expect noise to play a part in this.

3.2. Phase

The ideal design is a band pass filter, which should be have linear phase throught the passband. This is very important in audio processing because human ears are very sensitive to phase distortion. Therefore, we required the filter's phase to be linear in the pass band, although it could be non-linear in the stop band, as we are attenuating those frequency anyway. Thus, we aimed to check whether our design behaves linearly in passband. We obtained phase (figure 15) and group delay (figure 16) plots.

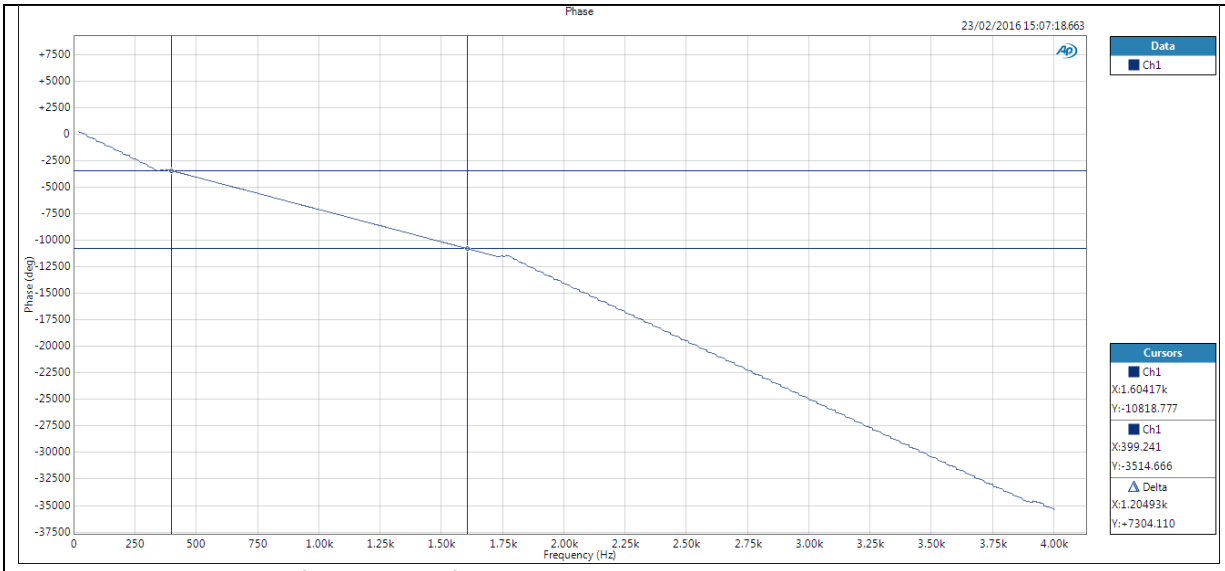


Figure 15: phase plot of our design from spectrum analyser

It seems that the phase behaves linearly from 450 to 1600 Hz, which is the pass band range. However, we couldn't be so confident to conclude its linearity simply from observation. Therefore, we further investigated the group delay, which is shown below in figure 16.

