RTDSP Lab 4

Design of a FIR filter

Loek Janssen – lfj10@ic.ac.uk

Sebastian Grubb – sg3510@ic.ac.uk

Contents

Filter Design Using Matlab

To start, a filter complying to the specifications was designed in Matlab. The specifications being to have a pass-band filter between 450Hz and 2000Hz with ripple of less than 0.4 dB.

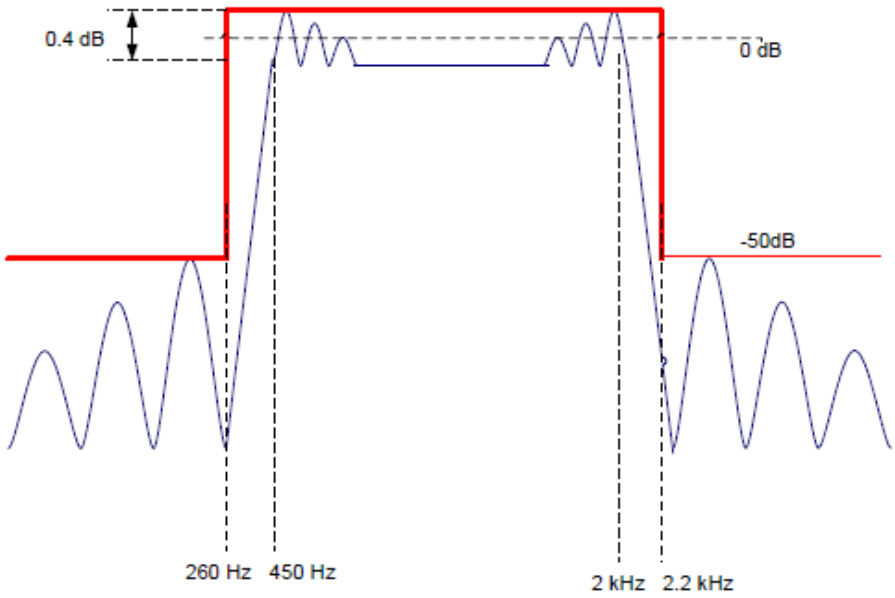


Figure 1. Figure of the specifications for the filter.

To achieve this the firpm and firpmord functions were used in matlab to first determine an approximately suitable order for our filter, as well as other parameters (using firpmord). Then by using firpm we can get the coefficient values that will be used for the filter.

The following code was used to perform this:

```
clc; %clear screen
f = [260,450,2000,2200]; %define ranges
fs = 8000; %define sampling frequency
a = [0,1,0]; %define a
rp = 0.4; %define ripple
sa=-50; %define cut off in dB
dev1 = (10^(rp/20)-1)/(10^(rp/20)+1);  %define deviation1
dev2 = 10^(sa/20); %define deviation1
dev = [dev2 dev1 dev2]; % store
[N,Fo,Ao,W]=firpmord(f,a,dev,fs); %get filter parameters
B=firpm(N+8,Fo,Ao,W); %get values for filter coefficients
freqz(B) %plot
%calculate and display ripple
[h,w]=freqz(B);
disp(['The ripple is:' ,num2str(max(10*log(abs(h(75:220))))-min(10*log(abs(h(75:220)))))]);
save coef.txt B -ASCII -DOUBLE –TABS %save coefficients
```

Notice that we are passing N+8 on to the coefficient generating function. This is because the ripple was originally found not to be within the specified boundary, exceeding 0.4dB as seen in Figure 2. Thus the filter order was increased to 92 which set the filter to be within the correct boundary for ripple requirements, with Figure 3 showing the expected frequency response and how it fits the specifications.

A small section of code was added to the script to calculate the ripple by isolating the indexes relating the values in the ripple section and calculating the maximum difference, giving a precise value of the ripple.

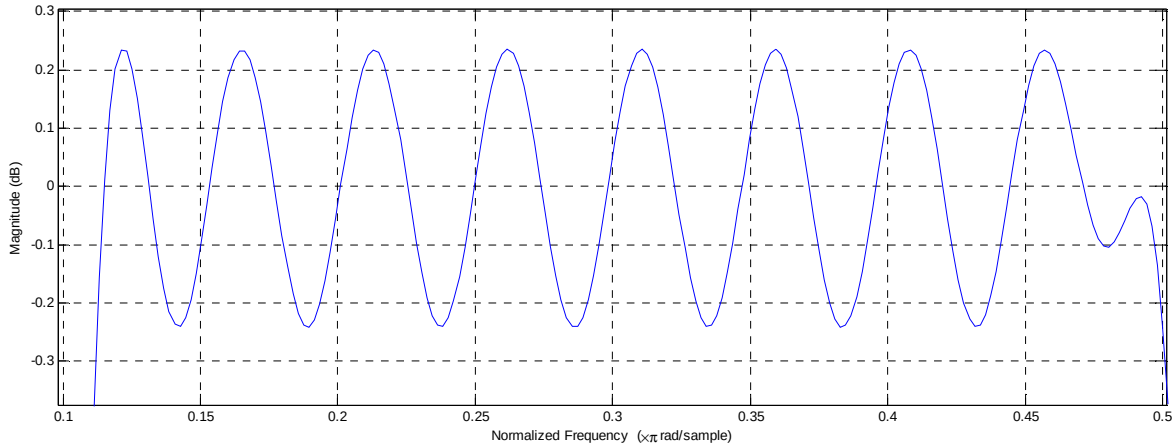
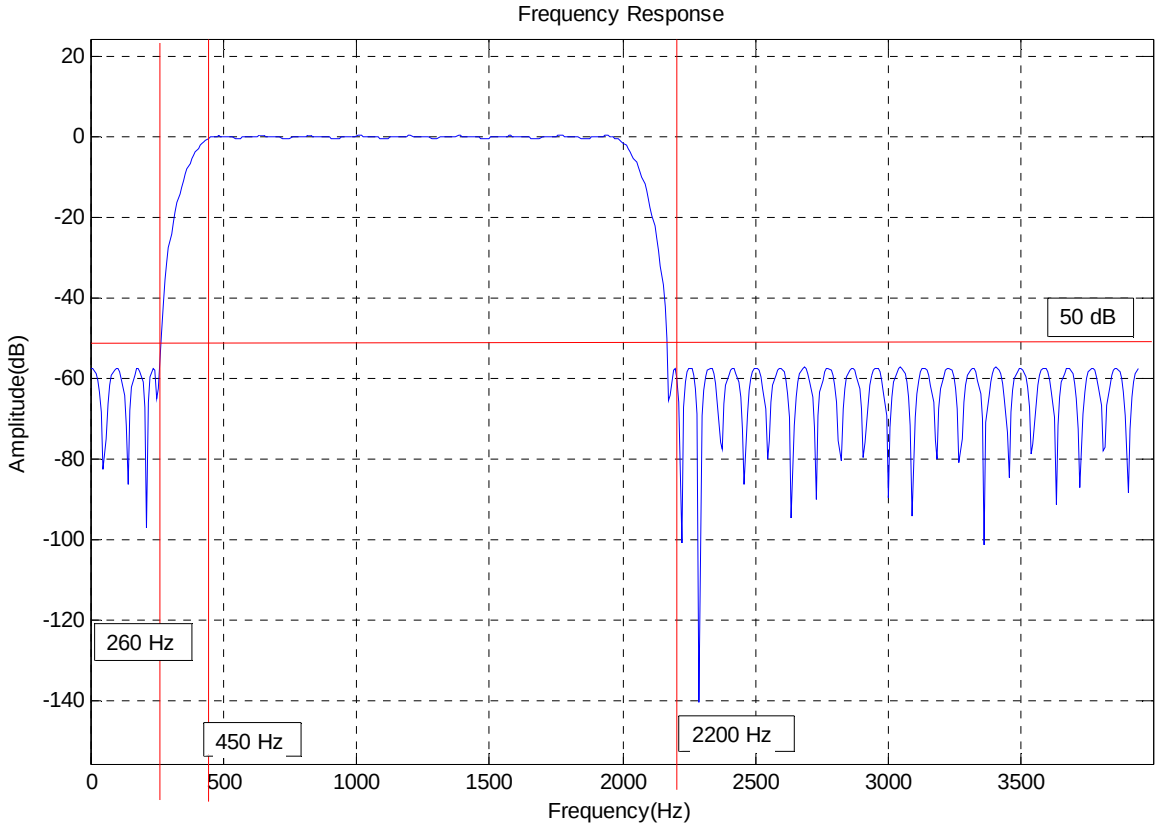


Figure 2. Plot of filter ripple with filter of order 84, clearly showing how it exceeds the requirements of 0.4 dB ripple.



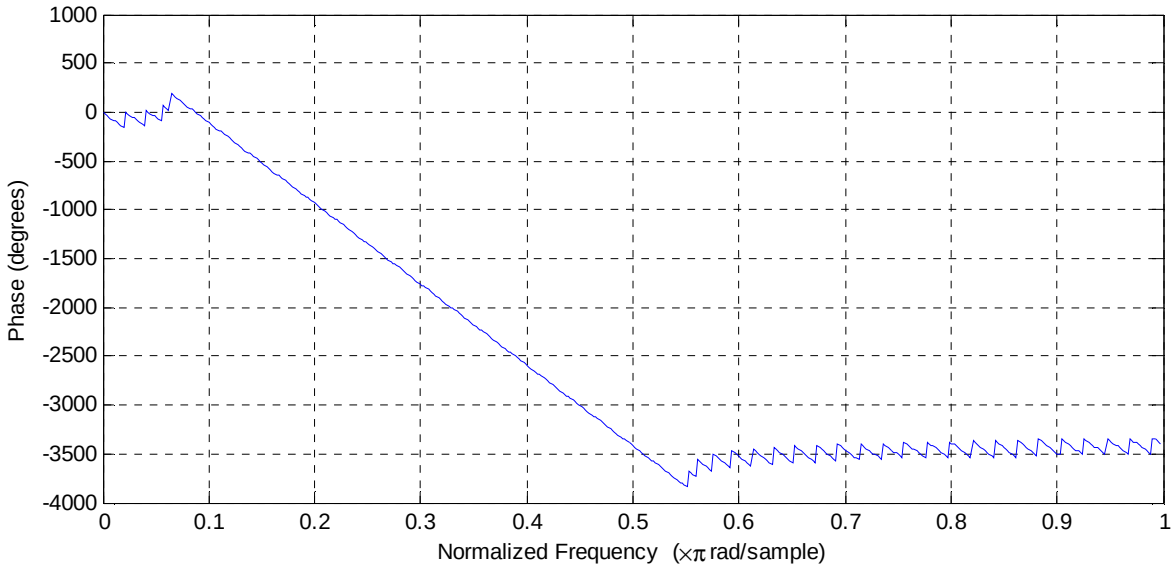


Figure 3. Plot of filter of order 92. The red lines represent the filter requirements, showing that they are met. The phase diagram shows that we are effectively dealing with a linear phase filter.

