Real-Time Digital Signal Processing : Lab 4 - Real-time Implementation of FIR Filters

Jiyu Fang CID: 01054797 Deland Liu CID: 01066080

February 2018

1 Objectives

- Learn to design FIR filters using Matlab.
- Implement the FIR filters using the C6713 DSK system in real-time in C and assembly
- Faster implementation using the concept of circular buffer
- Measure the filter response using a scope and a spectrum analyzer

2 Equipment

- Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator(DSK6713)
- Oscilloscope
- Software signal generator
- APX520 audio analyzer

3 Introduction

The lab involves first designing the FIR filter in Matlab using the Parks-McClelland algorithm, plotting frequency response and checking if specifications satisfied. Extracted coefficients are saved to fir_coef.txt and included in C application. Different versions of filter implementation in C are explored and filter characteristics are measured using both oscilloscope and APX520 audio analyzer.

4 FIR filter

The general form of a discrete-time LTI filter transfer function is:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}}.$$

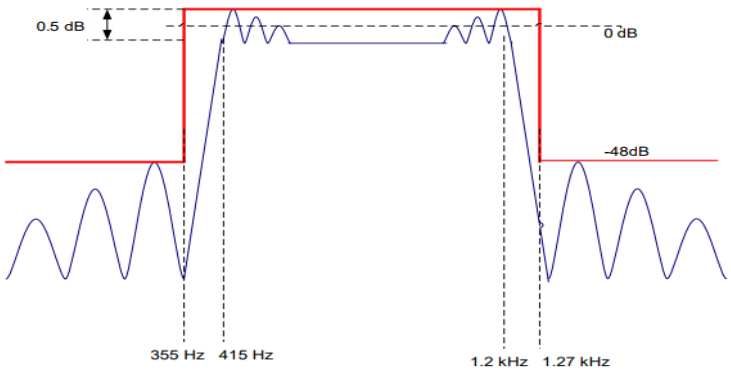
A FIR filter has all the denominator coefficients a as zero, which also means the FIR filters only have poles at zero and they are inherently stable since there are no poles outside the unit circle in the z-plane:

$$H(z) = b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}$$

The difference equation is obtained from inverse z-transform of the transfer function. An analysis of difference equation indicates that current output sample is computed as a weighted sum of current input and previous M inputs. There is no feedback from output to input:

$$y(n) = b_0 x(n) + b_1 x(n-1) + \cdots + b_M x(n-M)$$

5 Specification



$fs = 8kHz$
Pass band edges = $\{355Hz\ 415Hz\}$
Stop band edges = $\{1.2kHz\ 1.27kHz\}$
Maximum pass band ripple(peak-to-peak) =$0.5dB$
Minimum stop band attenuation = $48dB$

6 Filter Design

Given the above frequency response specification, design of filter is implemented on Matlab using the $firpm$ and $firpmord$ built-in functions since we could directly specify all the important filter parameters. The $firpmord$ returns order, normalized frequency band edges, frequency band amplitudes, and weights that satisfy specifications in cut-off frequencies(f), amplitudes(a), pass band ripple and stop band attenuation(dev).
$firpm$ function returns coefficients of linear phase digital filter to best meet specifications with minimum order.
To work out deviations (DEV), note that the maximum allowed gain value in pass band in dB is 20log(1+DEV) and minimum is given by 20log(1-DEV), given that pass band gain is 1(0dB). The difference the two is the peak-to-peak pass band ripple in dBs.
As shown in difference equation, coefficients of a FIR filter are also the impulse response. A linear phase FIR filter has real and symmetric coefficients($h(n) = h(M - n)$ for a filter of order M and do not introduce any phase distortion. This indicates that group delay is constant, and only delay is introduced for signals that only contain frequencies components inside the pass band. The Matlab functions employ an equiripple design method, hence ripples throughout pass band are almost equal.
Plot the frequency response in Matlab using $freqz$ function. By putting the data cursor on the points of interest we noticed that the attenuation specifications are not satisfied at $f = 355Hz$ and $f = 1.27kHz$, since the magnitude is above -48dB. Also, the maximum peak-to-peak ripple exceed 0.5dB, as shown by figures below:

```
rp = 0.5; %pass band ripple(peak-to-peak)
sg = 48; % min stop band attenuation
fs = 8000; %sampling frequency
f = [355 415 1200 1270]; %cut-off frequencies
a = [0 1 0]; %desired gain(not in dB)
Dstop = 10^(-sg/20); %max gain in stop band
Dpass = (10^(rp/20)-1)/(10^(rp/20)+1);%max allowed pass band gain deviation
dev = [Dstop Dpass Dstop];%vector of maximum gain deviations allowable for each band(not in dB)
[n,fo,ao,w] = firpmord(f,a,dev,fs); %[appximate filter order,normalised frequency band edges, frequency band amplitudes, weigh
b = firpm(n,fo,ao,w); %filter coefficients
freqz(b,1,1024,fs) %plot frequency response
title('FIR Filter')
%save fir_coef.txt b -ascii  -double -tabs; %save to fir_coef.txt
fileID = fopen('fir_coef.txt','w');
formSpec1 = 'double b[]={%.16e';
fprintf(fileID,formSpec1,b(1));
formSpec2 = ', %.16e';
fprintf(fileID,formSpec2,b(2:end));
fprintf(fileID,'};\n');
fclose(fileID);
```

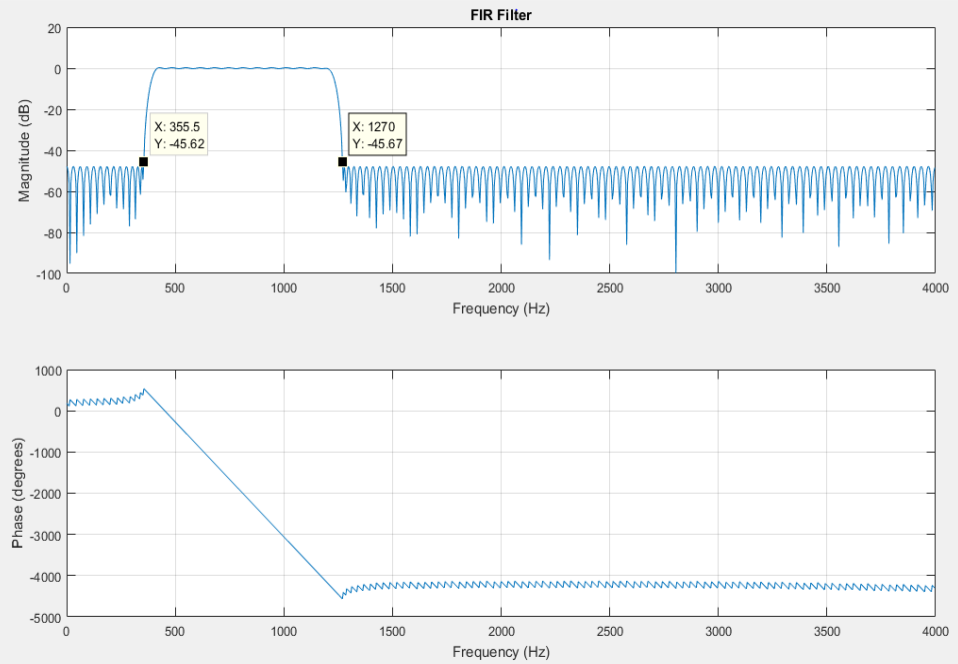
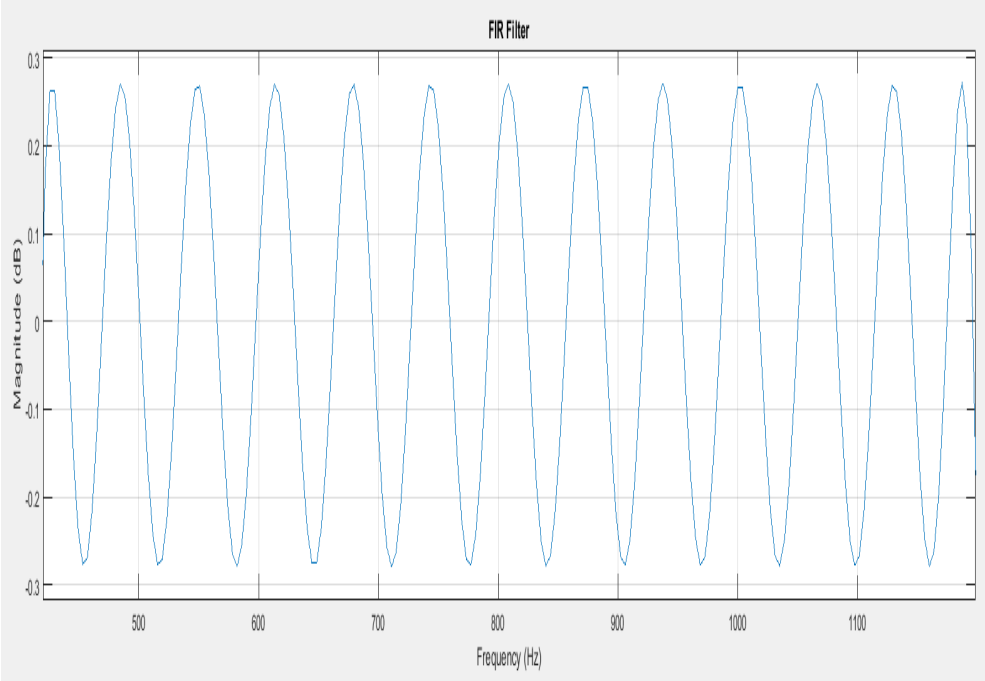


Figure 1: Plot of filter pass band ripple with order 248 exceeding requirement of 0.5dB ripple



By narrowing the transition band and hence increasing the filter order, a filter with order 255 is designed by Matlab that satisfies all specifications.

```
rp = 0.5; %pass band ripple(peak-to-peak)
sg = 48; % min stop band attenuation
fs = 8000; %sampling frequency
f = [356.75 415 1200 1268]; %cut-off frequencies
a = [0 1 0]; %desired gain(not in dB)
Dstop = 10^(-sg/20); %max gain in stop band
Dpass = (10^(rp/20)-1)/(10^(rp/20)+1);%max allowed pass band gain deviation
dev = [Dstop Dpass Dstop];%vector of maximum gain deviations allowable for each band(not in dB)
[n,fo,ao,w] = firpmord(f,a,dev,fs); %[appximate filter order,normalised frequency band edges, frequency band amplitudes, weigh
b = firpm(n,fo,ao,w); %filter coefficients
freqz(b,1,1024,fs) %plot frequency response
title('FIR Filter')
%save fir_coef.txt b -ascii  -double -tabs; %save to fir_coef.txt
fileID = fopen('fir_coef.txt','w');
formSpec1 = 'double b[]={%.16e';
fprintf(fileID,formSpec1,b(1));
formSpec2 = ', %.16e';
fprintf(fileID,formSpec2,b(2:end));
fprintf(fileID,'};\n');
fclose(fileID);
```