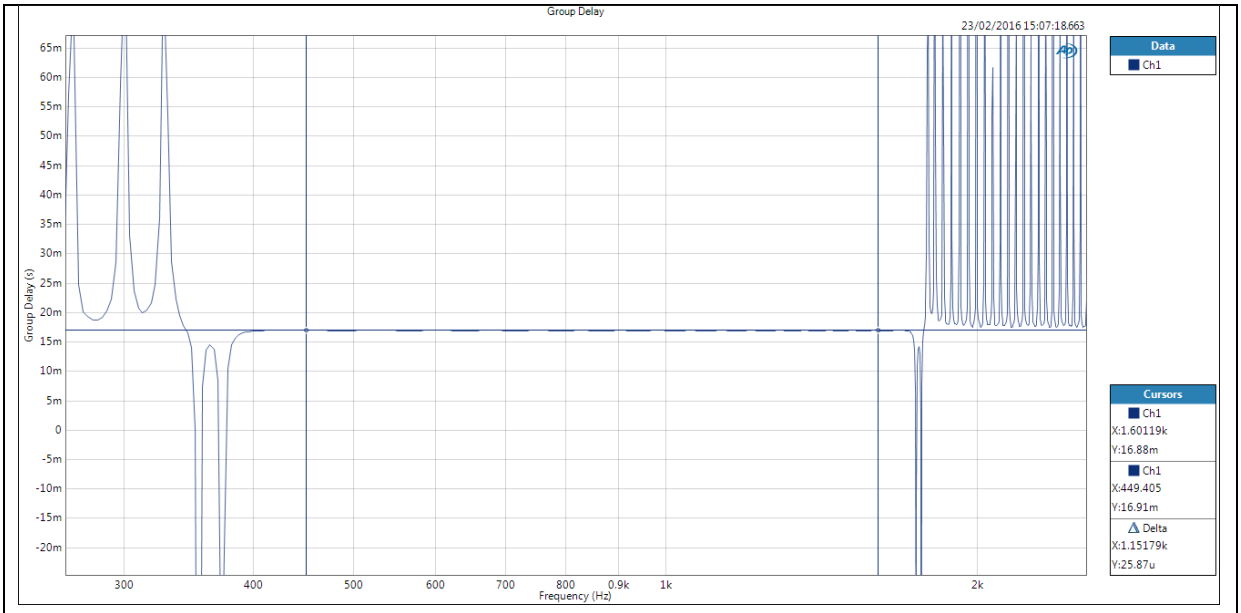


Figure 16: group delay

We only cared about passband. Thus, we placed two cursors at 450 Hz and 1600 Hz respectively. Therefore, the area between the cursors indicates pass band. We found that within the pass band, the group delay is constant around 16.9ms, which are what we expected since the ideal bandpass filter should have linear phase (a constant group delay) in the passband.

The group delay of ideal design can be expressed by the equation below:

$groudelay = \frac{N-1}{2} * \frac{1}{fs} = \frac{214-1}{2} * \frac{1}{8000} = 13.3ms$, where (N-1) is the filter order and *f*s is the sampling frequency.

We found that the both the group delays are constant but the one measured from audio analyser is larger than that of ideal design. This is what we expected because the DSP board has some delay due to the ADC/DAC, which can be considered as an offset to group delays.

Therefore, we measured the bare board group delay, which is expected to be 3.6ms and is shown in the figure 17 below.