The expected ideal filter ripple was 0.31617 dB for a filter of order 92.

Non-circular buffer filter implementation:

The ISR programme written for lab3 was copied over, the intio.c file was then altered to contain a delay buffer of 93 elements as a filter of N order will have N+1 taps. Within the interrupt receive the non_circ_FIR() function is called and then the global variable 'output' (which is varied within non_circ_FIR()) is written to the audio port.

```
/******************************** ISR_AIC*******************************/
void ISR_AIC(){

        //call non_circ_FIR which access the sample and implements convolution
        non_circ_FIR();

        /*the global variable 'output' (which was varied in non_circ_FIR())
         * is written to the audio output port*/
        mono_write_16Bit(output);
}
```

The code for the non_circ_FIR function is shown below. This function accesses the sample input at time of interrupt, place it into the end of our buffer and then convolutes all the values in the buffer with the result placed in the global variable 'output'. The values in the buffer are then shifted along by one to allow a new sample to be placed in x[0], with the sample taken 92 interrupts ago removed so the filter is of the correct and chosen order size for an FIR design. After which the function would end (returning void) and then, as stated above, 'output' is written to the audio output port.

```
double non_circ_FIR(){
        //this functions performs convolution of an input sample with
        //coefficient values and past inputs in a non-circular buffer
        //implementation
        //reset output, a global variable, to zero
        output = 0;
        x[0] = mono_read_16Bit(); // place new sample into buffer
        //perform convolution by looping through all past samples and multiple them
        //by their respective coefficients
        for (i=0;i<N;i++){
                //accumulate each multiply result to output
                output += b[i]*x[i];
        }
        //shift all elements such that input value can be placed in zero.
        for (i = N-1;i>0;i--)
        {
                x[i]=x[i-1]; //move data along buffer from lower
        } // element to next higher
        return output; //return output value
}
```

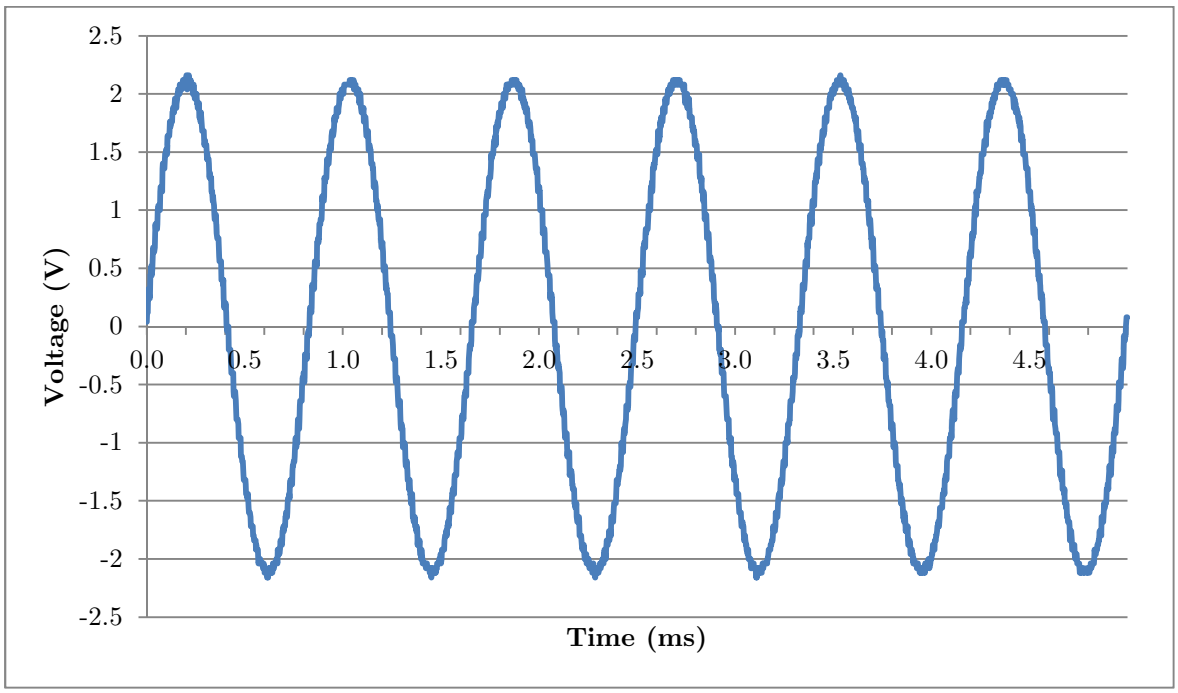
Performance

The code was then tested to see its performance in terms of instruction cycles, with this being measured from the beginning of the interrupt until the exit of the interrupt, hence including calling the function and reading in and writing the sample. The results for testing at the various levels are shown in table 1 and demonstrate the slight improvements of basic optimisation and then the sudden decrease to 666 cycles for optimisation level 2. These levels of optimisation are explained later with the discussion of how the TI complier

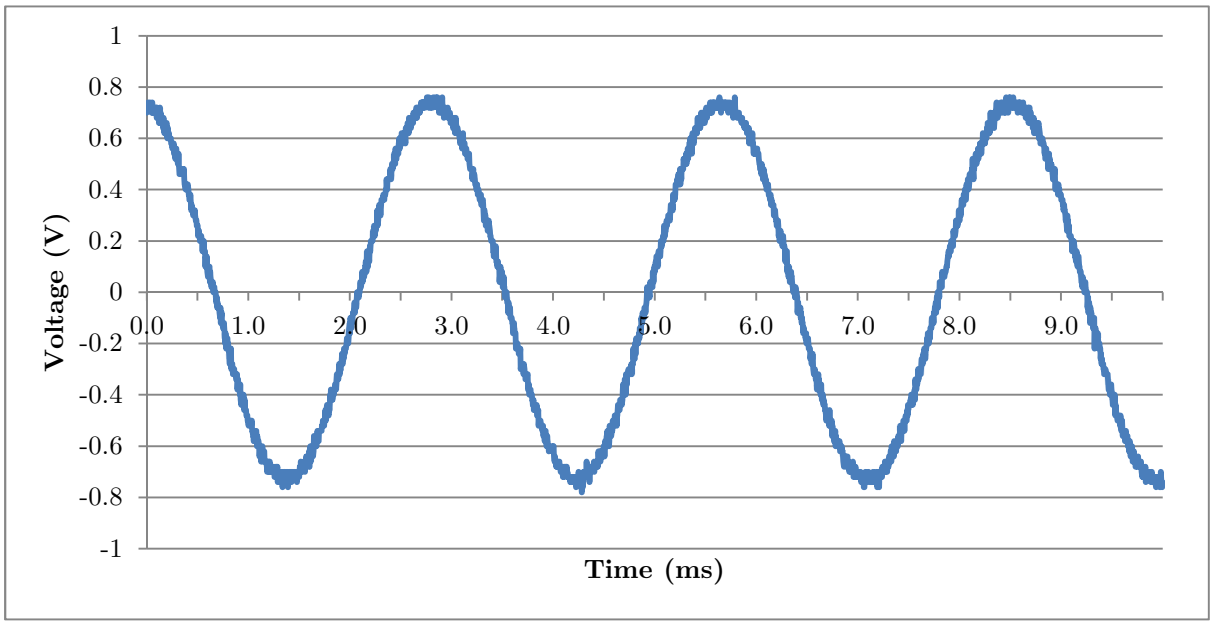
Op-level	Number of cycles
none	5868
0	5590
1	5355
2	705

optimisers the code at different levels.

Scope traces were then taken to check that the operation of the code was correct, with figures below showing the the output of a DSK for 1.2 kHz sinewave input which is roughly in the middle of the passband. As expected the signal carried little noise and reproduced a sinewave of exactly 1.2 kHz. The voltage input is not adjusted for all other scope traces.



A 350 Hz input sinewave resulted in the figure below. With the same input voltage we can see a decreased peak-peak voltage from roughly 4V to just under 1.6V. This is because the 350 Hz sinewave is placed in the transition band and so while suffering some attenuation the signal is still able to produce an output signal with amplitude greater than noise. However the wave can be seen to be slightly nosier due to it being of smaller amplitude and hence suffering from a lower SNR. This is similar to sinewave inputs between 2050 and 2130 Hz. Below 350 Hz and above 2150 Hz the attenuation prevented a readable output on the scope due to noise.



Circular FIR Filter Implementation in C

This section details the implementation of 3 different circular buffer FIR functions in C. The main idea behind this is to implement a FIR working in the following manner:

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

Where $index$ is the current location of the most recent input sample. It increases or decreases each loop (depending on the implementation) thus avoiding any need for shifting all values in the buffer which is an inefficient operation.

$b[i]$ contains all the b coefficients

$x[i]$ is the buffer that stores all the past values.

$y[n]$ is the output

N is the number of taps, i.e. the order of the filter+1.

A modN is included to take into account the fact that overflow may occur. Additionally different implementations are of the form $y[n] = \sum_{i=i}^{N-1} b[i] \times x[index - i]_{mod(N)}$wherein the index decreases instead of increasing.