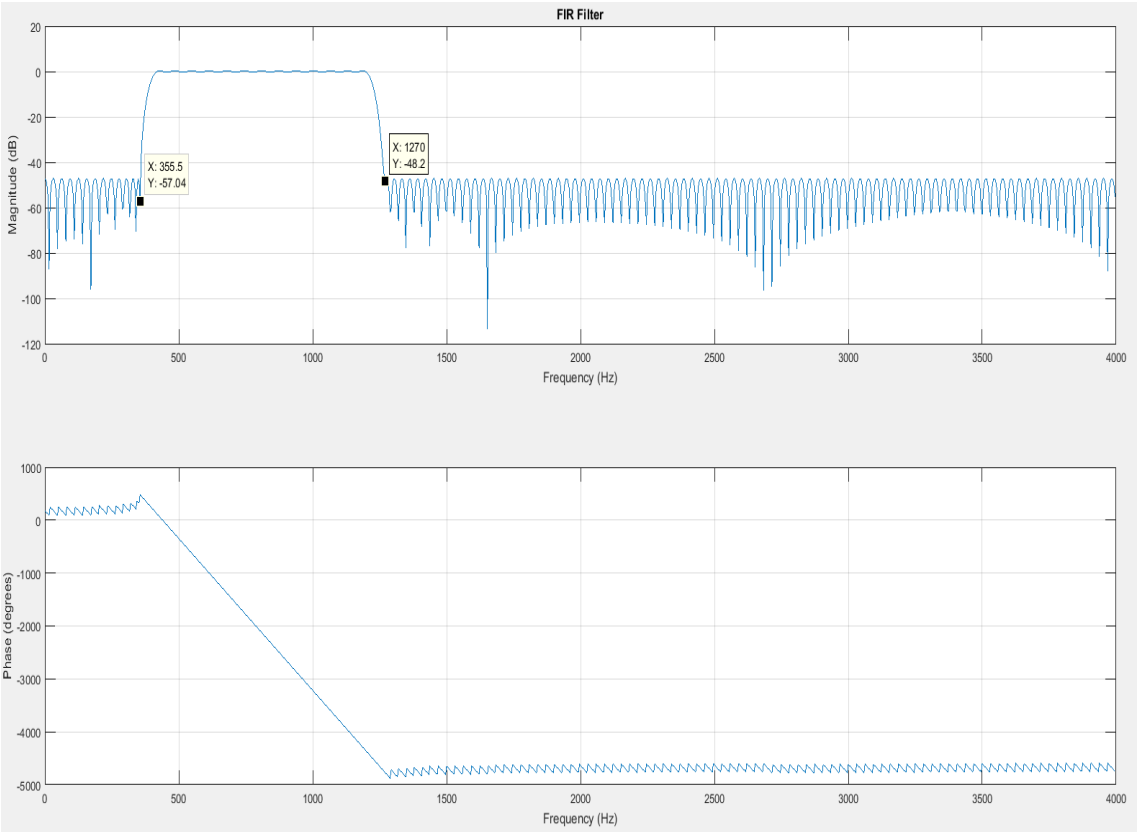
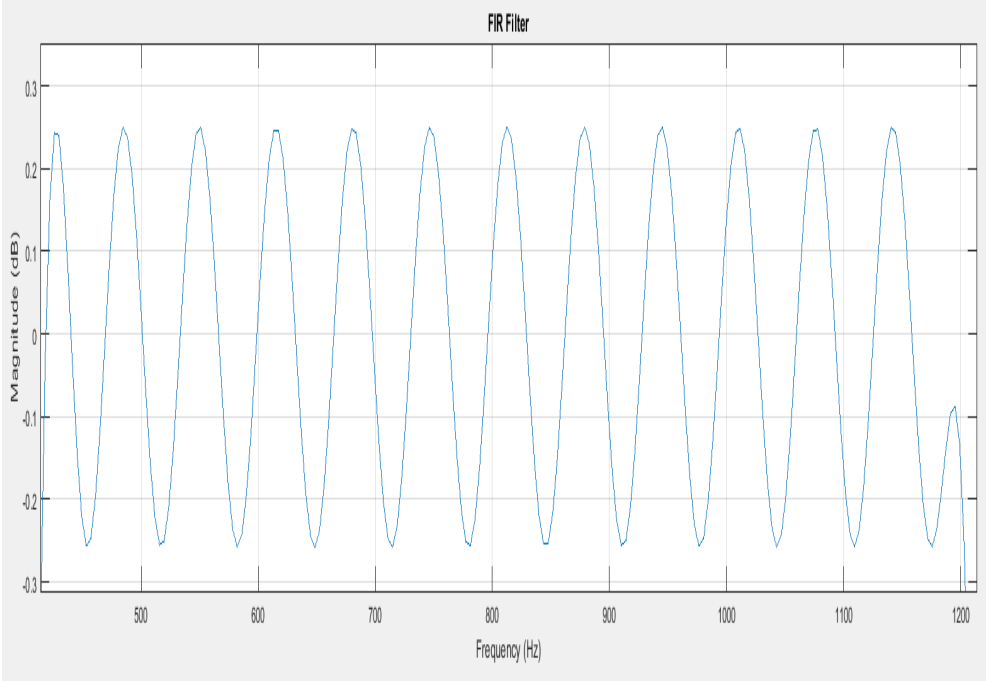


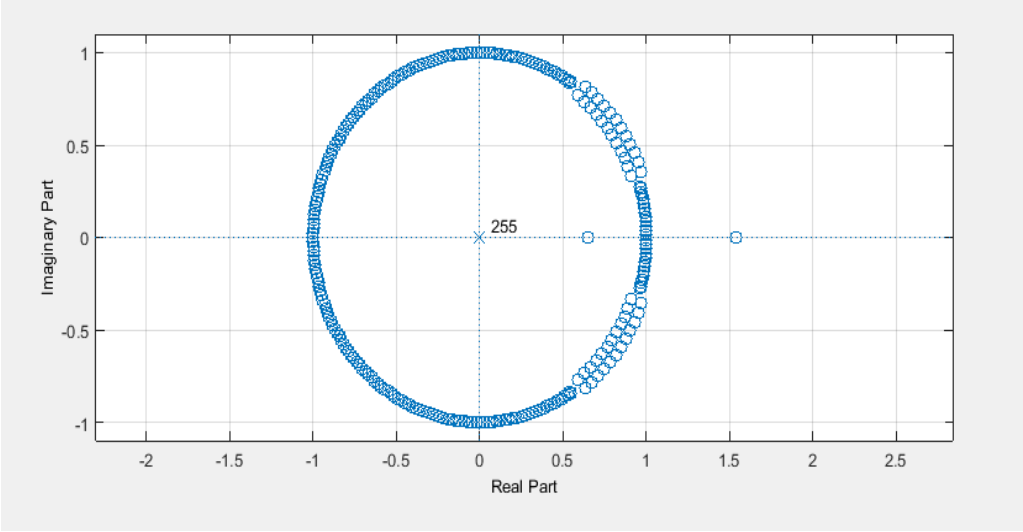
Figure 2: Plot of filter pass band ripple with order 255 satisfying requirement of 0.5dB ripple



As shown above, phase response is perfectly linear during pass band and the magnitude plot is what we expected. Hence, if the input signal only comprises of frequencies that are passed unaffected by the filter, the output should only be the delayed version of the input, with the same shape and all the samples are delayed by the same amount. Vector b of coefficients checked to ensure coefficients are symmetric. Use the command $fopen$, $fpintf$ and $fclose$ to automatically save the coefficient variable into the text file with double precision and put them into an array declaration of double type, which is then included

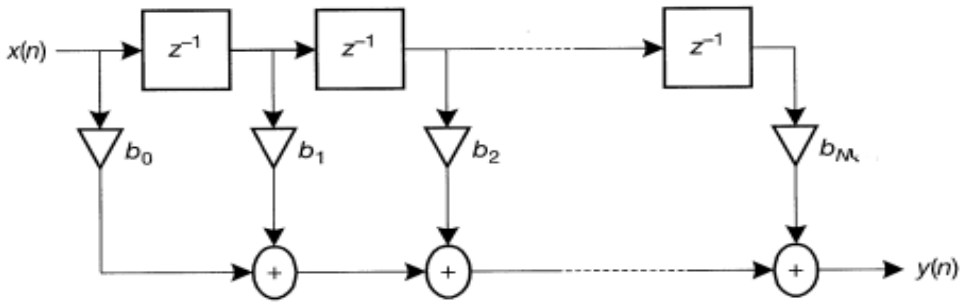
in the C program.
Stop band and Pass band Ripple: In discrete time, the imaginary axis maps to the unit circle. Zeroes on or close to the unit circle would pull down gain magnitude (zero or close to zero) at the corresponding frequency. Hence, for a band pass filter, zeros would be added below and above the frequency range that we want to pass. Ripple occurs in stop band while the frequency is transiting between zeros. If not all the zeros are on the unit circle, ripple might occur in the pass band as well.

Figure 3: Pole-zero Plots of Designed Filter



7 Filter Implementation

As mentioned above, to calculate new output sample, convolution sum is required with the current input sample through the M most recent samples, where M is the order of the FIR filter. Those input samples needed to be saved in buffer of *short* type of size N = M+1, since $mono_read_16Bit()$ only returns a 16-bit value. Non-circular buffer and circular buffer implementation are explored in this section based on the visualization below as a tapped delay line. In the C implementations for this lab, variable N is number of coefficients and N-1 is the order of filter.



Non-circular buffers: Similar to previous labs, the board, audio port and the hardware interrupts are initialized first. Then program enters an endless while loop until MCBSP receive interrupt occurs. MCBSP receive interrupt line is raised when a new sample is available in its internal buffer from the ADC. ISR is then loaded and executed. A buffer x of size of 256 (size of coefficient array b) is declared with initial values as zeroes to store the current input sample and previous N-1 input samples. Note a filter of order N-1 will have N taps (coefficients). Within the ISR, current sample is read using the $mono\_read\_16bit()$ function and always placed at index 0 position of the data buffer. Previous N-1 samples are moved along buffer from lower element to next higher before putting the new sample into buffer using a for loop. As a result, the age of the samples increases with the index of the data buffer. Then convolution function is performed involving MAC operations through another for loop. The current output sample $y(n)$ is then outputted to the codec using the $mono\_write\_16bit()$ function. Note that

samp must be declared as a double type variable and is initialized as zero each time ISR is executed, it is then automatically truncated to a 16-bit integer before outputting to the codec, hence rounding only occurs when current output sample computed and ready to be outputted, and the accuracy of zero positions are ensured to the maximum level.

```
void ISR_AIC (void)
{
        samp = 0;
        sampin = mono_read_16Bit();
        //mono_write_16Bit(mono_read_16Bit());
        //do a (N-1)th-order MAC with FIR Filter coefficients
        non_circ_FIR();
        //circ_FIR();
        //circ_FIR_faster1();
        //circ_FIR_faster2();
        //circ_FIR_faster3();
        //circ_FIR_faster4();
        mono_write_16Bit(samp);

}
```

```
void non_circ_FIR(void){

    int i;
    int j;
    for (i = N-1; i>0; i--)
    {
            x[i] = x[i-1]; //Previous samples moved along buffer from lower
                           //element to next higher
    }
    x[0] = sampin; //current input sample stored at index 0
    for (j = 0; j<N; j++) //for loop to compute convolution sum
    {
        samp += b[j]*x[j]; //MAC operation
    }
}
```

Circular buffers: The previous method that employs a non-circular buffer involves shifting data along buffer each time when new sample is read. This is inefficient in terms of run time as it involves copying lots of data around, especially for larger order filters. The $circ\_FIR()$ function manipulates pointer instead of shifting data. A *ptr* variable is initialized to N-1 (the order of FIR). *ptr* is decremented each time $circ\_FIR()$ called so that it always points to the most current input sample with the samples getting older towards the higher index positions in a circular fashion. When the index reaches beginning of array, it is wrapped around to the end. Only one for loop within the function to carry out MAC operations. An if statement is used within for loop to handle overflow and ensure N consecutive samples are considered for convolution sum of the FIR filter. The *ptr* is correctly updated outside the loop using % operator.

```
void circ_FIR(void){
    int k;
    x[ptr] = sampin; //current input sample stored at the position where ptr points to
    for (k = 0; k<N; k++)
    {
        if(ptr+k > N-1)
        {
            samp += b[k]*x[ptr+k-N]; //Allows convolution sum of consecutive samples weighted by filter coefficients
        }
        else{
            samp += b[k]*x[ptr+k]; //MAC operation
        }
    }
    ptr = (ptr+2*N-1)%N;//pointer update using modulus operator
}
```

However, the implementation of %(modulus) operator in assembly code is more complicated, which takes around 19 clock cycles (without optimization):

```
0x00006D10:   0200B06E        LDW.D2T2    *+B14[176],B4
0x00006D14:   0280FFAA        MVK.S2      0x01ff,B5
0x00006D18:   01807FA8        MVK.S1      0x00ff,A3
0x00006D1C:   018FEDD8        NOT.L1      A3,A3
0x00006D20:   00000000        NOP
0x00006D24:   0210A07A        ADD.L2      B5,B4,B4
0x00006D28:   0290EDA2        SHR.S2      B4,0x7,B5
0x00006D2C:   029709A2        SHRU.S2     B5,0x18,B5
0x00006D30:   0290A07A        ADD.L2      B5,B4,B5
0x00006D34:   01947F78        AND.L1X     A3,B5,A3
0x00006D38:   020C90FA        SUB.L2X     B4,A3,B4
0x00006D3C:   0200B07E        STW.D2T2    B4,*+B14[176]
0x00006D40:   07BD005A        ADD.L2      8,SP,SP
0x00006D44:   000C0362        B.S2        B3
0x00006D48:   00008000        NOP         5
```

where initially B4 contains value of ptr, B5 contains value of $2N-1$. A3 is 0xff and used to get the lower two bytes of $ptr+2N-1$. The manipulations with value in B5 seem unnecessary since no matter what the result of SHR and SHRU instructions is, B5 is only non-zero for the lower two bytes. The code is always trying to get the lower two bytes of $ptr+2N-1$ since $N=2^8$, in which case dividing N is equivalent to shifting the dividend to the right by eight bits, hence the remainder of the division will be the value of lower two bytes of $ptr+2N-1$.