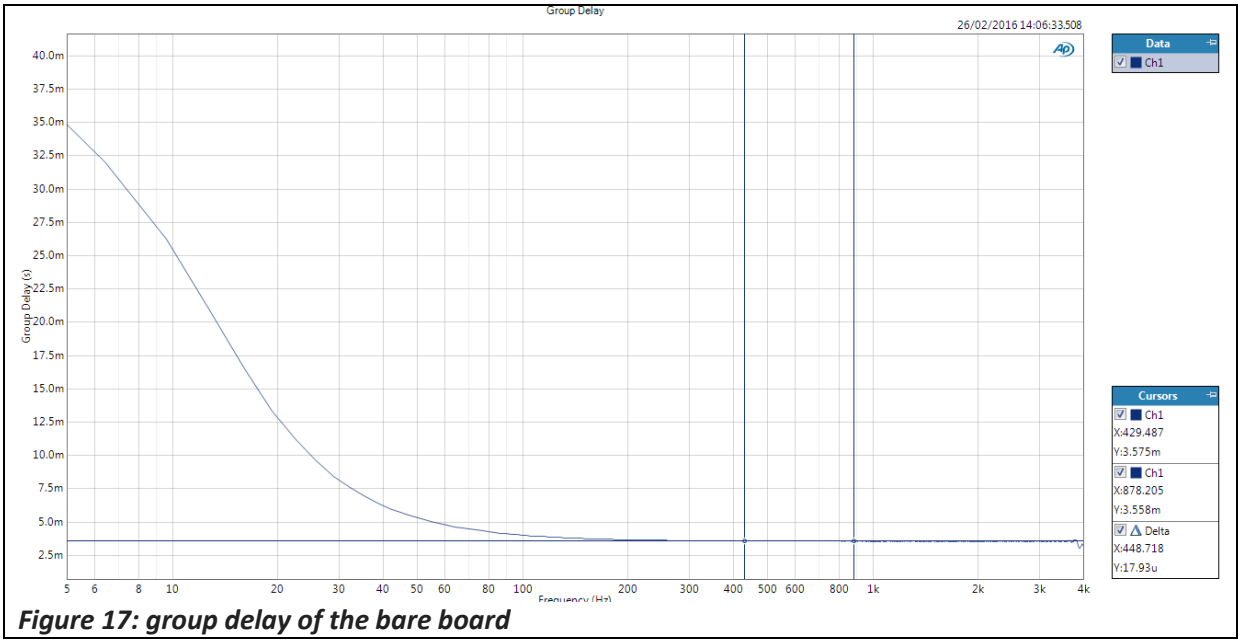


Figure 17: group delay of the bare board

From figure 17, we found that the group delay of the bare board was 3.56ms, which is approximately equivalent to our expectation, 3.6ms. Hence, we verified that the group delay for the passband is what we expect, and the offset is caused by the ADC/DAC in the board.

In conclusion, our design has linear phase in the passband.

4. Appendix

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

//read the text file generated by Matlab into our program,
//which contains the coefficient of the FIR filter in an array b[] and defines the number of coefficients as N.
#include "file_b.txt"

/********************************* 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);
double base_circ_FIR(void);
double symmetry_circ_FIR(void);
double X_doublesize_symmetry_circ_FIR(void);

/******************************* Global Variables **********************************/
/* Sampling frequency in HZ. Must only be set to 8000, 16000, 24000
32000, 44100 (CD standard), 48000 or 96000  */
int sampling_freq = 8000;
/* Use this variable in your code to set the frequency of your sine wave
  be carefull that you do not set it above the current nyquist frequency! */
float sine_freq = 1000.0;
// set the sample as global variable such that its value can be observed easily in the watch table
Int16 samp;
//double x[N];//comment this line of code when using the fastest design, which is implemented by function 'doublesize_circ_FIR()'.
double x[2*N];//uncomment to use the fastest design
int i;
double output = 0.0;
int index = 0;
int index2 = N;//index2 will only be used in the fastest design, which is function 'doublesize_circ_FIR()'.

/********************************** 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
   //it resets all the hardware to the default settings
   DSK6713_init();

   // Start the AIC23 codec using the settings defined above in config
   //set the sampling frequency to 8K and the bit resolution to 16 bits
   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){
  //mono_write_16Bit((Int16) (non_circ_FIR()));//write sample back to memory using non-circular convolution
                                //casting double into 16 bit int data type.
  //mono_write_16Bit((Int16) (base_circ_FIR()));//write sample back to memory using base circular convolution (no optimisation).
                                 //casting double into 16 bit int data type.
  //mono_write_16Bit((Int16) (symmetry_circ_FIR()));//write sample back to memory using symmetry property of circular convolution
                                   //casting double into 16 bit int data type.
  mono_write_16Bit((Int16) (doublesize_circ_FIR()));//write sample back to memory using double memory to implement circular
convolution
                                   //casting double into 16 bit int data type.
                                   //this is the fastest design we implemented.
}