Base Case

The first function is implemented by simply using a circular buffer implementation which loops back once the index overflows.

```
double base_circ_FIR(){
        x[index] = mono_read_16Bit(); //store in x[index] the input
        output = 0;                                          //reset output
        //loop from 0 to N-index which is the point just before
        //which overflow occurs. This avoids an
        //if function inside a for loop
        for(i = 0;i<N-index;i++){
                //simply accumulate the convolution result
                output+=x[index+i]*b[i];
        }
        //loop from N-index to N which when
        //overflow occurs. This handles the overflow.
        for(i = N-index;i<N;i++){
                output+=x[index+i-N]*b[i];//simply accumulate result
        }
        index++; //increase index
        if (index==N){index=0;} //check for overflow
        return output; //return value
}
```

In this lab the index increases, storing the next value in index+1. Two for loops are used to take into account the fact that when i (the looping variable) reaches $N - index$, overflow occurs in the sense that a non-existing index is attempted to be accessed. For example if index = 20, N=93 and i reaches 73 we naturally get an overflow, by trying to access $x[93]$ which does not exist. Thus we enter a new loop which fixes this problem by subtracting it by N. This method avoids an if statement inside a for loop or alternatively avoiding a mod function, both which are inefficient as they execute a check each cycle.

Using FIR properties

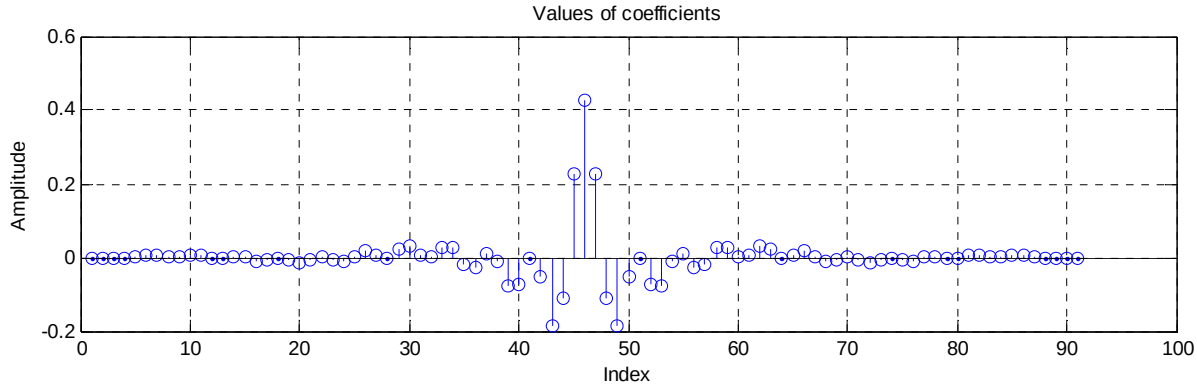


Figure 4. This graph shows the values of B to illustrate the symmetry of FIR filters. This is also the impulse response of our filter.

We know that any linear phase FIR filter has symmetric coefficients (or anti-symmetric coefficients) which means that $b[i] = b[N - i - 1]$, which is true for the coefficients created in matlab. Thus it is possible to take advantage of these properties and perform a for loop half the size, by using factorisation properties. Thus a loop would only have to loop $\frac{N-1}{2}$ times (if N is odd).

i, index=0	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
index+i, x cell	0	1	2	3	4	5	6	7	8	9	10	11
index-1-i, x cell	22	21	20	19	18	17	16	15	14	13	12	11
i, cell of b accessed	0	1	2	3	4	5	6	7	8	9	10	11

i, index = 6	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
index+i, x cell	6	7	8	9	10	11	12	13	14	15	16	17
index-1-i, x cell	5	4	3	2	1	0	22	21	20	19	18	17
i, cell of b accessed	0	1	2	3	4	5	6	7	8	9	10	11

Figure 5. Example of how a filter with N=23 would only need 11 loops to complete a convolution as the output simply is: $y[n] = \sum_{i=0}^{(N-1)/2} b[i] \times \left(x[index + i]_{mod(N)} + x[index - i - 1]_{mod(N)}\right)$, meaning that less loops are done. In these example we use the cases when index=0 and index=6.

As seen in Figure 5 a rough table description of how different coefficients are multiplied with their respective sample values. It is important to note that for odd N values we get $[index + i]_{mod(N)} = [index - i - 1]_{mod(N)}$ which means this has to be handled to avoid a value being used twice. One way to solve this is to halve the middle coefficient; the other way is to have even coefficients which avoid this happening. Finally we can handle the middle coefficient multiplication outside of the loop, which is the method employed in this case.

The code employed avoids the use of any modulo or if function inside a for loop due to efficiency concerns.

```
double circ_FIR(){
        x[index] = mono_read_16Bit(); //read input and store in correct cell
        output = 0; //reset output
        //check where index is to determine which values will overflow
        if (index <= (N-1)/2){
```

```
                //handle the case where index -1 - i may overflow.
                //first loop which handles all values without overflow
                for (i=0;i<index;i++){
                        //performs factorized convolution
                        output += (x[index-1-i]+x[index + i])*b[i];
                }
                //second loop handling the overflow
                for (i=index;i<(N-1)/2;i++){
                        output += (x[index-1+N-i]+x[index + i])*b[i];
                }
                //handles the middle coefficient separate to avoid doubling
                //a component of the output. This is only necessary for odd
                //N values.
                output += (x[index + (N-1)/2])*b[(N-1)/2];
        }else
        {
                //handle the case where index + i may overflow.
                //first loop
                for (i=0;i<N-index;i++){
                        //performs factorized convolution
                        output += (x[index-1-i]+x[index + i])*b[i];
                }
                //second loop
                for (i=N-index;i<(N-1)/2;i++){
                        //performs factorized convolution
                        output += (x[index-1-i]+x[index - N +i])*b[i];

                }
                //handle odd case
                output += (x[index-(N-1)/2 - 1])*b[(N-1)/2];
        }
        index--;//decrease index
        if (index<0){index=N-1;}//handle overflow of index
        return output; //return output
}
```

Using double the memory

Using the symmetry properties of the FIR coefficients also requires a certain amount of branching and processing to check for overflow. To avoid any processing due to overflow checking we can instead choose to use double the memory by having two indexes. The idea would be to have one index always within the range of 0 to N-1 and the other index2=index+N. This would then allow convolution to be done as follows (notice the lack of mod):

$$y[n] = \sum_{i=0}^{N-1} b[i] \times x[index + i]$$

This essentially allows our double buffer method to work in a very similar way than the non-circular implementation by simply adding an offset of index. Since we are also storing values at index+N no access overflow will happen, avoiding extra computation. Figure 6 illustrates how this memory allocation can be done.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
	index				index2		

Figure 6. Example of how a double memory use would work for storing values in the x array (the buffer) for the case of N=4, thus when index = 1, index2 =5. The green region represents all values x can take and the blue region all values index2 can take.

We also use the fact that coefficients are symmetric and use the following equation to define our algorithm:

$$y[n] = \sum_{i=o}^{\frac{N-1}{2}} b[i] \times (x[index + i] + x[index2 - i - 1])$$

$$+ b\left[\frac{N-1}{2}\right] \times x[index + \frac{N-1}{2}]$$

Again this is designed for the case where N is odd and explains the addition of the extra term at the end which handles the case for the middle coefficient.

```
double doublesize_circ_FIR(){
    //read input and place in memory
    //this is more efficient than reading input twice
    sample = (double)mono_read_16Bit();
    x[index] = sample; //store sample in its first index
    x[index2] = sample; //store sample in its second index
    output = 0; //reset output
    //loop from 0 to N-1/2
    for(i=0;i<(N-1)/2;i++){
            //use symmetry properties and perform convolution
            output += b[i]*(x[index+i]+x[index2-i-1]);
    }
    //use for the case where N is odd
    output += b[(N-1)/2]*x[index+(N-1)/2];
    index++; //increase index
    if (index==N){index=0;} //handle overflow
    index2 = index+N; //make index2 follow index
    return output; //return output value.
}
```

Comparison of different circular buffer methods

All 3 methods have their advantages. The first one being that it is simple, the second one in using the symmetry properties of the coefficients to realise better performance and the third one for being the fastest (while using double the memory though).

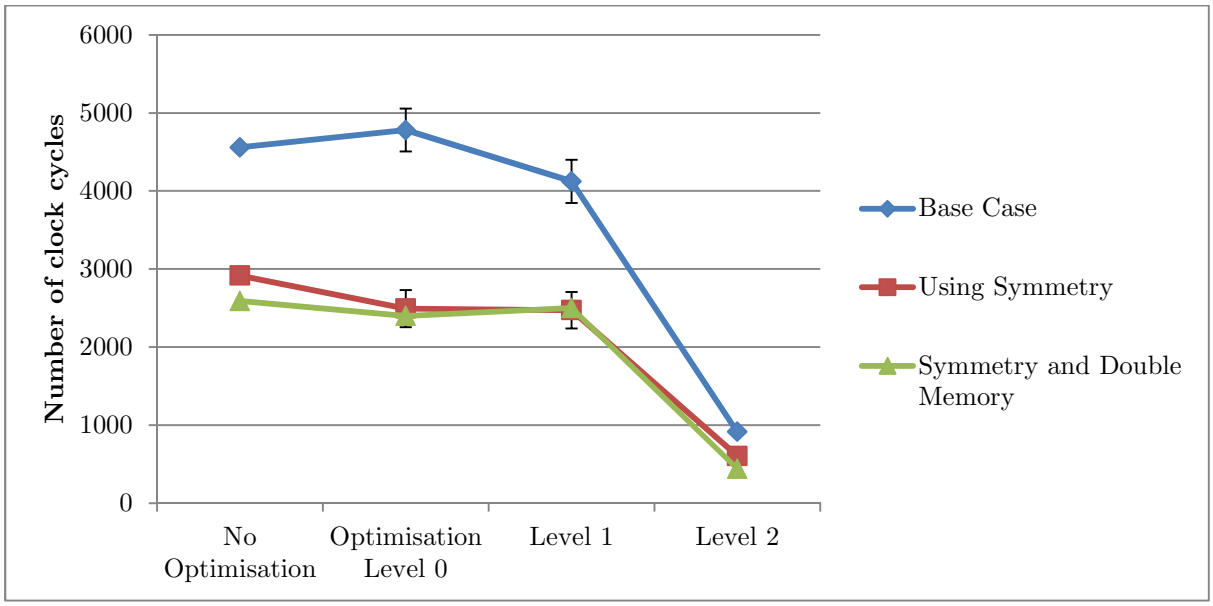


Figure 7. Comparison of the efficiency of the three different circular methods.