Faster Implementation: By using bit-wise *AND* operator instead of using modulus for pointer update, we could achieve a faster implementation. Note that a *(ptr1+1)&255* operation for pointer update is same as *(ptr1+1)%256*. By reversing the order of data storing, we could eliminate the multiplication inside the brackets. Now the update of pointer only takes 15 clock cycles (without optimization):

```
void circ_FIR_faster1(void){
    int n;
    x[ptr1] = sampin; //Current sample stored at the position where ptr1 points to
    for (n = 0; n<N; n++)
    {
        if(ptr1-n < 0)
        {
            samp += b[n]*x[ptr1-n+N]; //Allows convolution sum of consecutive samples at the position where ptr1 points to
        }
        else{
            samp += b[n]*x[ptr1-n];//MAC operation
        }
    }

    ptr1 = (ptr1+1)&255; //pointer update using bit-wise and operator

}
```

For N that is not a power of 2, we could use both modulus operator(%) or $if - else$ methods to update the pointer:

```
if(ptr1 == N-1){
    ptr1 = 0;
}
else{
    ptr1++;
}
```

Although $if - else$ takes 2 more cycles than modulus method, it is actually much faster under -o2 optimization.

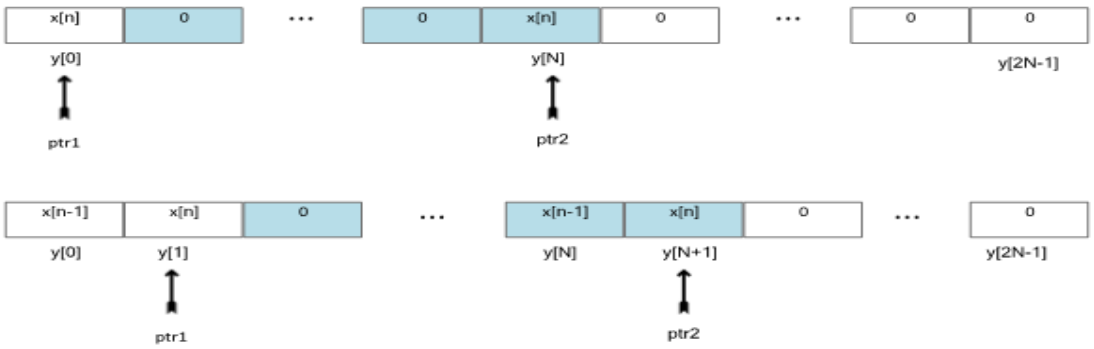
-	No optimisation	-o0	-o2
Modulus	19	22	17
If-else	21	21	13
Bit-wise AND	15	15	13

The code could be further improved. At each iteration when we access the data array, extra clock cycles needed to calculate the correct index, however, when $ptr1 - n$ is greater than zero, the correct sample is just at the next left position.A new variable, *index* is introduced to keep track of the data. It decrements at each iteration. If *index* reaches 0, it would be updated to N-1, hence allowing fetching of consecutive data. This avoids the need to carry out heavy arithmetic operations to fetch samples each time the loop iterates.

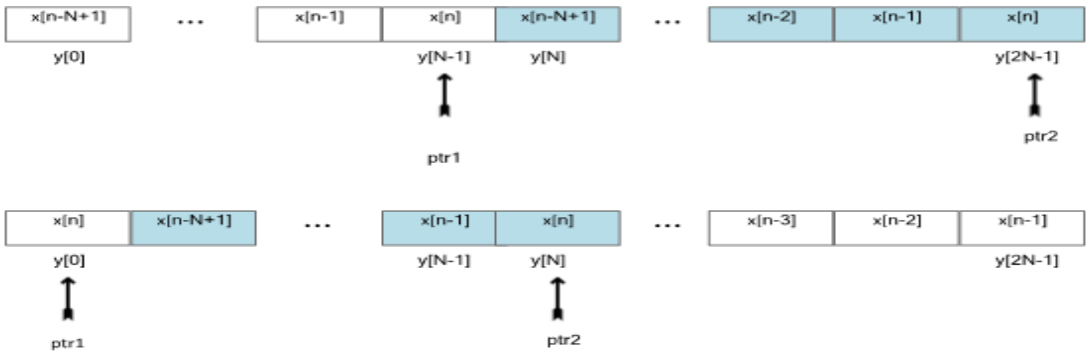
```
void circ_FIR_faster2(void){
    int k;
    int index;
    index = ptr1;
    x[ptr1] = sampin; //Current sample stored at the position where ptr1 points to
    for (k = 0; k<N; k++, index--)
    {
        if(index <0)
        {
            index += N; //Allows convolution sum of consecutive samples weighted by filter coefficients
        }
        samp += b[k]*x[index]; //MAC operation
    }
        ptr1 = (ptr1+1)&255; //pointer update using bit-wise and operator
}
```

However, an if statement in assembly code is implemented using conditional branch which has 5 delay slots. We could remove this if in loop by adopting double-size circular buffer implementation. In doing so, there is no need to check for overflow inside the loop. The idea is the following:

- The buffer is divided into two circular buffers each of size N, input samples are stored in both circular buffers in the same direction from lower to higher indexes. Hence, new sample received is stored in 2 indexes, with the second index always first index + N. In doing so, there's no need to check for overflow to compute convolution sum, as all N-1 previous samples are situated towards the left of second index. The shaded regions are the input samples used to carry out the convolution sum:



- When *ptr*2 reaches the end of the buffer, it is wrapped back to N.



Because the buffer size is 2N, there are always N-1 input samples available directly to the left of the position where *ptr*2 points to, hence no need to check the offset in the loop.

The code is as below:

```
void circ_FIR_faster3(void){
    int m;
    int index;
    index = ptr2;
    y[ptr1] = sampin;//store current input into first circular buffer of size N
    y[ptr2] = sampin;//store current input into second circular buffer of size N
    for(m=0; m<N; m++, index--)//convolution sum of consecutive samples weighted by filter coefficients
    {
    samp += b[m]*y[index]; //MAC operation
    }
    ptr1 = (ptr1+1)&255;//ptr1 goes back to 0 when reaching N-1, otherwise increases by 1
    ptr2 = ptr1+N;//ptr2 is always larger than ptr1 by N
}
```

Faster Implementation: Knowing that linear phase FIR filters have symmetrical coefficients, hence it is possible to use a *for* loop half the size as before. For linear phase FIR filters, coefficients b[i] = b[N-i-1]. Since order N = 256 and is even, loop of N/2 times is required. The middle coefficient does not have to be considered. The faster implementation takes advantage of symmetrical property and utilizes double the memory to avoid overflow checks. Two variables *index* and *index1* are introduced, whereas *index* is the index of current sample used in calculation and *index1* = *index*-N+1 is the index of its symmetry pair. To take advantage of symmetrical properties, *index* is decremented each cycle and *index1* incremented. A new variable *ptr*3 is used instead of the *ptr*1 above, which is an unsigned 8-bit integer, so that it will overflow after reaching N-1.

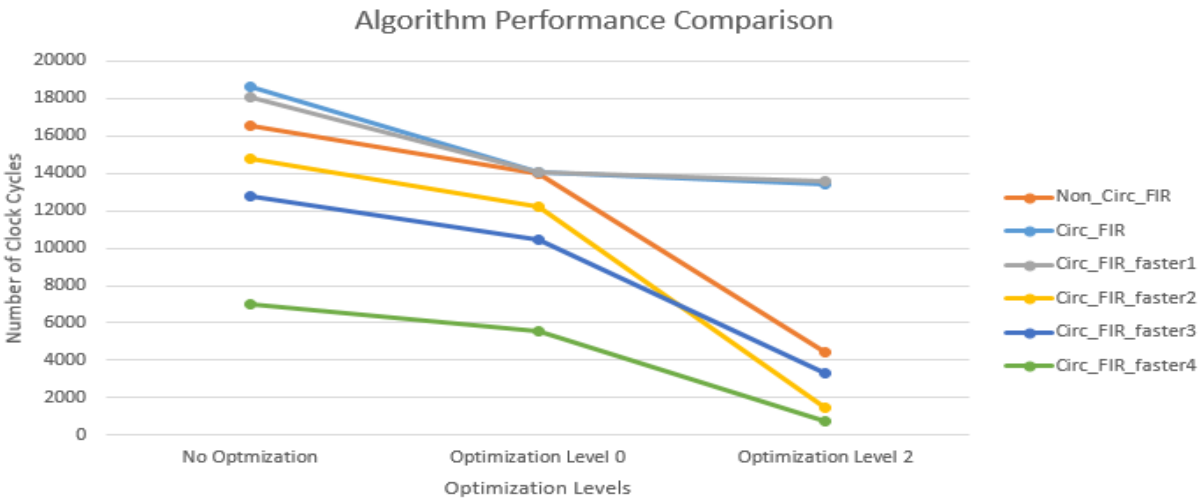
```
void circ_FIR_faster4(void){

    int m;
    int index,index1;
    y[ptr2] = sampin;//store current input into first circular buffer of size N
    y[ptr3] = sampin;//store current input into second circular buffer of size N
    index = ptr2;//index of input sample used in current calculation
    index1 = ptr3+1;//index of its symmetric pair
    for(m=0; m<(N/2); m++, index--,index1++)//convolution sum of consecutive samples weighted by filter coefficients
    {                                       //loop from 0 to N/2 (N even) cause coefficient symmetry of linear phase FIR filter
    samp += b[m]*(y[index]+y[index1]);//use symmetry properties
    }
    ptr3++;//ptr1 goes back to 0 after reaching N-1, otherwise increases by 1
    ptr2 = ptr3+N;//ptr2 is always larger than ptr3 by N
}
```

Summary of Algorithm difference:

- **Non_circular_buffers**: involves two *for* loops, one for shifting of data as new sample arrives, the other one for computation of convolution sum.
- **Circ_FIR**:manipulates *ptr* instead of copying data. Newer samples saved at lower indexes in circular fashion.*If* statement for overflow check and % operator for pointer update.
- **Circ_FIR_faster1**: Reversed order of data storage and bit-wise & for pointer update.
- **Circ_FIR_faster2**: Extra variable *index* used and decremented each time for MAC operations.
- **Circ_FIR_faster3**: Used double-size circular buffers to rid *If* statement within for loop.
- **Circ_FIR_faster4**: Used symmetrical properties of FIR filter coefficients , double memory and a 8-bit number pointer.

Algorithm Performance:

-	No optimisation	-o0	-o2
Non_circ_FIR	16553	14005	4396
Circ_FIR	18648	14040	13448
Circ_FIR_faster1	18092	14028	13597
Circ_FIR_faster2	14802	12245	1501
Circ_FIR_faster3	12778	10446	3293
Circ_FIR_faster4	6980	5572	779



Algorithm performance is measured using the profiling clock method after breakpoints are added at the start and end of the ISR function, including mono_read_16Bit() and mono_read_16Bit() operations. Note

that the speed could be improved for all implementations by a few hundreds of cycles by using other data types for *samp*, such as $float$ or *short*, since they occupy less number of bits and arithmetic operations over them takes less time. However in this case we always use type *double* to ensure the accuracy. As shown in the table above, when no optimization takes place, the cycle number decreases from top to bottom. As optimization level increases, all implementations are improved significantly.At -O2 level, circ_FIR_faster4 is the fastest implementation, which at best case only takes 779 cycles to compute one output sample, while circ_FIR_faster2 is actually much better than cir_FIR_faster3. This behavior draws our attention to what optimizations the C compiler performs at different levels:

- -O0
  - Perform control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates unused code
  - Simplifies expressions and statements
  - Expands calls to functions declared inline
- -O2
  - Performs all -O0 optimizations
  - Performs local copy/constant propagation
  - Removes unused assignments
  - Eliminates local common expressions
  - 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

At Optimization level 0, compiler analyses branching behavior to remove branches and redundant conditions. Note that these features are more common in the first four implementations, hence more significant improvement is witnessed in these implementations. Loop rotations are performed such that compiler evaluates loop conditionals at end of loops; this saves extra cycles for all 6 implementations. At optimization level 2, all 6 implementations are optimized with decrease in instruction cycles. This is due to software pipelining, in which all processing units are used and operations with no data dependencies are executed in parallel.