/******************** NON-CIRCULAR FILTERING CODE CALLED BY A ISR**********************************/
double non_circ_FIR(void){
//this function performs non-circular convolution
  samp = mono_read_16Bit();//read input and place in a 16 bit integer memory
  output = 0.0; //reset output to 0
  for(i=N-1; i>0; i--)
  {
    x[i] = x[i-1]; // move data along buffer from lower element to next higher
  }
  x[0] = (double) samp;// convert the new sample into double data type and put it into the buffer
  for(i=0; i<N; i++){  // convolution
    output += x[i]*b[i];// multiply accumulation
  }
  return output;
}
/******************** BASE-CIRCULAR FILTERING CODE CALLED BY A ISR**********************************/
double base_circ_FIR(void){
//this function performs basic circular convolution

  //the two for loops below avoids overflow/underflow of the index of x (index of x cell must lie in the range between 0 and N-1).
  //In this case, index is a global variable which is initially defined as 0. Thus avoids a problem of index being out of range.
  x[index] = (double) mono_read_16Bit();   //read input and store it in a buffer
  output = 0.0; //reset output
  //1st for loop loops from i=0 to i=(N-index-1) which is the point just before the overflow of the index of x. (index+i=N-1)
  for(i=0;i<N-index;i++){
    output += x[index+i]*b[i];
  }
  //This loop handles the overflow of the index of x and continues looping from i=(N-index+1) to i=N-1,
  //which is the end of the coefficient vector.
  for(i=N-index;i<N;i++){
    output += x[index-N+i]*b[i];
  }
  index--;//decrease index, therefore new input will be stored in decending order.
        // this instruction performs circular shift
  if(index == -1){index = N-1;}//note that N stands for the no. of elements in vector b[].
                      // Therefore, the index for the last element is N-1.
  return output;
}
```

```
/******************** IMPROVEMENT ONE**********************************/
double symmetry_circ_FIR(void){
  x[index] = (double) mono_read_16Bit();//read input and store it into the buffer
  output = 0.0; //reset output
  if(index<N/2){//when underflow occurs in the right half of the buffers.
    for(i=0;i<index;i++){ //avoid the attempt that right half of the buffer underflows.
                  //when i=index-1, x[index-i-1]=x[0] is at the margin of being below its range.
      //MAC factorisation
      output += (x[index+i] + x[index-i-1])*b[i];
    }
    for(i=index;i<N/2;i++){//N is even (214), for loop stops when i=(N/2)-1, and convolution completes.
      //MAC factorisation
      output += (x[index+i] + x[index-i+N-1])*b[i];
    }
  }
  else{//when overflow occurs in the left half of the buffers.
    for(i=0;i<N-index;i++){//avoid the attempt that left half of the buffer overflows.
                  //when i=N-index-1, x[index+i]=x[N-1] is at the margin of being below its range.
      //MAC factorisation
      output += (x[index+i] + x[index-i-1])*b[i];
    }
    for(i=N-index;i<N/2;i++){//N is even (214), for loop stops when i=N/2-1, and convolution completes.
          //MAC factorisation
      output += (x[index+i-N] + x[index-i-1])*b[i];
    }
  }
  index--; //index decreases by 1 every time this function is called therefore the past inputs are stored in ascending order.
  if(index==-1){index=N-1;} //hanle underflow
  return output;
}
/******************** IMPROVEMENT TWO **********************************/
double X_doublesize_symmetry_circ_FIR(void){
  //this function uses x[2*N] instead of x[N]

  //read input and store it into a buffer
  x[index] = (double)mono_read_16Bit(); //store input in its first index
  x[index2] = x[index]; //store the input in its second index
              //this is more efficient than reading input twice
  output = 0.0; //reset output
  //loop from first coefficient to the (N/2-1)th coefficient
  for(i=0;i<N/2;i++){
    //factorised convolution using symmetry properties
    output += b[i]*(x[index+i]+x[index2-i-1]);
  }
  index--; //decrease index
  if (index==-1){index=N-1;} //handle underflow
  index2 = index+N; //make index2 follow index
  return output; //return output value.
}
```