	Base Case	Using Symmetry	Symmetry and Double Memory
No Optimisation	4559±26	2915±89	2590±6
Optimisation Level 0	4780±275	2492±238	2400±7
Level 1	4123±276	2472±232	2500±5
Level 2	914	604	441

Figure 8. Comparison of the efficiency of the three different circular methods. Some code showed variance in the way the number of cycles explain the boundaries provided for certain cases. *Note: Clock cycles counted are for the entire ISR function thus including* `mono_read_16Bit()` *and* `mono_write_16Bit()`*, two clock intensive operations*

Figure 7 and 8 show how using symmetry properties of the FIR drastically help in terms of efficiency and that using a double memory buffer does in fact help in a minor way to reduce the number of calculations.

These figures also allow us to see how aggressive each optimisation level is.

- Optimisation level 0, which mainly performs control-flow graph optimisation as well as eliminating unused code while simplifying user written expressions, does not necessarily fare that well in optimising as it sometimes even creates a worse clock count.
- Optimisation level 1 does all level 0 optimisations while optimising for local copy propagation and constant propagation, i.e. checking whether a constant could be used instead of a variable which may improve the code. It also removes unnecessary assignments, which can contribute to higher clock cycles. It somewhat helps with the optimisation of our code but only for the basic circular buffer implementation.
- Optimisation level 2 drastically improves our code for the following 3 reasons:

- *Software pipelining*: ensures that, if possible, all 8 processor units are being used at once
- *Performs loop optimisation*: as our functions are mainly loops this help improves the general performance of a loop, for example this optimisation may move the branch instruction higher up in the code to avoid the pipeline being stalled with NOP instructions and rather execute code.
- *Unrolling loops*: as we loop N times (or N-1/2 times) each time a sample is processed, unrolling (which removes lost cycles due do pipeline stalling) will help in this sense.

As in RTDSP the main concern is simply clock efficiency the best method so far is our double memory implementation which would theoretically allow us to sample at up to $500KHz$ $(= \frac{225Mhz}{450})$[1].

We also notice the ranges given for certain optimisations levels. This is most likely due to the loops having different indices at different times and overflowing different and thus optimised in different ways.

[1] The DSK6713 operates at 225Mhz, thus allowing up to 1800 million instructions per second (225M*8=1800M). http://www.ti.com/tool/tmdsdsk6713
450 is the rounding up of 441 cycles of the level 2 optimisation of 441 cycles for the double memory circular buffer.

Analysis of resulting filter

Note: It is important that all 3 methods were checked with the spectrum analyser and outputted exactly the same frequency response as the non-circular buffer implementation – implying that all MAC operations were carried out correctly as omitting one or two coefficients can give similar cut-offs but with irregular roll-off regions (i.e. it would be difficult to determine on an oscilloscope).

Using a spectrum analyser and the provided software the frequency response of the implemented filter was obtained and is shown in Figure 9.

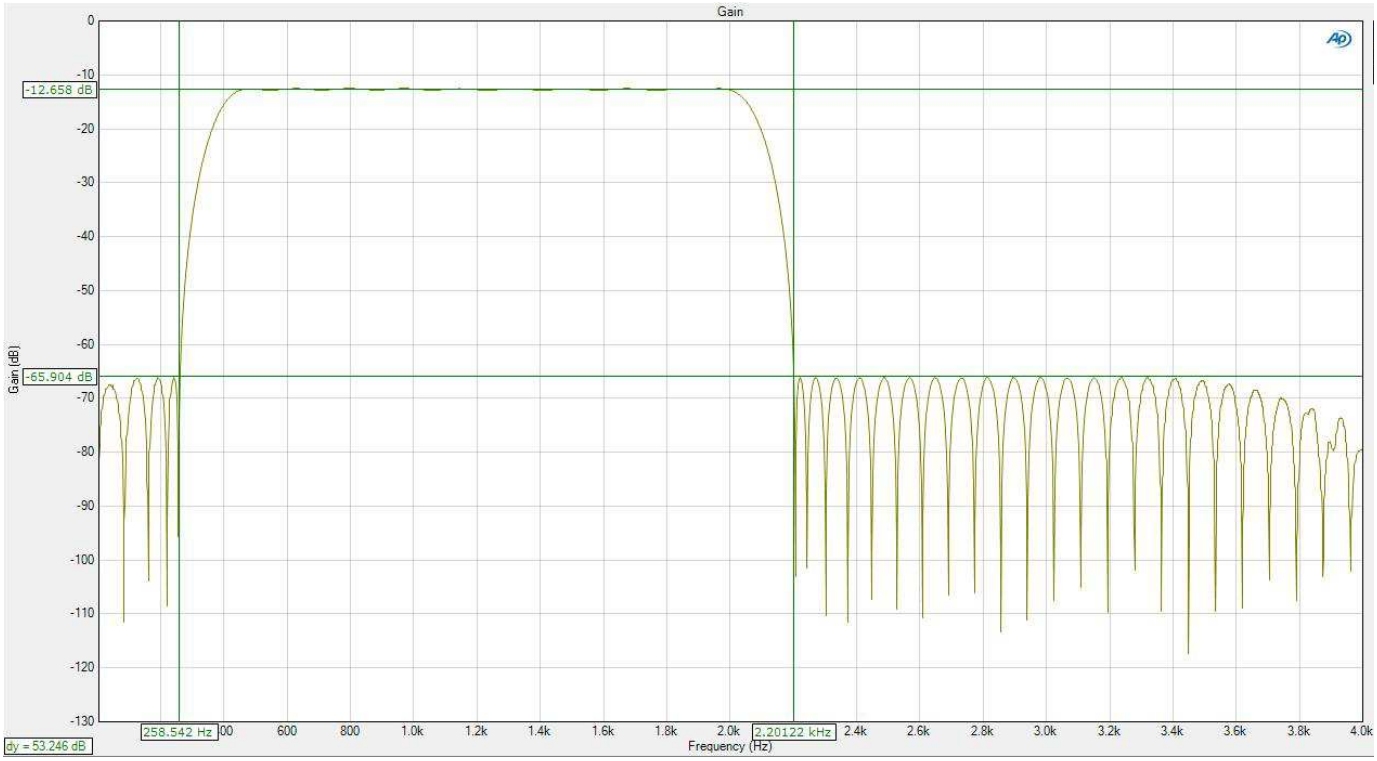


Figure 9. Graph of filter obtained with the spectrum analyser for the designed FIR filter. The circ_FIR() function was used though any other implementation would have also given the same results.

The stopband is attenuated by just over 50 dB (53.2 dB in this case) as required by the specifications, with the stopband frequencies at 258.5 Hz and 2.20 kHz which fall very close to our specified frequencies of 260 Hz and 2.2 kHz respectively. The resulting frequency response is very close to that predicted by the ideal filter design in matlab. This can be seen when the results from the frequency analyser and matlab (with gain corrections as discussed below) are overlaid as in figure 10, it is a near perfect copy except a very low and very high frequencies.

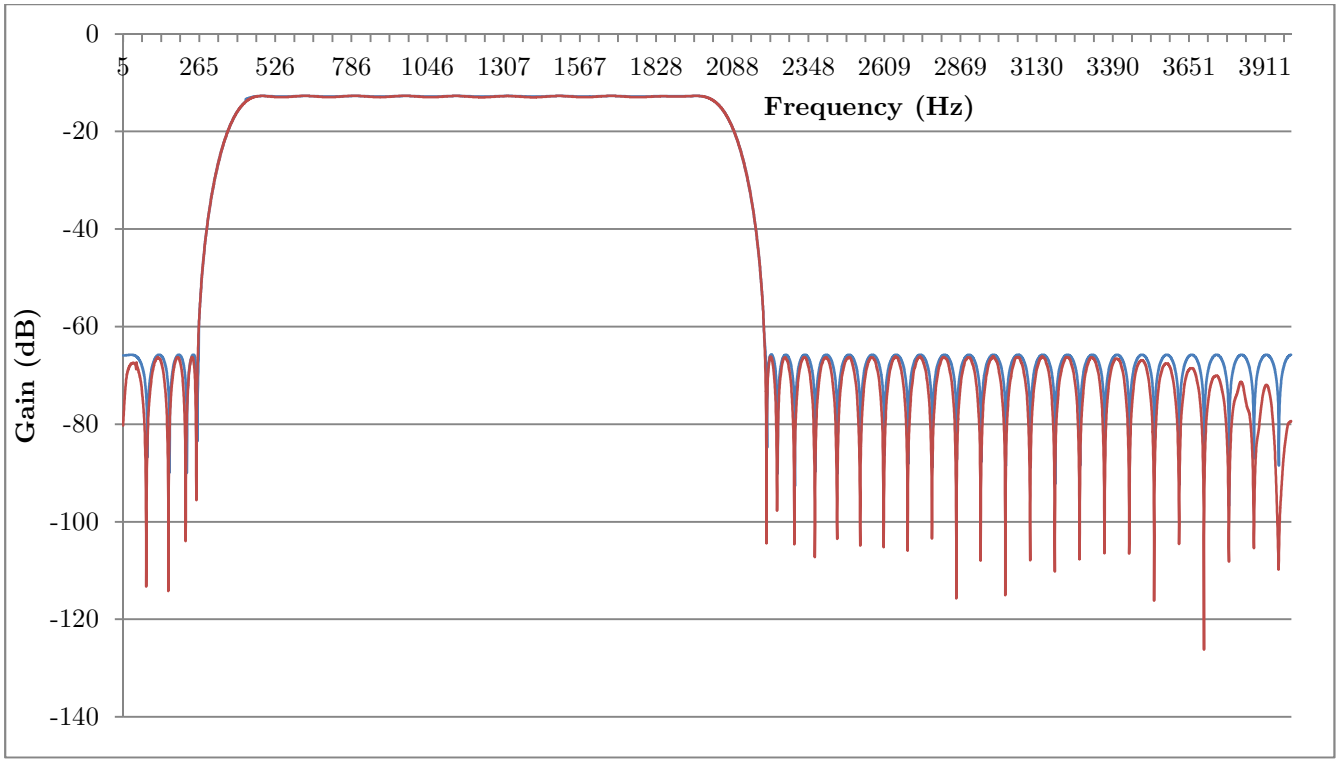


Figure 10. Comparison of ideal filter with filter obtained from the DSK. The red line represents the obtained filter results and the blue line represents the ideal filter as obtained in matlab.
Note: The matlab results were adjusted to take into account the 12dB offset of the measured filter.