8 Lab Results

Firstly, the results of implementation of band-pass filter are shown with a sine wave attached to the input by using software signal generator. Make sure that the input does not exceed $2V_{RMS}$ to avoid distortions, since the full-scale range of ADC at the line in of AIC23 audio chip is $1V_{RMS}$ at $V_{DD} = 3.3V$, and potential divider circuit at the input will divide the input voltage by two. Implementations with non-circular and circular buffers have achieved identical results:

Figure 4: Filter output with Sine wave input freq 300Hz

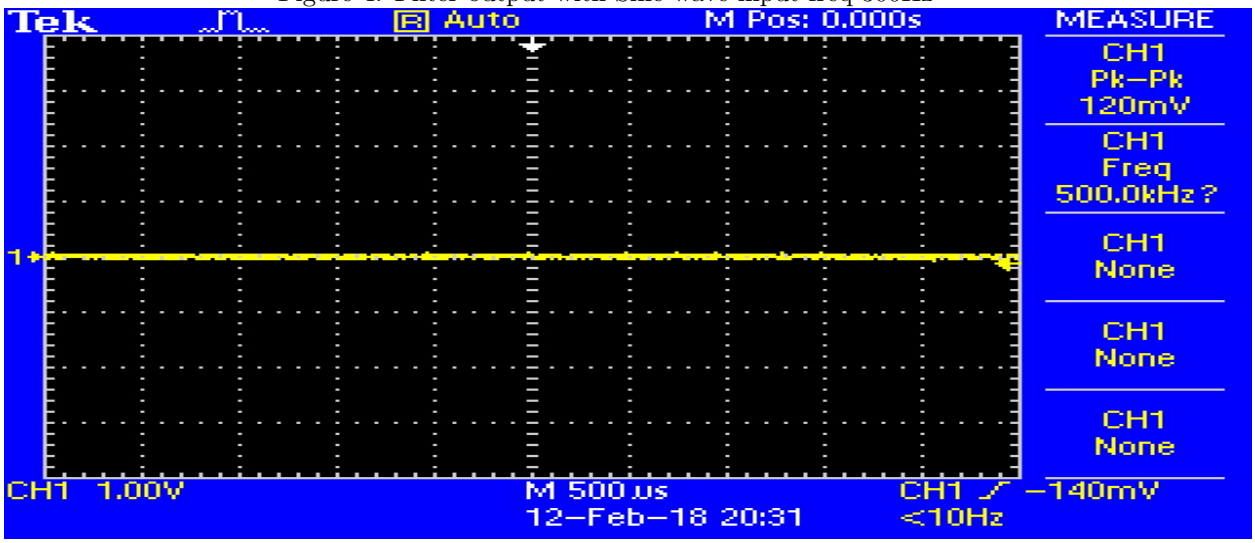


Figure 5: Filter output with Sine wave input freq 354Hz

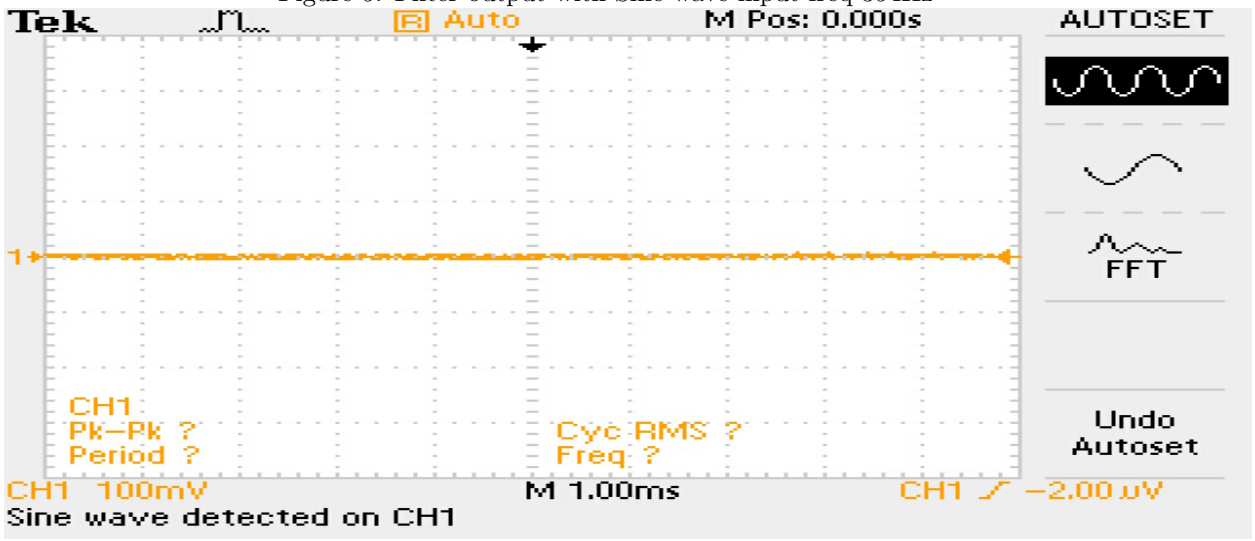


Figure 6: Filter output with Sine wave input freq 400Hz

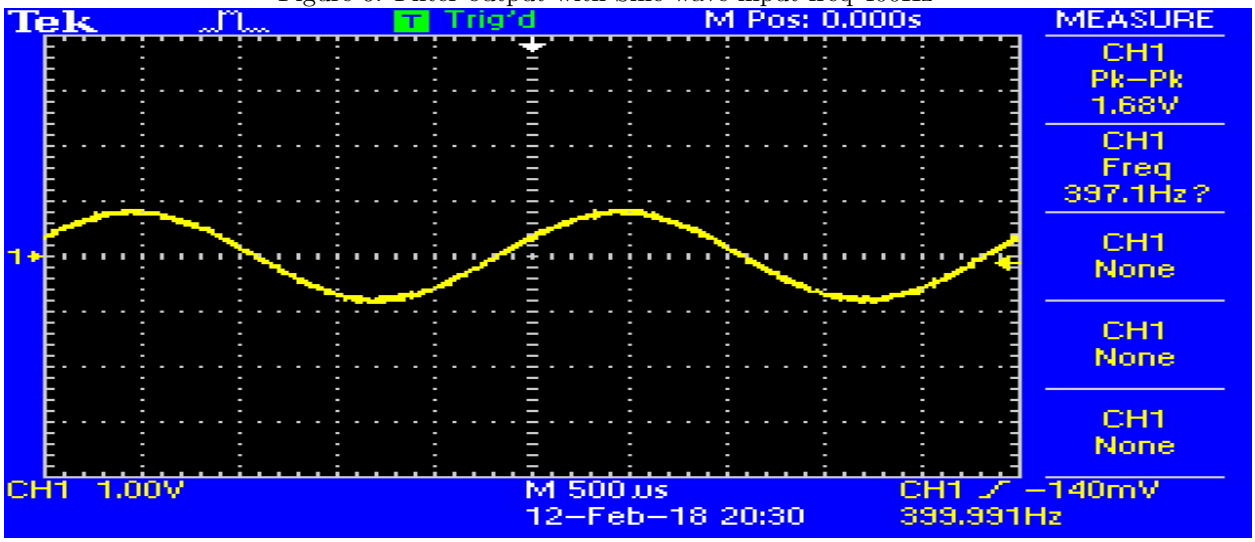


Figure 7: Filter output with Sine wave input freq 416Hz

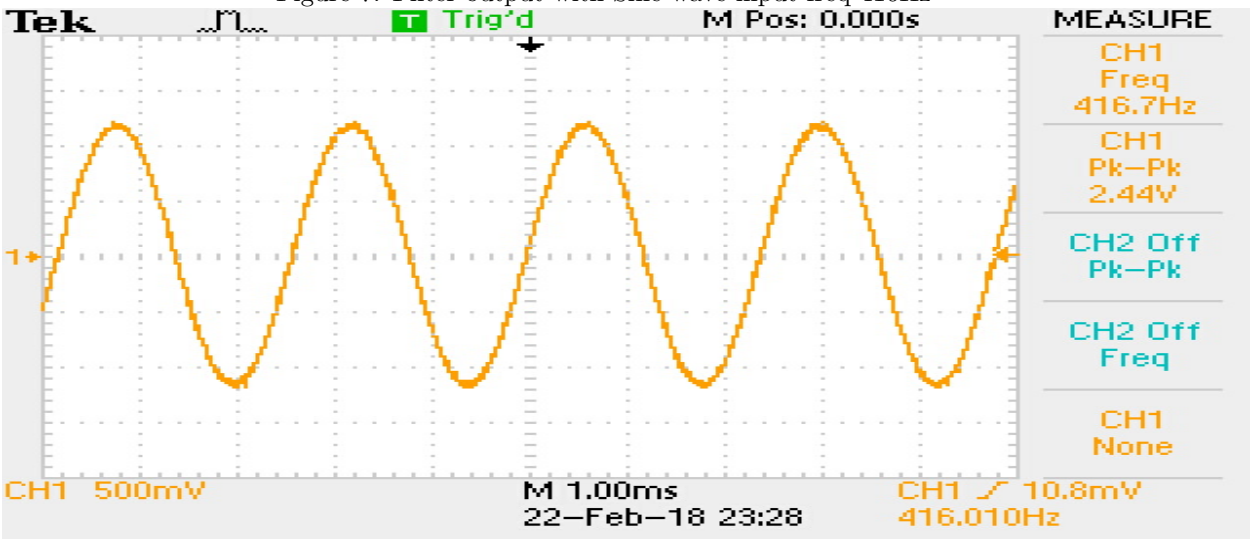


Figure 8: Filter output with Sine wave input freq 500Hz

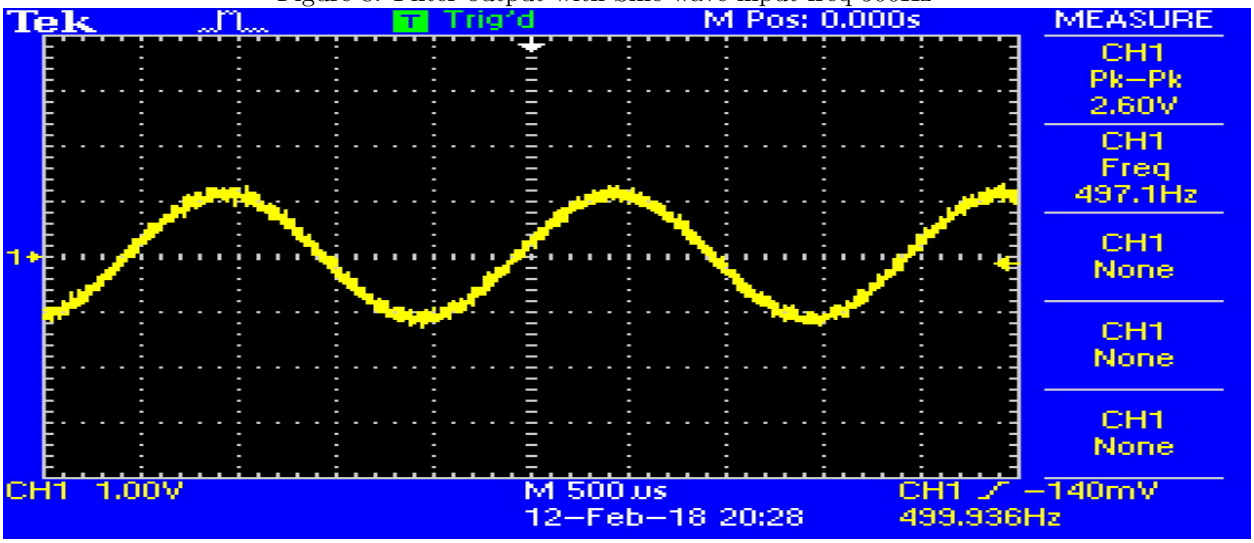


Figure 9: Filter output with Sine wave input freq 1kHz

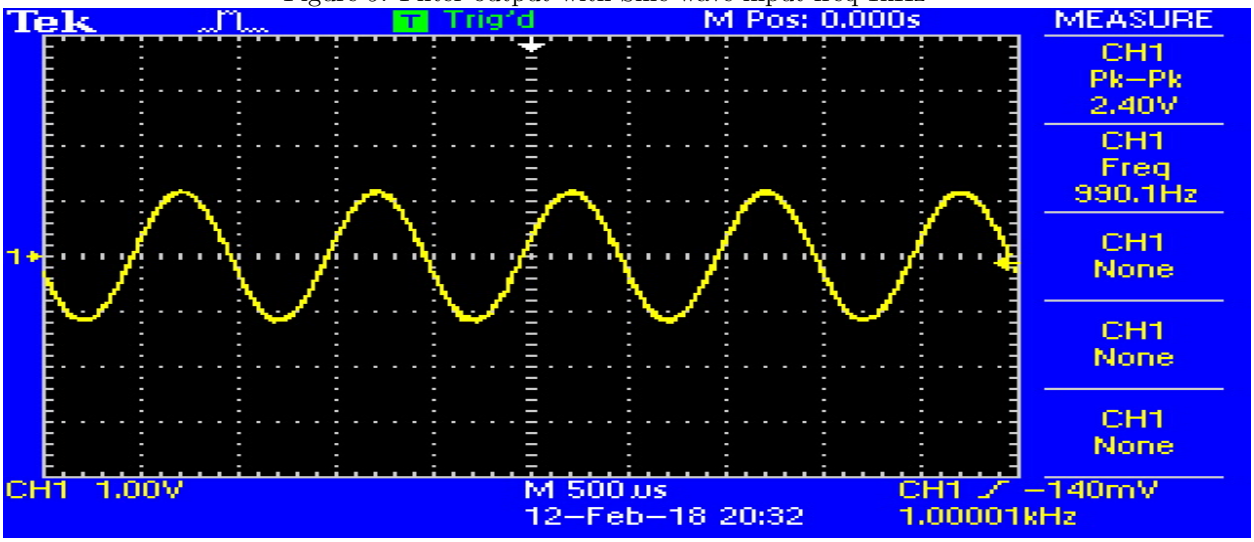


Figure 10: Filter output with Sine wave input freq 1.199KHz

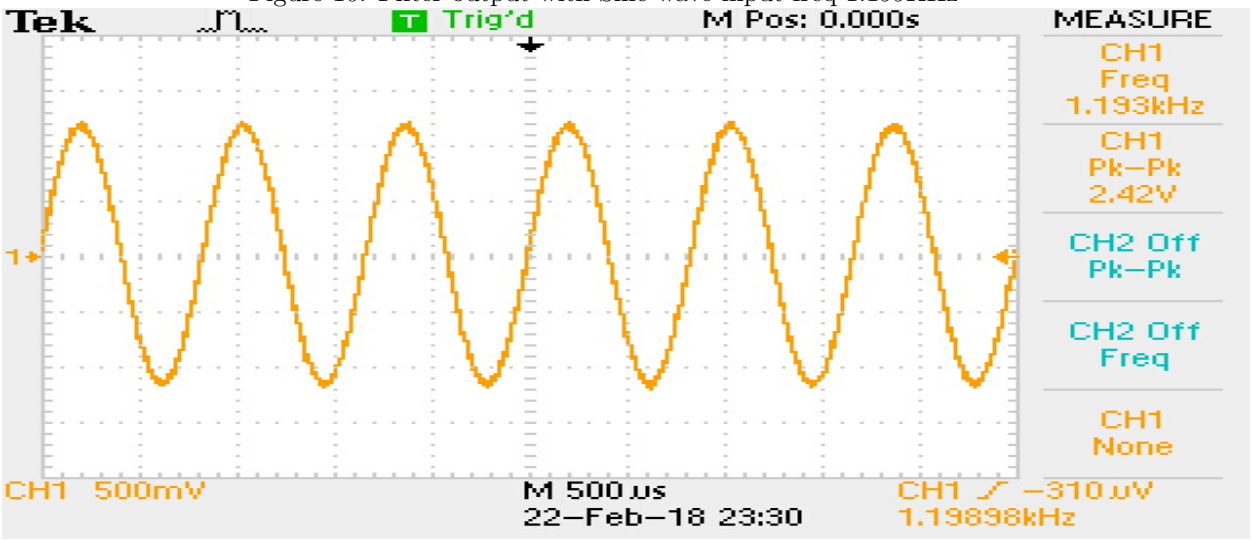


Figure 11: Filter output with Sine wave input freq 1.271Hz

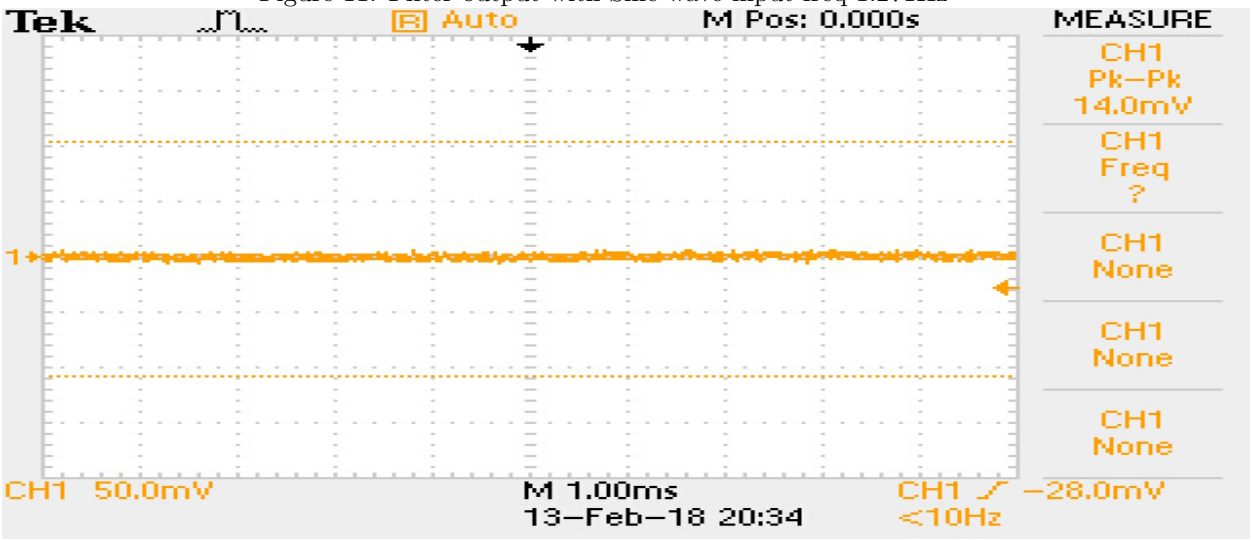
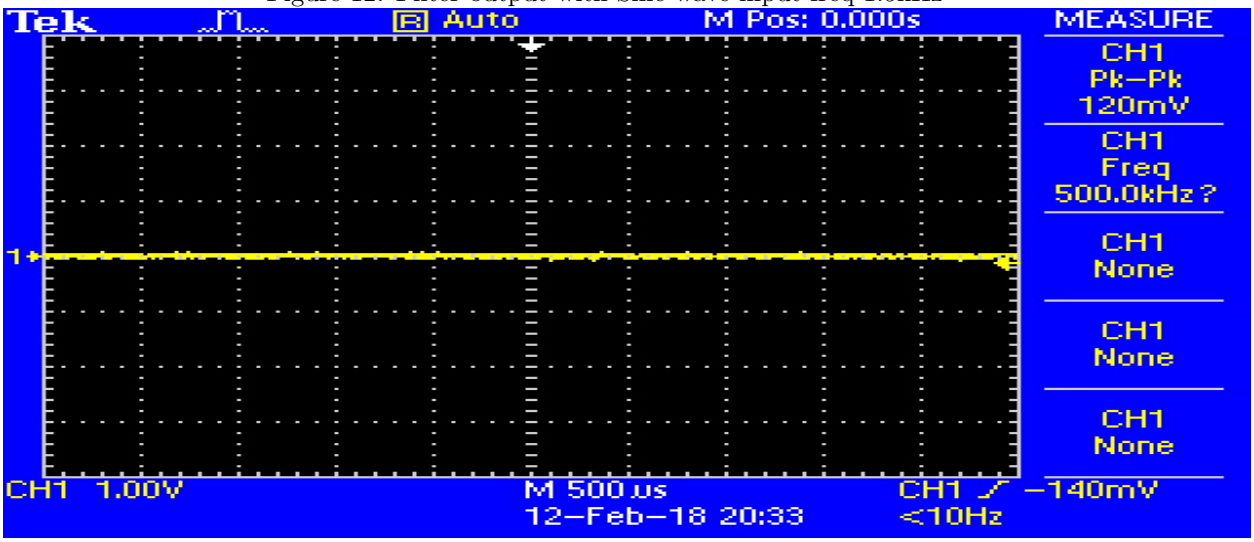