As we can see in our spectrum analysis the gain from the passband is $\approx 12\,dB$ rather than the 0 dB that is expected from the initial matlab simulations as shown in Figure 3. This is due to the input into the audio port being halved twice before being used by the DSP processor. It is first halved by the potential divider on the audio chip input. Then when the sample is read using the `mono_read_16Bit()` function the processor takes the average of the right and left line in, hence with only one line in it is halved again. The output will reflect that the input amplitude is quartered and so with output amplitude multiplied by $\frac{1}{4}$ we get an attenuation of [2]

$$20\log\frac{1}{4} = -12.04\,dB\ (3\,s.f)$$

And so explains why the gain is roughly 12 dB below our matlab simulation.

Exploring the low frequency section of the overlays further we can identify that our real filter has an increased attenuation over our matlab simulation as the frequency tends to 0 Hz. This is most likely due to the high pass filter out of the AIC23 codec having a larger affect upon the output than our FIR filter at this stage.

[2] We are using 20log(value) as we are reading an input voltage

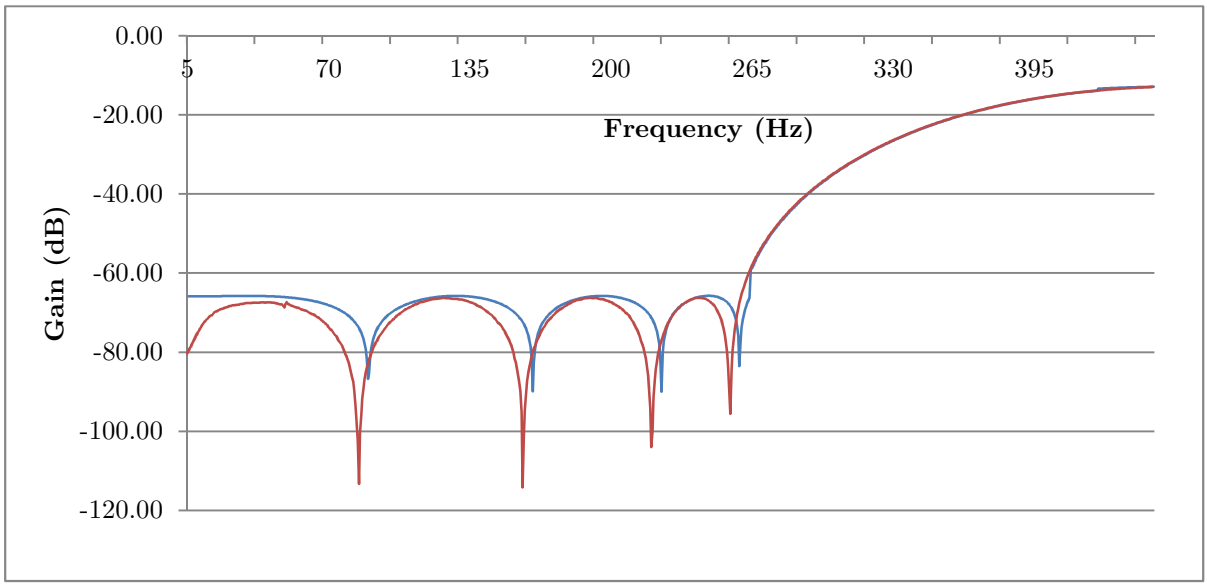


Figure 11. Comparison at low frequencies of the implemented filter. The red line represents the obtained filter results and the blue line represents the ideal filter as obtained in matlab. *Note: The poles may not seem to align in this graph but this is only due to the way the data collection of the APx500 software is done, which samples at irregular frequencies in the low frequency regions thus making it difficult to completely align the collected data with the matlab values. This also explains some of the graph discontinuities.*

Figure 10 shows an attenuation not found the filter design in matlab as the frequency approaches $f_s/2$ (in this case 4kHz). The main cause of this is the low-pass reconstruction filter within the codec which is non-ideal, and hence begins to attenuate the output signal slightly at values above $f_s/2$ as shown in figure Z providing the differences between the real and simulated filter.

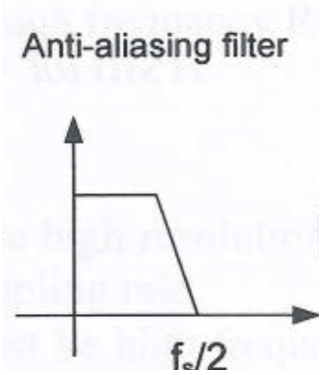


Figure 12. Graph of the non-ideal low pass filter.

To find out the main/"base" filter in the DSK and codec, the following function was run in the ISR:

```
void ISR_AIC(){
        mono_write_16Bit(mono_read_16Bit());
}
```

This was passed through the spectrum analyser to have an idea of what filters affect the DSK in its "base case" where no sample processing is performed.

Figure 13 below shows how at low frequencies attenuation in our FIR filter would be expected, and similarly how from around 3500 Hz extra attenuation is observed in Figure 10 which does not match the matlab values.

10)

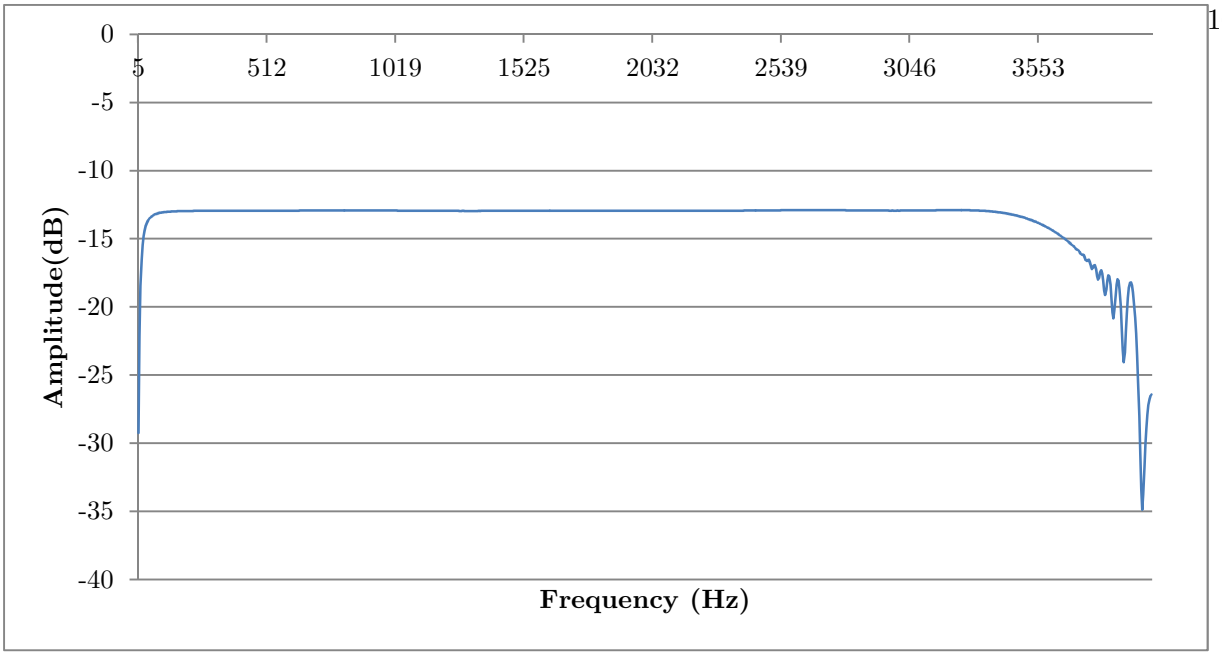


Figure 13. Graph of the DSK and codec "base" frequency response, where a sample is simply read and passed out.

Focusing in on the pass band we find that in figure 14 there is a passband ripple of 0.316 dB which is well within the specified maximum of 0.4 dB and is, within 3 significant figures, the same as the expected ripple calculated by the matlab script (see the first section). The passband frequencies are 451 Hz and 2001 Hz which almost exactly follow the specification of 450 Hz and 2.0 kHz respectively. When compared to our expected response found using the Parks-McClelland algorithm in matlab (shown in figure 15) we can see that as described in the matlab code above we get a voltage ripple of almost exactly 0.316 dB. We also get a very similar (if not exactly the same) ripple waveform, with a kink in the final ripple present in both our simulation and real spectrum analysis.

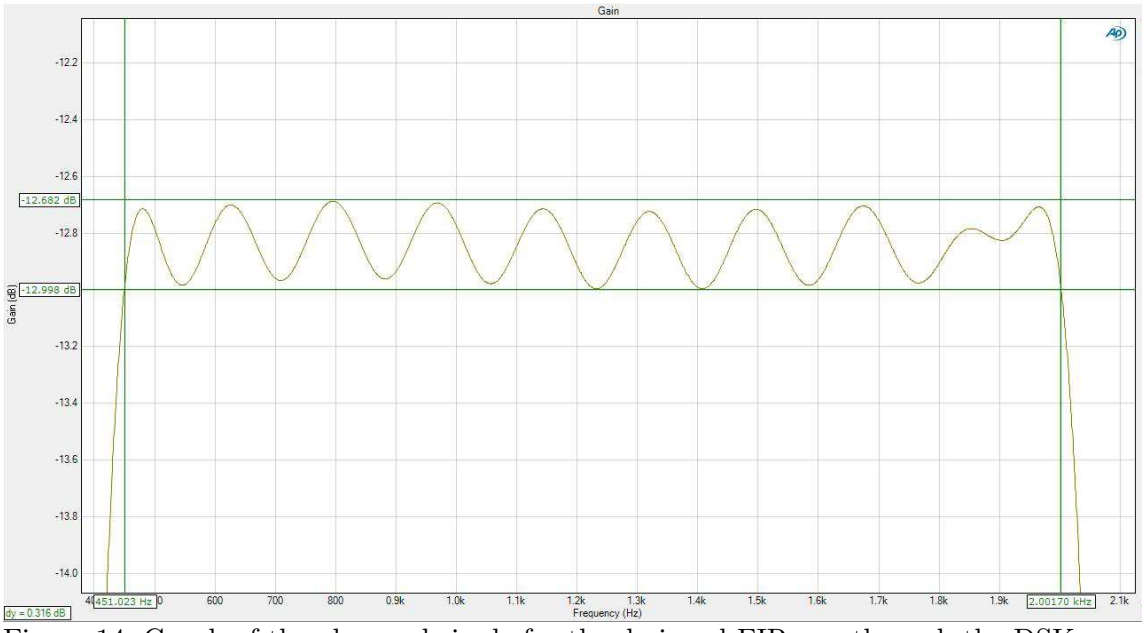


Figure 14. Graph of the observed ripple for the designed FIR run through the DSK.

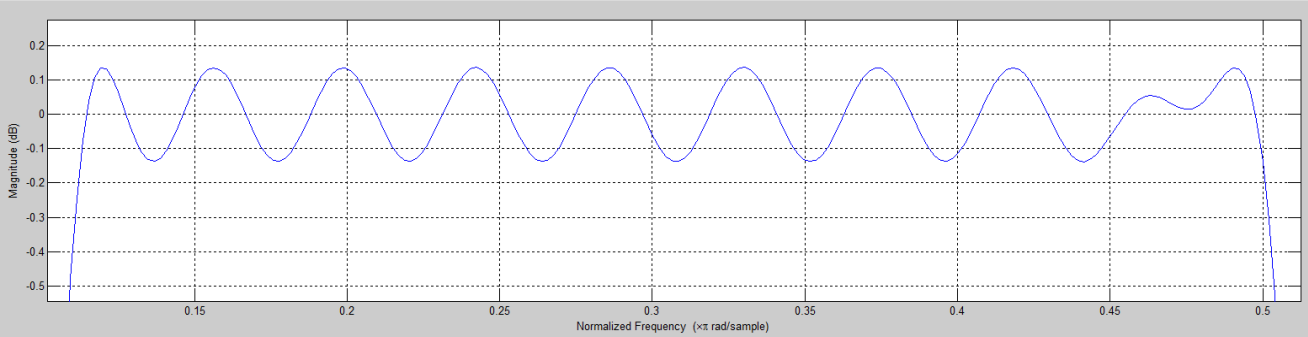


Figure 15. Graph of the ideal ripple for the FIR filter designed in matlab.

The filter bands were also found to comply with the specified filter requirements. Figure 16 shows the first (increasing gain) transition band as measured using the spectrum analyser, and an attenuation of over 50 dB at the first peak of the roll-off is achieved. The stopband frequency is 260 Hz (3 s.f.) exactly as specified (as seen in Figure 1), the passband frequency of 451 Hz is also very close to our requirements.

The second transition band with increasing attenuation is then measured using the network analyser and is shown in figure 17. The passband frequency is identified as 2.00 kHz (3 s.f.) and the stopband frequency as 2.20 kHz (3s.f) this is exactly as required for the filter response.

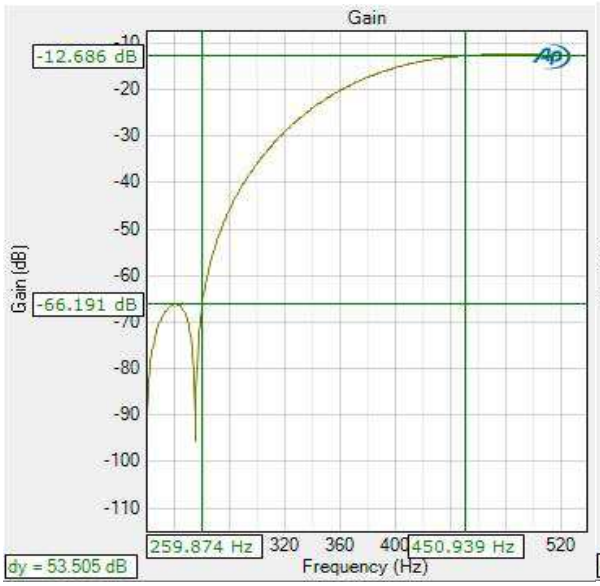


Figure 16. Graph of observed data for the low frequency region of the FIR filter.

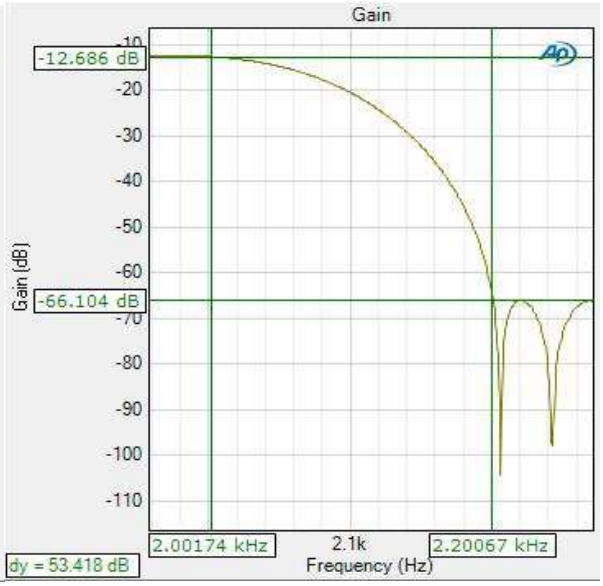


Figure 17. Graph of observed data for cutoff frequency region of the FIR filter.

The phase response of the filter is shown in figure 18 and shows how we have a linear phase delay. The straight line from 220 Hz to 2.2kHz represents the linear phase in the passband, this is also shown in our matlab simulation (figure 19) which is as expected for base frequency response of the DSK. If it was not linear phase we would get different delay times for different frequency components and hence would have phase distortion in our result.

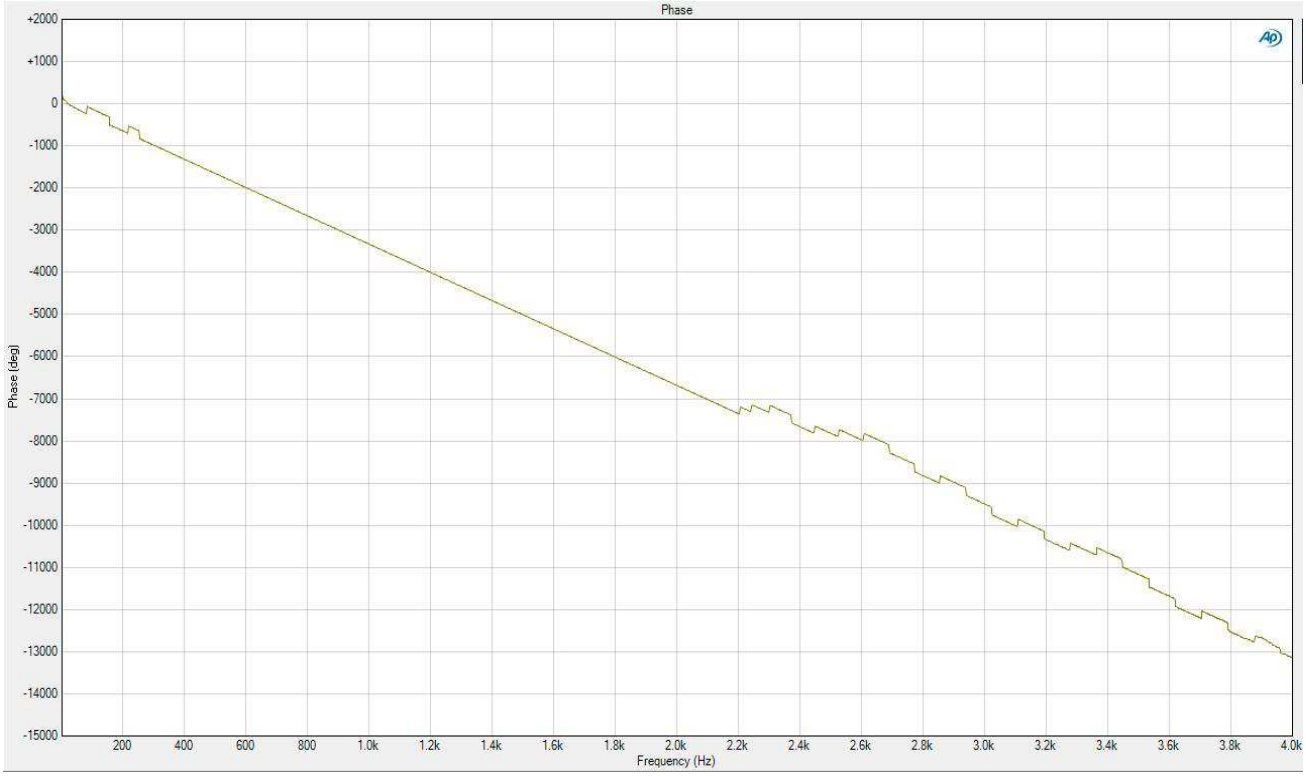


Figure 18. Phase response of the designed FIR filter.

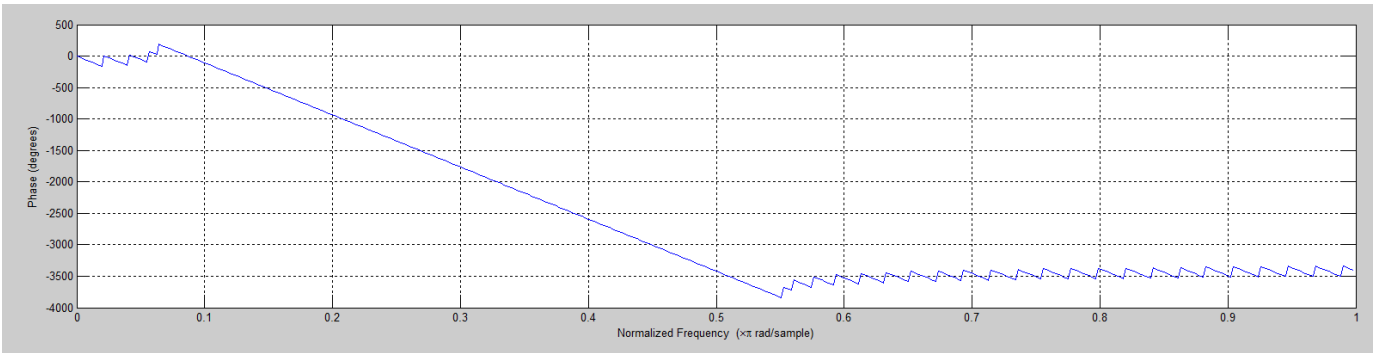


Figure 19. Graph of the idea filter response of the FIR filter designed in matlab.