Figure 12: Filter output with Sine wave input freq 1.5kHz



As we can see from the results above, for an input sine wave below 355Hz and above 1.27kHz, no constructive output is observed, since output signal amplitude is lower than that of noise. A sine wave with increasing amplitude is observed around the first transition band($355Hz - 415Hz$), and that with decreasing amplitude is observed at the next transition band($1.2kHz - 1.27kHz$).
In the next section, unity gain and phase responses are obtained using Audio Precision APX520. To achieve this, in the ISR the samples are simply read and written out directly, with no FIR filter applied:

Figure 13: Code to obtain unity gain and phase response

```
void ISR_AIC (void)
{
        //samp = 0;
        //sampin = mono_read_16Bit();
        mono_write_16Bit(mono_read_16Bit());
        //do a (N-1)th-order MAC with FIR Filter coefficients
        //non_circ_FIR();
        //circ_FIR();
        //circ_FIR_faster1();
        //circ_FIR_faster2();
        //circ_FIR_faster3();
        //circ_FIR_faster4();

        //samp = (short)samp; //truncate samp to a 16-bit integer
        //mono_write_16Bit(samp);

}
```

Figure 14: Unity Gain Response.

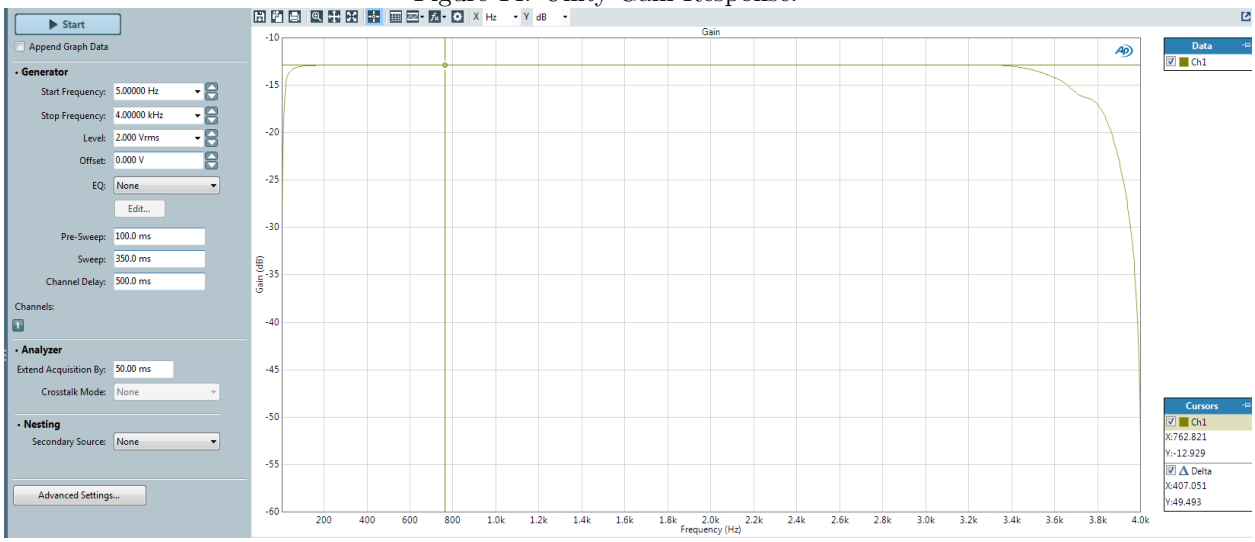
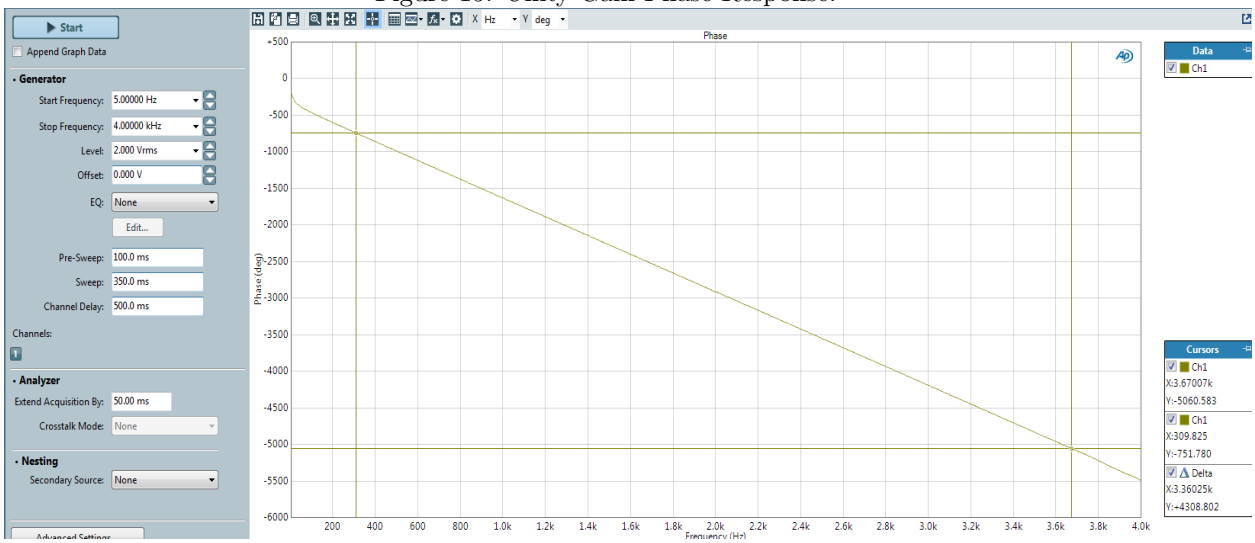


Figure 15: Unity Gain Phase Response.