There are however significant differences between the phase which is measured using the spectrum analyser and our matlab simulation. With a much steeper gradient than expected, dropping to over -7000 degrees as opposed to only -4000 in the matlab simulation. This is caused by the same reasons as the gain attenuation at extreme frequencies, the filters on the input and outputs of the audio chip give an additional phase variation which sums with the phase variation of our specified filter. The change of phase for a unity gain filter is shown in figure X and is also linear phase. For the stopbands in our bandpass filter our simulation gives only a sawtooth wave centred on a fixed phase, we can now see with the addition of the DSK phase change we would get a sawtooth wave with a decreasing centre phase which is the result we get from the frequency analyser.

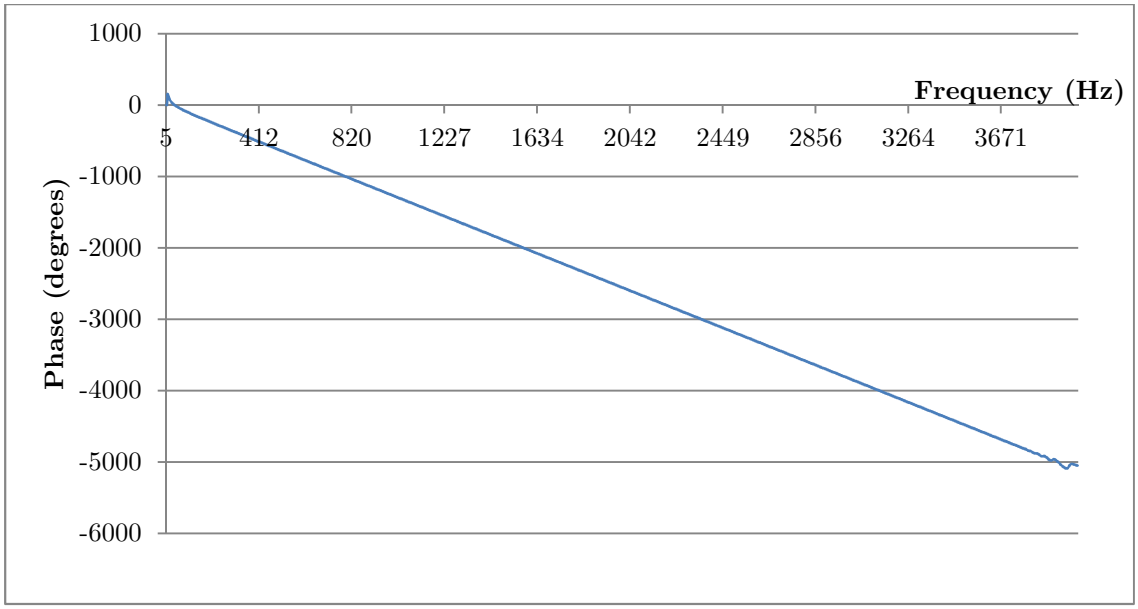


Figure 20. Phase response of the unity gain filter implemented by having `mono_write_16Bit(mono_read_16Bit())` in the ISR function. This phase adds up to the phase response of the ideal filter found in matlab (Figure 19) to end up with the phase response in Figure 18.

Finally the group delay was calculated using the frequency analyser to verify that the filter is perfectly linear phase during the passband. With linear phase we would expect a straight line, which is exactly as measured as shown in figure X. The straight line has a value of 9.316ms which is about 3 times higher than a commercially accepted group delay to reproduce high-quality sound. [3]

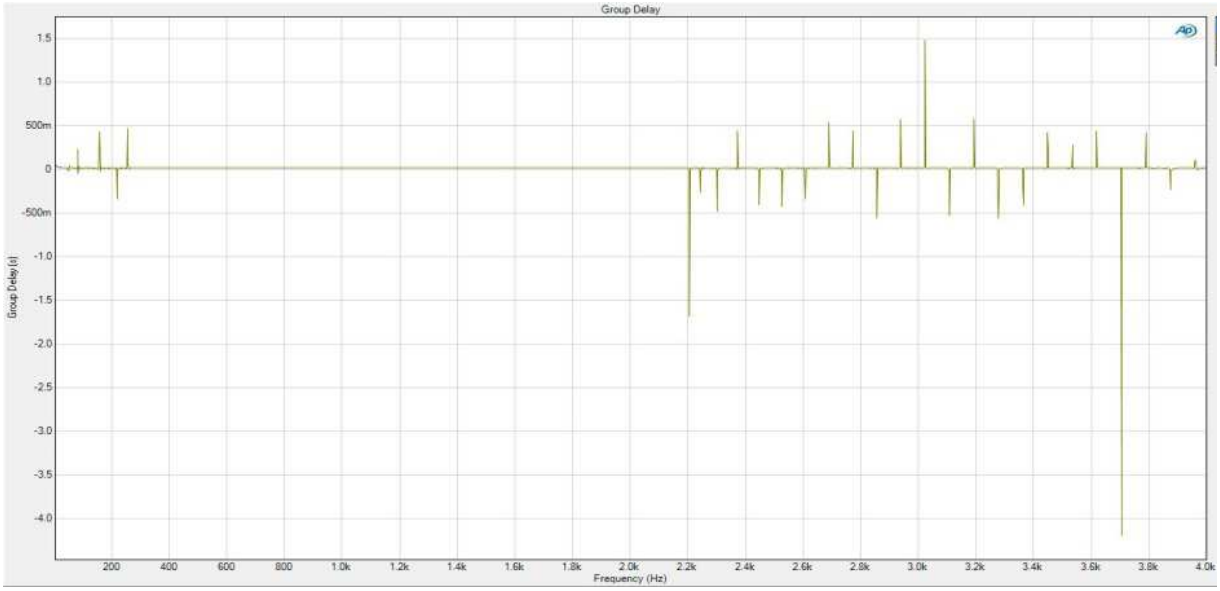


Figure 21. Group delay of the implemented FIR filter. It can be seen that this filter is constant during the passband.

[3] Blauert, J.; Laws, P. (May 1978), "Group Delay Distortions in Electroacoustical Systems", *Journal of the Acoustical Society of America* **63** (5): 1478–1483
This article requires that the group delay for systems be less than 3.2 to 1.5 ms for frequencies ranging from 500hz to 4kHz, respectively.

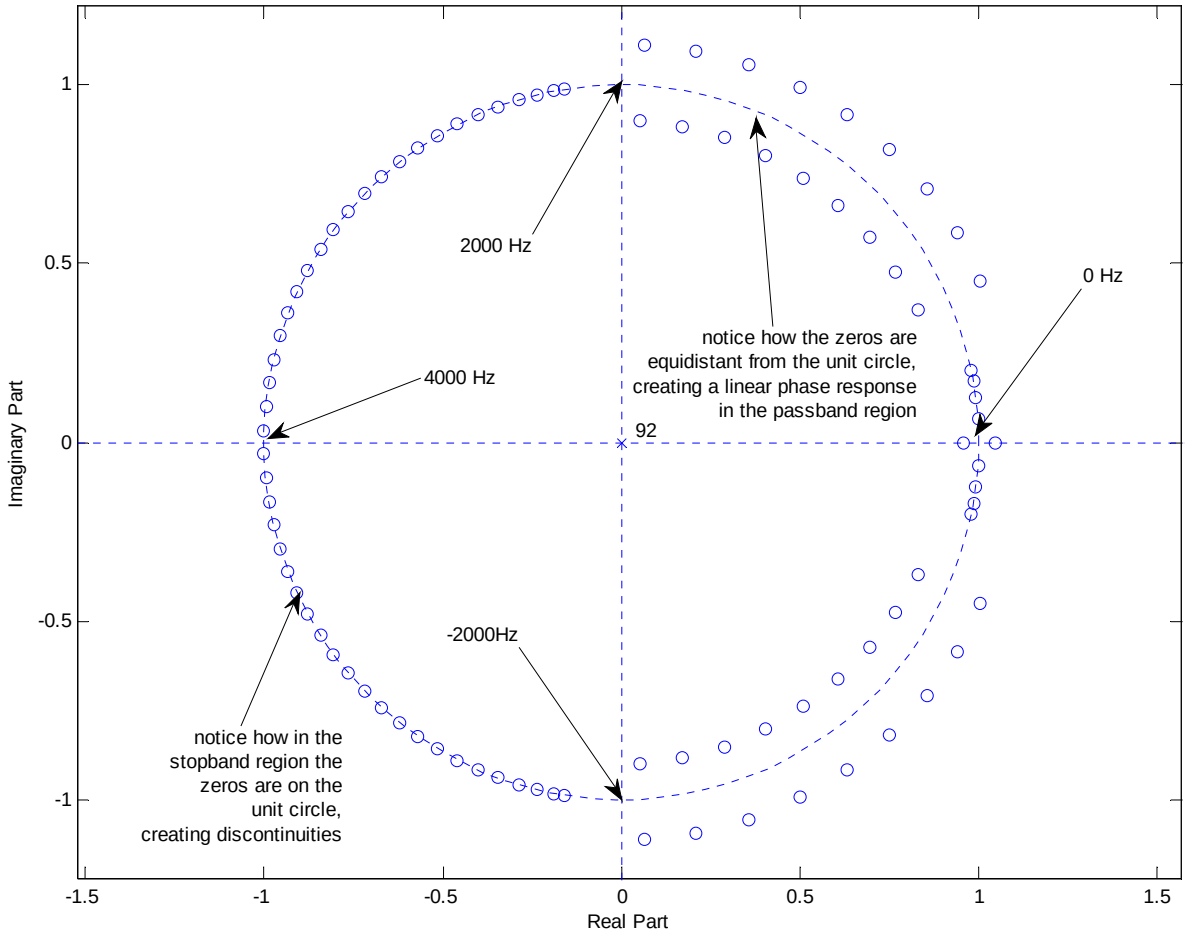
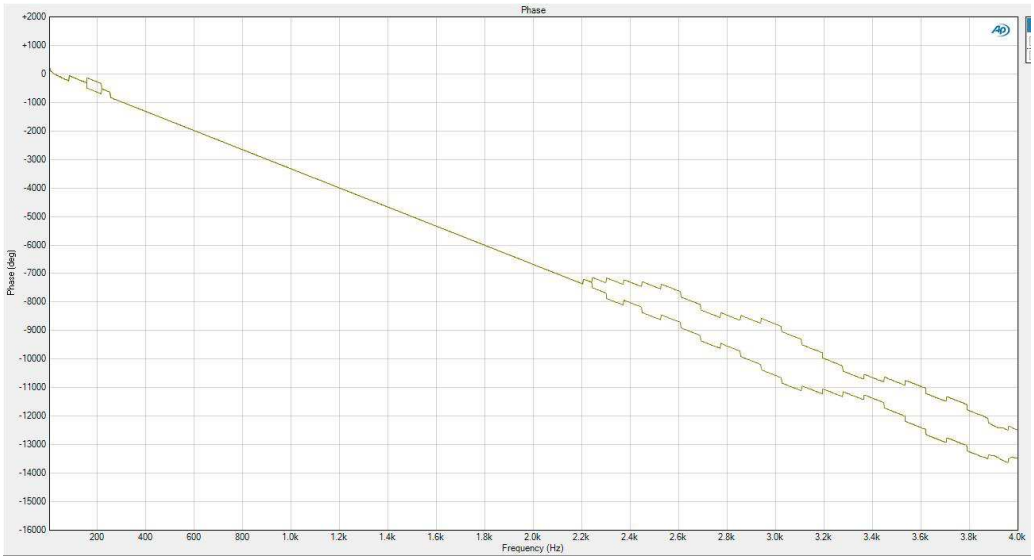


Figure 22. Plot of poles and zero of the transfer function.