As shown by figure 14 and 15, the unity gain response has a pass band gain of about -12.9dB, and rolls off at some very small frequency and when approaching $4kHz$. These might be due to other filtering operations inside the audio chip, since the gain in dBs of series filters will sum up towards the resultant behavior:

- There is a gain of 1/4 on input stage of DSK unit, which translates to -12.04dB. The input samples are first halved due to potential dividers on input stage and the second halving can be attributed to mono_read_16Bit(), where the processor reads Left and Right sample, halves each amplitudes and computes sum to create a mono input. This attenuation of 12.04dB plus some other attenuation inside the chip explains the pass band gain of -12.9dB.
- The unity gain response starts to roll off when approaching 4kHz. This can be attributed to anti-aliasing filter and the reconstruction filter at the input and output of the AIC23 audio chip respectively, which have cutoff at Nyquist Frequency($f_N = 4kHz$):

Figure 16: Anti-aliasing Filter

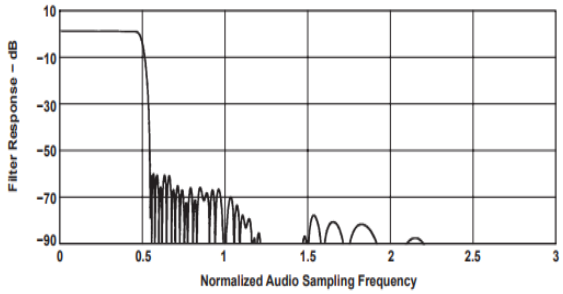
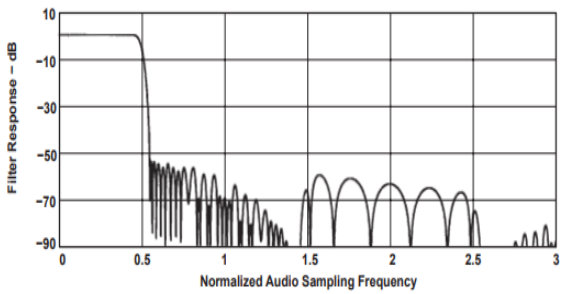
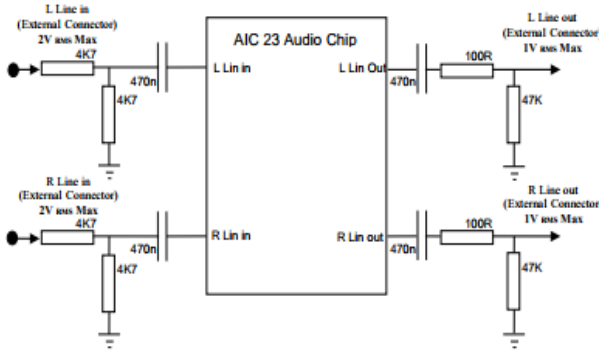


Figure 17: Reconstruction Filter



- There is a high-pass filter with cut-off of about 7Hz at the output of the chip, which explains the roll-off at small frequencies.

Figure 18: AIC23 Audio chip external components



On the other hand, the unit gain phase response is perfectly linear. The following are the results when FIR filtering is applied:

Figure 19: Magnitude Response.

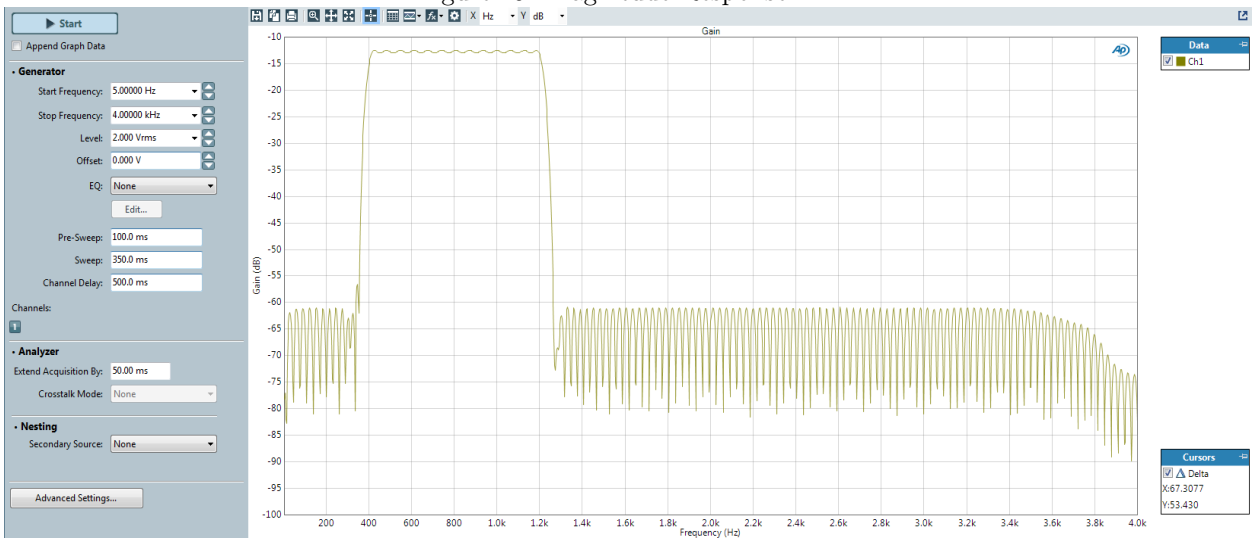


Figure 20: Pass band ripple Specification is fulfilled

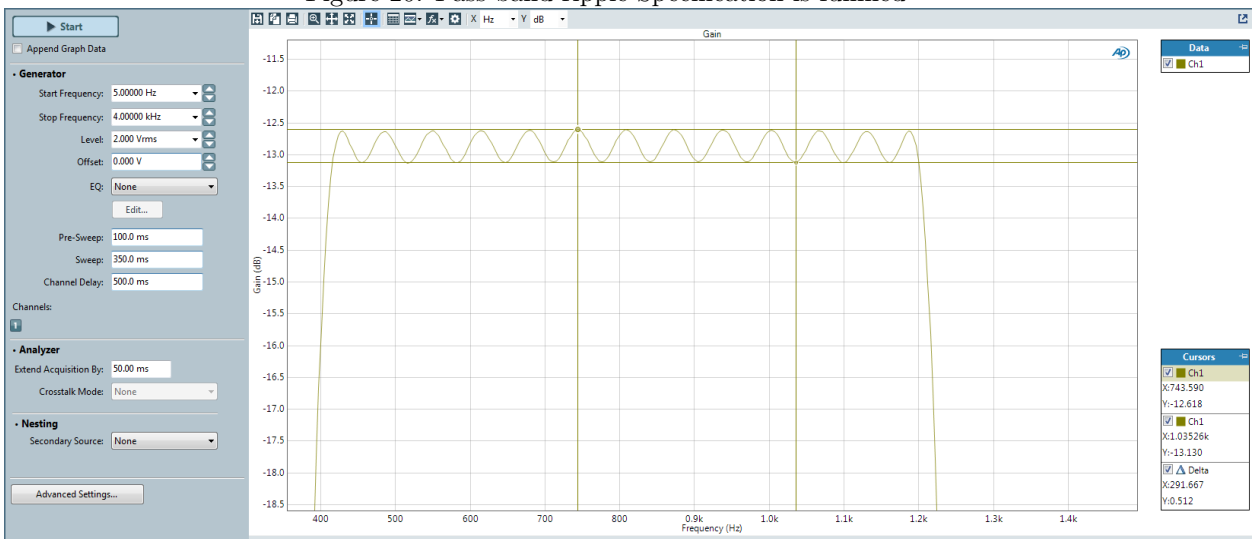


Figure 21: Cutoff frequency specification is fulfilled
ff1.pdf ff1.png ff1.jpg ff1.mps ff1.jpeg ff1.jbig2 ff1.jb2 ff1.PDF ff1.PNG ff1.JPG ff1.JPEG ff1.JBIG2 ff1.JB2 ff1.eps

Figure 22: Cutoff frequency specification is fulfilled

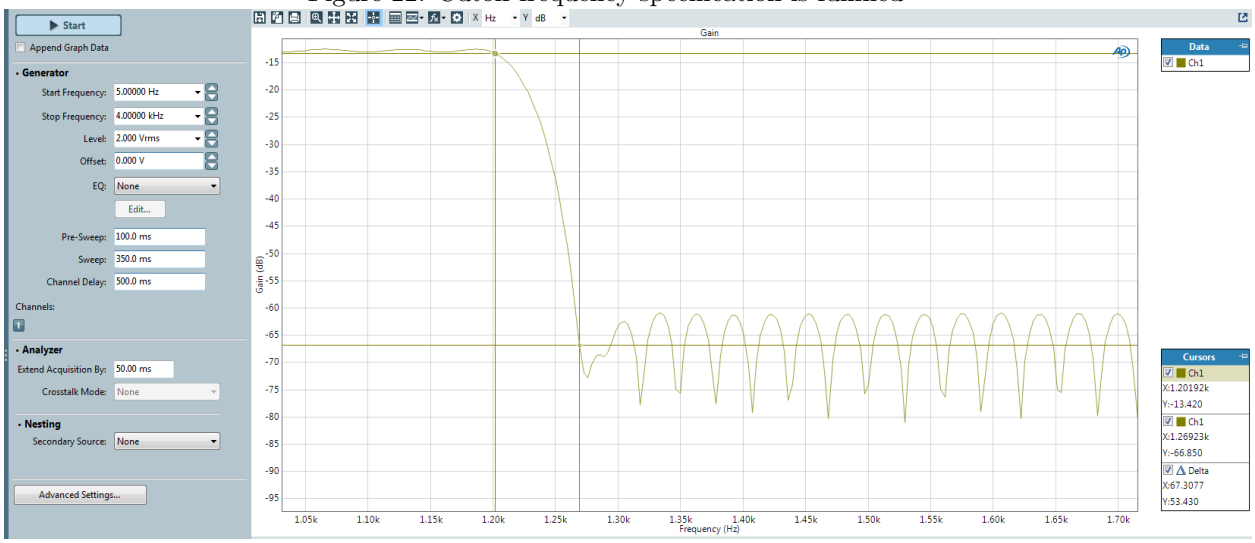
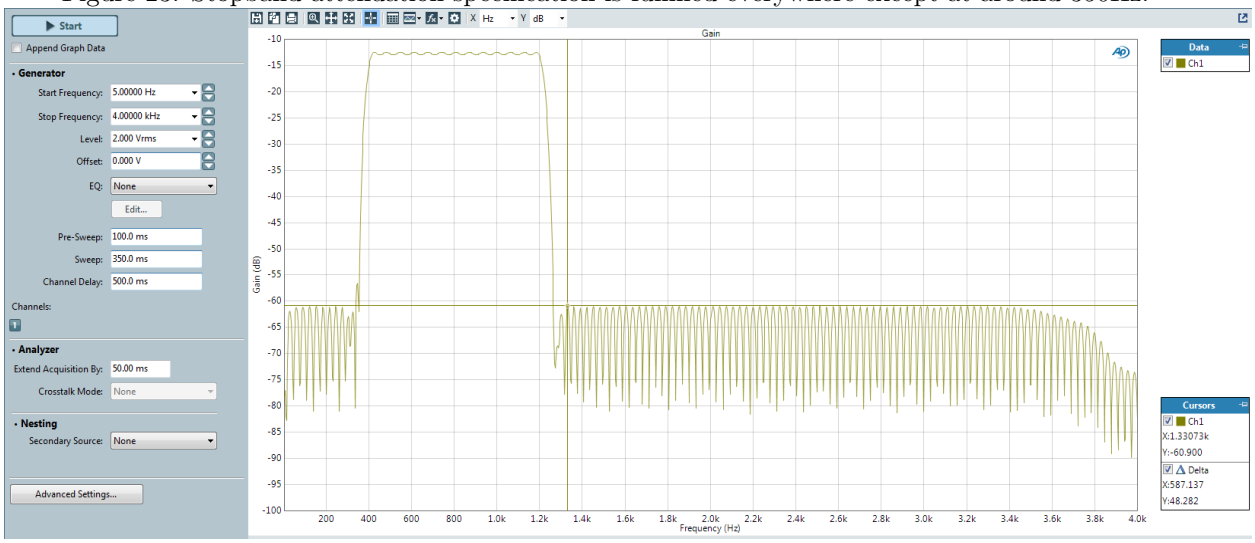
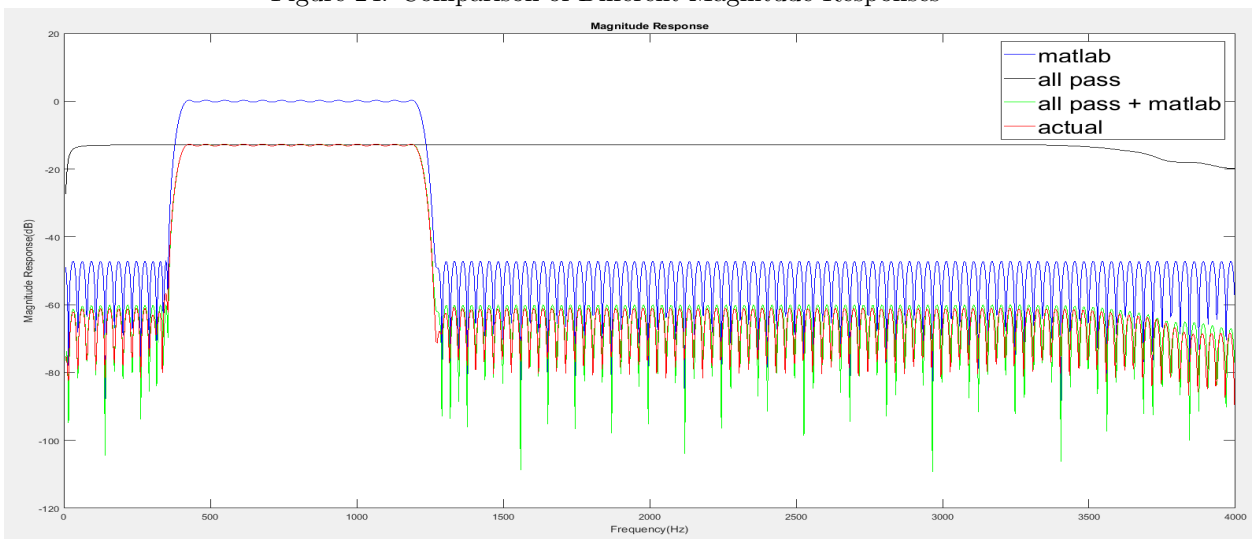


Figure 23: Stopband attenuation specification is fulfilled everywhere except at around 350Hz.



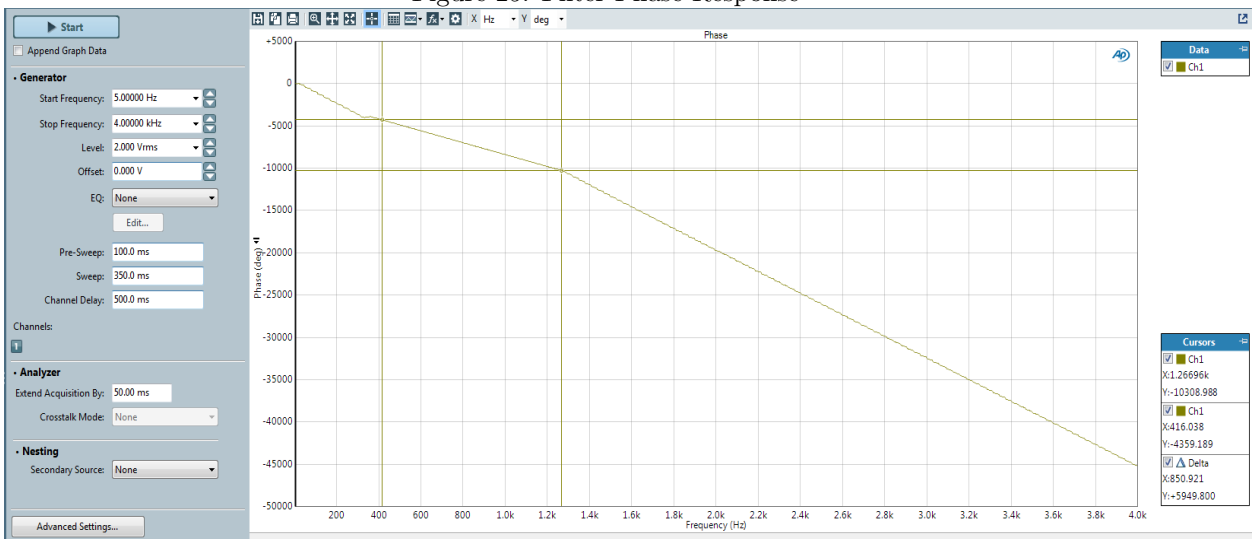
The results for all 6 implementations are checked with spectrum analyzer and produce identical results. Apart from stopband gain, all specifications listed in the **Filter Specification** section are met. It is obvious in theory that the resultant magnitude response is different from the Matlab plot, since it should be approximately the magnitude response given in the Matlab plus the unity gain response. This explains why the average pass band gain is about -12.9dB and we can see the peak amplitude in the stop band starts to roll-off when approaching 4kHz while the roll-off at low frequency is less obvious. There is a small peak observed in the magnitude response just before the first transition band (roughly 350Hz). This is explained in the new paragraph.

Figure 24: Comparison of Different Magnitude Responses



To compare the actual FIR filter response and theoretical response, data are exported from Audio Precision. The slight deviation of actual response(the red curve) from the theoretical response(the green curve) is probably due to finite precision error, as filter coefficients generated from Matlab only have double precision. Also, the output samples are automatically truncated to a 16-bit value before outputting from the audio chip calling $mono\_write\_16Bit()$ function, this gives rise to extra loss of accuracy by moving the zeros slightly away from their theoretical positions.

Figure 25: Filter Phase Response



A perfectly linear phase response is observed between 415Hz and 1.2kHz, suggesting no phase distortion introduced. In the ideal phase plot generated from MATLAB, during the two stop bands the phase plot has a flat roughly "sawtooth" waveform. Since FIR filter phase plot from Audio Precision should be approximately equal to the sum of the unity gain phase plot and ideal phase plot, we see "sawtooth" waveforms with decreasing center at the stopbands. The group delay of the FIR filter from audio precision should be the sum of group delay of ideal filter response plus that of unity gain. The group delays of passband are calculated below using the cursor data from the lab results above.

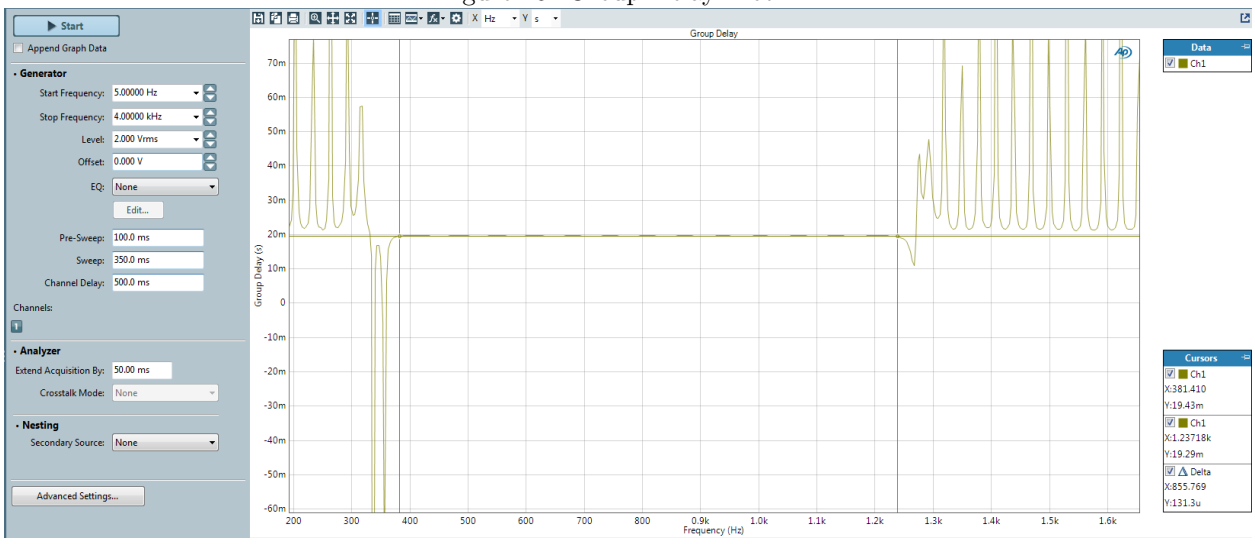
$$Group\ Delay\ in\ seconds = \frac{-1}{360^\circ} \cdot \frac{d\Theta}{df},\ where\ \Theta\ is\ in\ degrees\ and\ f\ is\ in\ Hz$$

$$t_{g,filter(Matlab)} = \frac{-1}{360^\circ} \cdot \frac{-1813}{316} = 15.94ms$$

$$t_{g,unity} = \frac{-1}{360^\circ} \cdot \frac{-4308.802}{3360.25} = 3.56ms$$

By looking at the Group Delay plot, we could see that the group delay during the pass band(415Hz-1.2kHz) is constant and equal to $19.29ms$, which is approximately the sum of two group delays calculated above.

Figure 26: Group Delay Plot



Since there exist deviations between real filter response and Matlab simulation, the stopband spec-

ification is not met. In new filter design, the stopband gain is decreased from -48dB to -60dB. Order increases from 255 to 303 and all specifications are now met.

Figure 27: Magnitude response

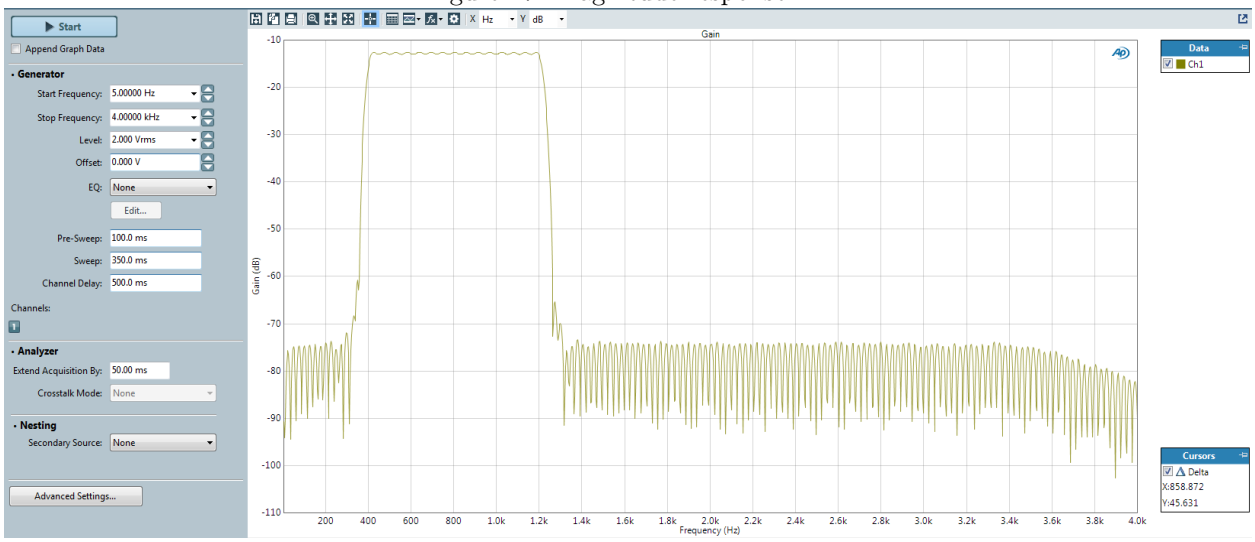


Figure 28: The small peak before 1st transition band apparent in previous result is dragged down. Cutoff frequency specifications are met

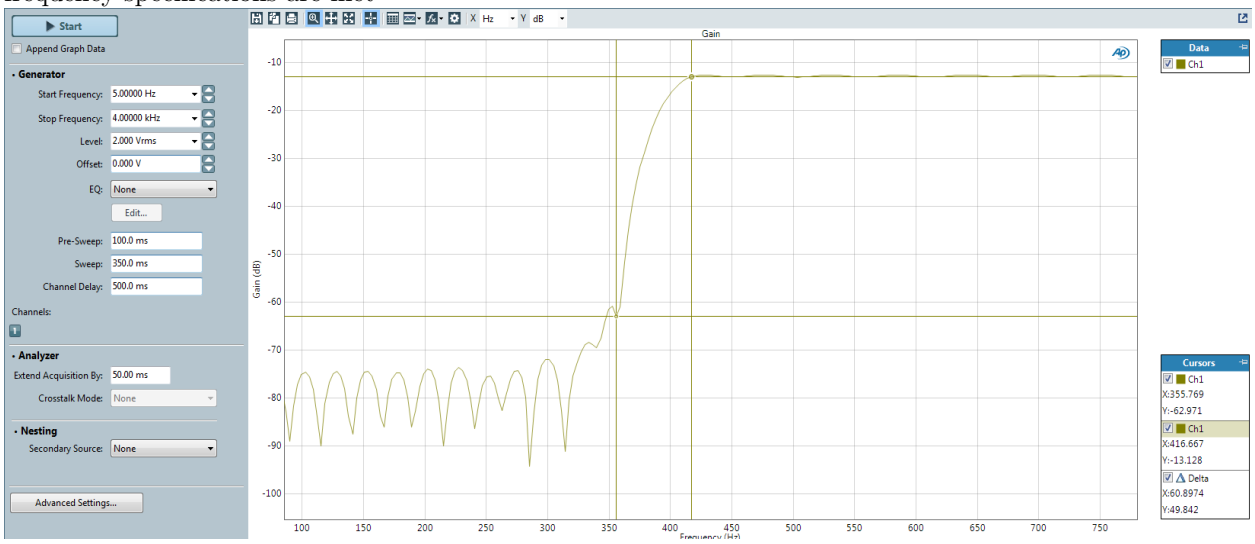


Figure 29: Cutoff frequency specification is met

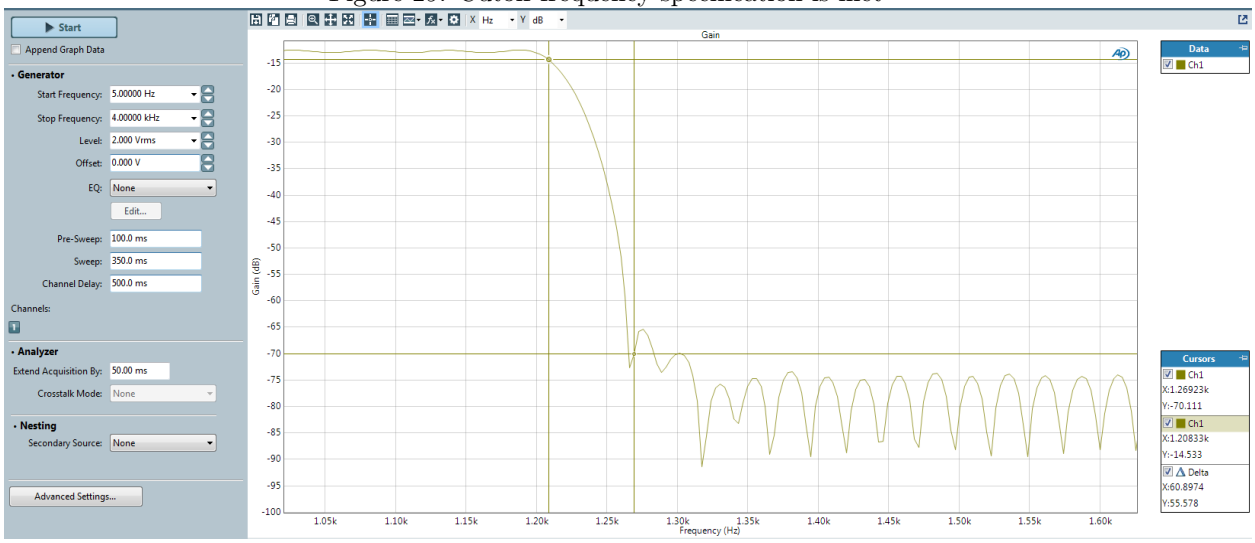


Figure 30: Passband ripple specification is met

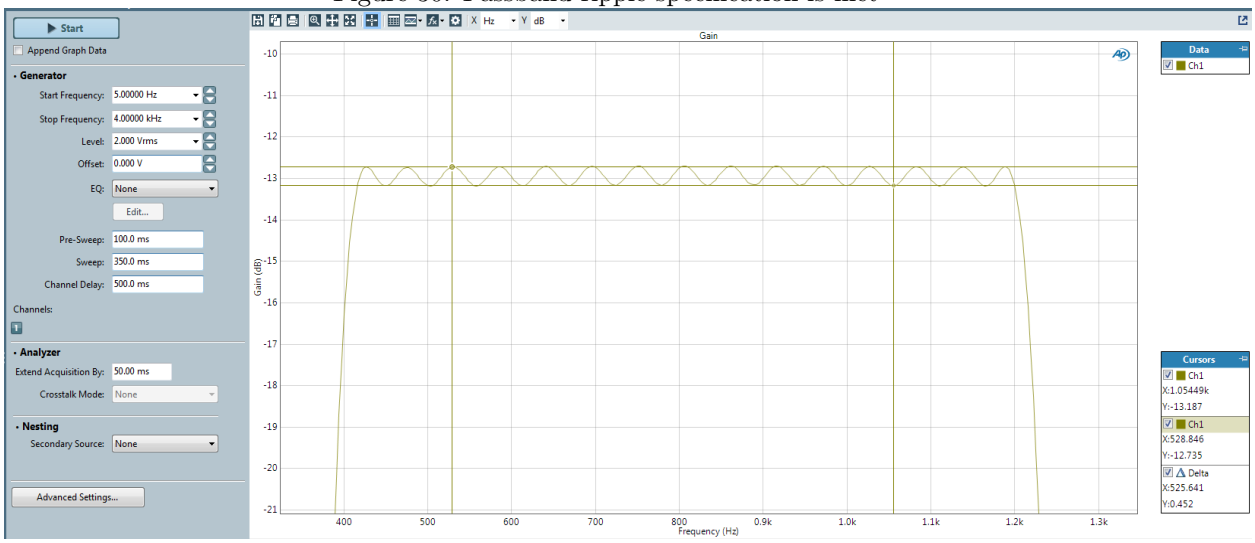
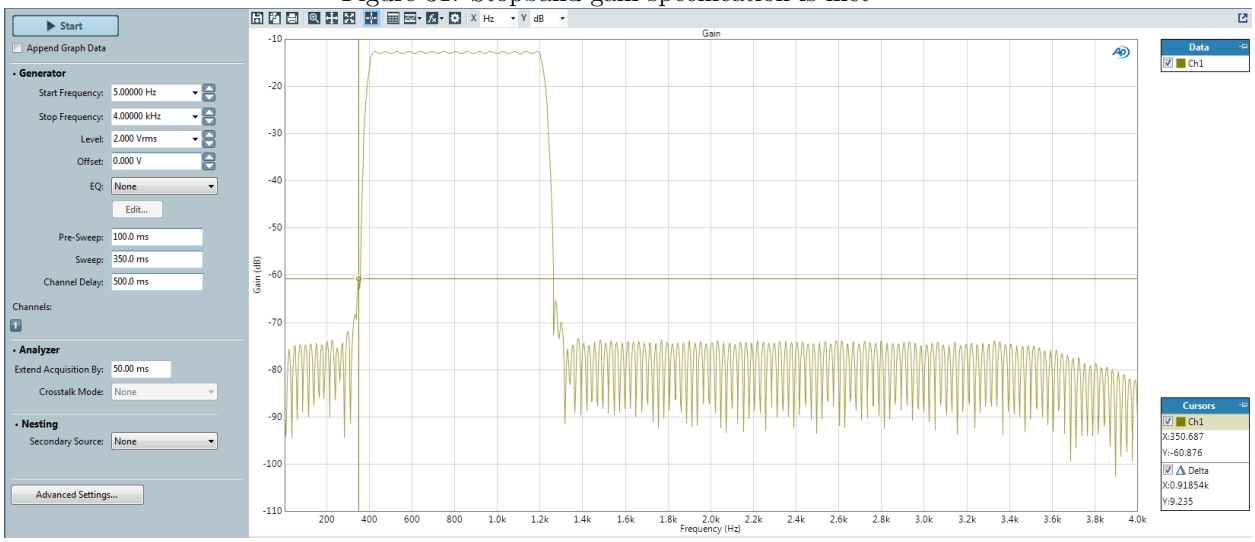


Figure 31: Stopband gain specification is met



9 Appendix

- Filter Design

```
rp = 0.5; %pass band ripple(peak-to-peak)
sg = 48; % min stop band attenuation
fs = 8000; %sampling frequency
f = [356.75 415 1200 1268]; %cut-off frequencies
a = [0 1 0]; %desired gain(not in dB)
Dstop = 10^(-sg/20); %max gain in stop band
Dpass = (10^(rp/20)-1)/(10^(rp/20)+1);%max allowed pass band gain deviation
dev = [Dstop Dpass Dstop];%vector of maximum gain deviations allowable for each band(not in dB)
[n,fo,ao,w] = firpmord(f,a,dev,fs); %[appximate filter order,normalised frequency band edges, freque
b = firpm(n,fo,ao,w); %filter coefficients
freqz(b,1,1024,fs) %plot frequency response
title('FIR Filter')
%save fir_coef.txt b -ascii  -double -tabs; %save to fir_coef.txt
fileID = fopen('fir_coef.txt','w');
formSpec1 = 'double b[]={%.16e';
fprintf(fileID,formSpec1,b(1));
formSpec2 = ', %.16e';
fprintf(fileID,formSpec2,b(2:end));
fprintf(fileID,'};\n');
fclose(fileID);
```

- Filter Implementation

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

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

                                                        LAB 4: Real-time Implementation of FIR Fil

                                                ********* 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>
// Inculded so that we could declare an integer of certain number of bit
#include<stdint.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>

//FIR filter coefficients
#include "fir_coef.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                              SETT
                        /*************************************************************************/\
    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;

/******************************* Global Variables ********************************/
//define the number of coefficients
#define N 256
//define input and output samples
short sampin;
double samp;
//define a buffer to store input samples, N = filter order + 1 and initialises to zero
short x[N] = {0};
//define a buffer of twice the size for faster implementation
short y[2*N] = {0};
//define a pointer for circular buffer, initializes it so that it points to the last element
int ptr = N-1;
//define a pointer for circular buffer in reverse order, intializes it so that it points to the firs
int ptr1 = 0;
//define a pointer that points to the middle of the double size circular buffer
int ptr2 = N;
//define a pointer which is an unsigned 8-bit integer so that it goes automatically to 0 after reach
uint8_t ptr3 = 0;



 /******************************* Function prototypes *********************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC (void);
void non_circ_FIR(void);
void circ_FIR(void);
void circ_FIR_faster1(void);
void circ_FIR_faster2(void);
void circ_FIR_faster3(void);
void circ_FIR_faster4(void);
void circ_FIR_faster5(void);

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


      // initialize board and the audio port
  init_hardware();
   // initialize the look-up table

  /* 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 de
        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)
{
            samp = 0;
                sampin = mono_read_16Bit();
                //mono_write_16Bit(mono_read_16Bit());
                //do a (N-1)th-order MAC with FIR Filter coefficients
                non_circ_FIR();
                //circ_FIR();
                //circ_FIR_faster1();
            //circ_FIR_faster2();
                //circ_FIR_faster3();
                //circ_FIR_faster4();
                mono_write_16Bit(samp);

}
```

```
void non_circ_FIR(void){

        int i;
        int j;
        for (i = N-1; i>0; i--)
        {
                        x[i] = x[i-1]; //Previous samples moved along buffer from lower
                                                        //element to next higher
        }
        x[0] = sampin; //current input sample stored at index 0
        for (j = 0; j<N; j++) //for loop to compute convolution sum
        {
                samp += b[j]*x[j]; //MAC operation
        }
}

void circ_FIR(void){
        int k;
        x[ptr] = sampin; //current input sample stored at the position where ptr points to
        for (k = 0; k<N; k++)
        {
                if(ptr+k > N-1)
                {
                        samp += b[k]*x[ptr+k-N]; //Allows convolution sum of consecutive samples wei
                }
                else{
                        samp += b[k]*x[ptr+k]; //MAC operation
                }
        }
        ptr = (ptr+2*N-1)%N;//pointer update using modulus operator
}


void circ_FIR_faster1(void){
        int n;
        x[ptr1] = sampin; //Current sample stored at the position where ptr1 points to
        for (n = 0; n<N; n++)
        {
                if(ptr1-n < 0)
                {
                        samp += b[n]*x[ptr1-n+N]; //Allows convolution sum of consecutive samples at
                }
                else{
                        samp += b[n]*x[ptr1-n];//MAC operation
                }
        }

        ptr1 = (ptr1+1)&255; //pointer update using bit-wise and operator

}
void circ_FIR_faster2(void){
        int k;
        int index;
        index = ptr1;
        x[ptr1] = sampin; //Current sample stored at the position where ptr1 points to
        for (k = 0; k<N; k++, index--)
        {
                if(index <0)
```

```
                {
                        index += N; //Allows convolution sum of consecutive samples weighted by filt
                }
                samp += b[k]*x[index]; //MAC operation
        }
                ptr1 = (ptr1+1)&255; //pointer update using bit-wise and operator

}


void circ_FIR_faster3(void){
        int m;
        int index;
        index = ptr2;
        y[ptr1] = sampin;//store current input into first circular buffer of size N
        y[ptr2] = sampin;//store current input into second circular buffer of size N
        for(m=0; m<N; m++, index--)//convolution sum of consecutive samples weighted by filter coeff
        {
        samp += b[m]*y[index]; //MAC operation
        }
        ptr1 = (ptr1+1)&255;//ptr1 goes back to 0 when reaching N-1, otherwise increases by 1
        ptr2 = ptr1+N;//ptr2 is always larger than ptr1 by N
}

void circ_FIR_faster4(void){

        int m;
        int index,index1;
        y[ptr2] = sampin;//store current input into first circular buffer of size N
        y[ptr3] = sampin;//store current input into second circular buffer of size N
        index = ptr2;//index of input sample used in current calculation
        index1 = ptr3+1;//index of its symmetric pair
        for(m=0; m<(N/2); m++, index--,index1++)//convolution sum of consecutive samples weighted by
        {                                                                                //loop from
        samp += b[m]*(y[index]+y[index1]);//use symmetry properties
        }
        ptr3++;//ptr3 overflows after reaching N-1
        ptr2 = ptr3+N;//ptr2 is always larger than ptr3 by N
}
```