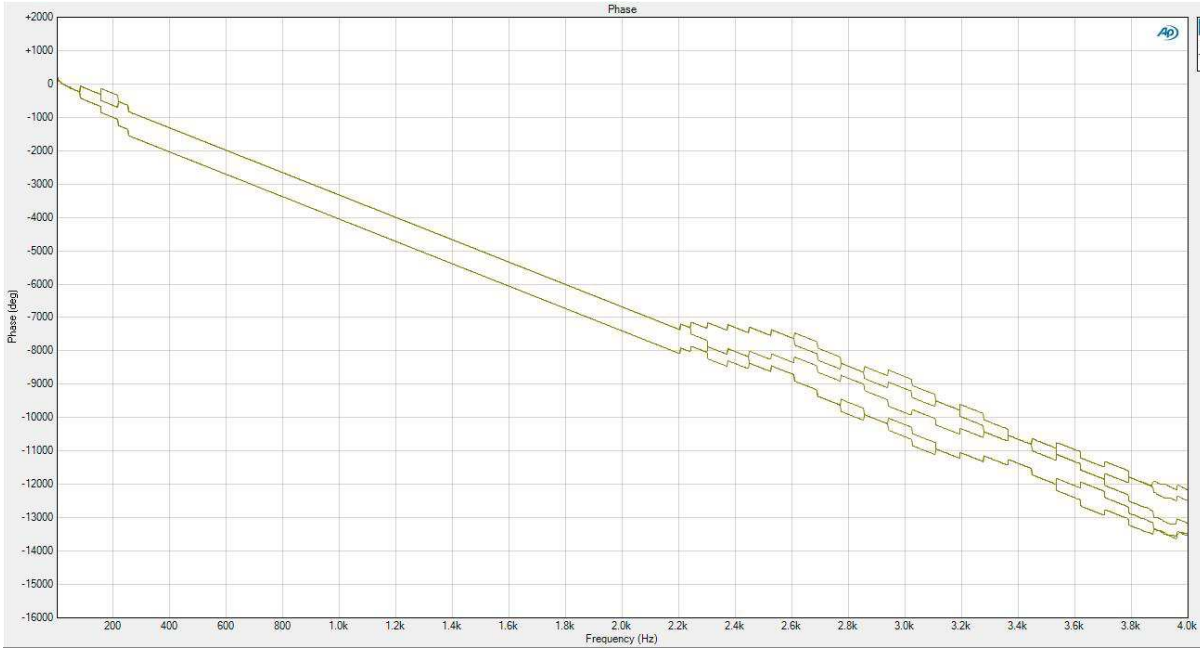
Figure 22 above is a plot of zeros and poles for the transfer function of the filter created using the Parks-McClelland algorithm in matlab. The zeros placed equidistant from the unit circle provide the passband, and act as "centres of gravity" for the gain, the ripple is the pull and tug between these equidistant points and the number of peaks in the ripple is the same as the number of zero pairs. In this case we get 9 zero pairs and we get 9 peaks (see ripple figure) as the final kink is merely the pull of the zeros on the unit circle.



We can also see many circles on the unit circle, these zeros at their frequencies represent the stop-band. However they cannot be placed exactly on the unit circle and through rounding errors and discontinuities introduce variations in phase over the stop-band. This is

why in our matlab phase plot we get discontinuities (a saw-tooth wave) as the frequency moves from zero to zero.

In matlab the discontinuities are all on the same side of the unit circle (causing a shift up at the zero in our graph) but when the phase of our real filter is measured using the frequency analyser we can see that due to rounding errors in the DSK and different starting inputs the zeros are not placed explicitly inside or out of the unit circles. This gives rise to the figure on page 20 where for exactly the same code we get a different phase diagram we the discontinuities shifting up or down at different points. The figure on page 20 shows the difference between two different runs on exactly the same code, DSK etc, the below shows how with 5 different runs we still get different phase responses which switch between matching each other and giving different responses.



Assembly Implementation

Basic Assembly Implementation

With the optimisation of the circular-buffer C code the next stage is to study the implementation of circular buffers in assembly, as some DSPs contain dedicated hardware which allow circular addressing for a register, "so it is no longer necessary to have to wrap the pointer at the end of the buffer" (lab 4 – Dr Mitcheson).

Within the assembly code the circular buffers are set using the "addressing mode register" (AMR), which is loaded by a 32 bit register (in this case B2; see appendix). Using the TI instruction set reference guide I set register A5 (where the pointer is stored) to have circular addressing using the instruction, as bits 2-3 are used to set A5 and 01 sets it to circular addressing using the BK0 field as our block buffer size. The BK0 field is then set to 1024 bytes using the look up table given. The value 1024 was chosen because the number of data elements the buffer can hold is dependent on the equation

$$Max\ no\ of\ elements\ in\ buffer = \frac{Buffer\ size\ in\ bytes}{Data\ element\ size\ in\ bytes}$$

As this must be greater than 93 and the element size is 64 bits (hence 8 bytes) the buffer size was chosen to be 1024 bytes, this was because it must be a power of 2 of 512 bytes gave the number of elements as 64, lower than the required 93. With a buffer size of 1024 bytes we could therefore have up to 128 elements in the buffer. The instructions below show how the AMR is set.

```
MVK.S2          00000101B,B2;(0) ;Set register A5 to circular addressing, block size from BK0
MVKLH.S2            00001001B,B2;(0) ;set buffer block size for BK0 to 1024
MVC.S2          B2,AMR          ;(0) set AMR reg
```

In order to align the byte boundaries the declaration

```
#pragma DATA_ALIGN(x_buffer, BUFFER_BYTE_SIZE)
```

Was added to the code and then the C code was modified with the addition of a delay buffer with the code

```
#define BUFFER_SIZE 128
double x_buffer[BUFFER_SIZE];
```

The assembly function is then called in the ISR C code using the code shown below. Several variables are passed to function starting with a pointer to X_PTR which itself is a pointer to the front of the delay buffer; a pointer to the first coefficient (shown as &b[0]);a pointer to the new sample (&sample);a pointer to the output variable (&output) and also the number of coeffecients (defined as N).

```
/******************************** ISR_AIC*******************************/
void ISR_AIC(){
        sample = (double)mono_read_16Bit(); //place value at time of interrupt into sample
        circ_FIR_DP(&X_PTR,&b[0],&sample,&output, N);  //call assembly function, passing correct variables to it.
        mono_write_16Bit(output); //write output of above function to audio port
}
```

Within the assembly the operation of the prolog code was studied and then additions were made to the loop to create an assembly code which convoluted the input sample (of a sinewave) with our chosen filter coeffecients. This was first done in a basic set-up with no

parallelism and setting the delays to those given in the reference guide. The code written within the loop is shown below. The NOPs are added to prevent a read after write hazard as the LDDW takes 4 cycles and the MPYDP takes 9.

The functional units are selected both to prevent clashes in use and also for the correct register file, i.e. **`LDDW *B4++,B11:B10`** must use functional unit .D2.

The code operates by incrementing the pointer which points to the coefficients along by one, and then multiplying it with the samples in the delay buffer; with the delay buffer pointer incremented each time after the multiplication as the first A11:A10 value (our new ISR sample) is loaded in the prologue. The result of the multiplication is then added to our accumulator A15:14 which at the end of the loop is written back to the memory address of our output variable (from the C code).

```
loop:

        LDDW.D2         *B4++,B11:B10     ;load new B value
         NOP 4
        MPYDP.M1         B11:B10,A11:A10,A1:A0   ;multiply b value with value in circular buffer and store in result
         NOP 9
         ADDDP.L1        A15:A14,A1:A0,A15:A14   ;add result to accumulate result

        LDDW.D1         *++A5,A11:A10 ;load new value from circular buffer in memory

      SUB.L2          B0,1,B0              ; (0) b0 - 1 -> b0
       [B0] B.S2      loop     ; (5) loop back if b0 is not zero
       NOP             5
```

The length of the loop is determined earlier in the prologue by the instruction

```
MV.S2X          A8, B0      ;(0) move parameter (numCoefs) passed from C into b0
```

Where register A8 contains the value of N (number of coefficients) which is passed to the assembly from the function. Hence B0, a conditional register and so can be used to condition our branch unlike A8, is used as the counter within the loop, being decremented with every iteration of the loop.

The code was then benchmarked for the entire interrupt (including **`mono_read`** and **`_write`** which add some additional clock cycles) and takes 2399 cycles with almost no changes over different optimisation levels.

After which a frequency analyser was used to test that the written assembly was acting as expected, during which we noticed that code which multiplied one or two coefficients out incorrectly, would on an oscilloscope appear to give the current filter but on the analyser give a slightly different frequency response. When adjusted to the above code the frequency analyser then measured figure 23 which as we can see is very (near exact) similar to that

produced for our fastest implementation and what is predicted by the matlab simulation.

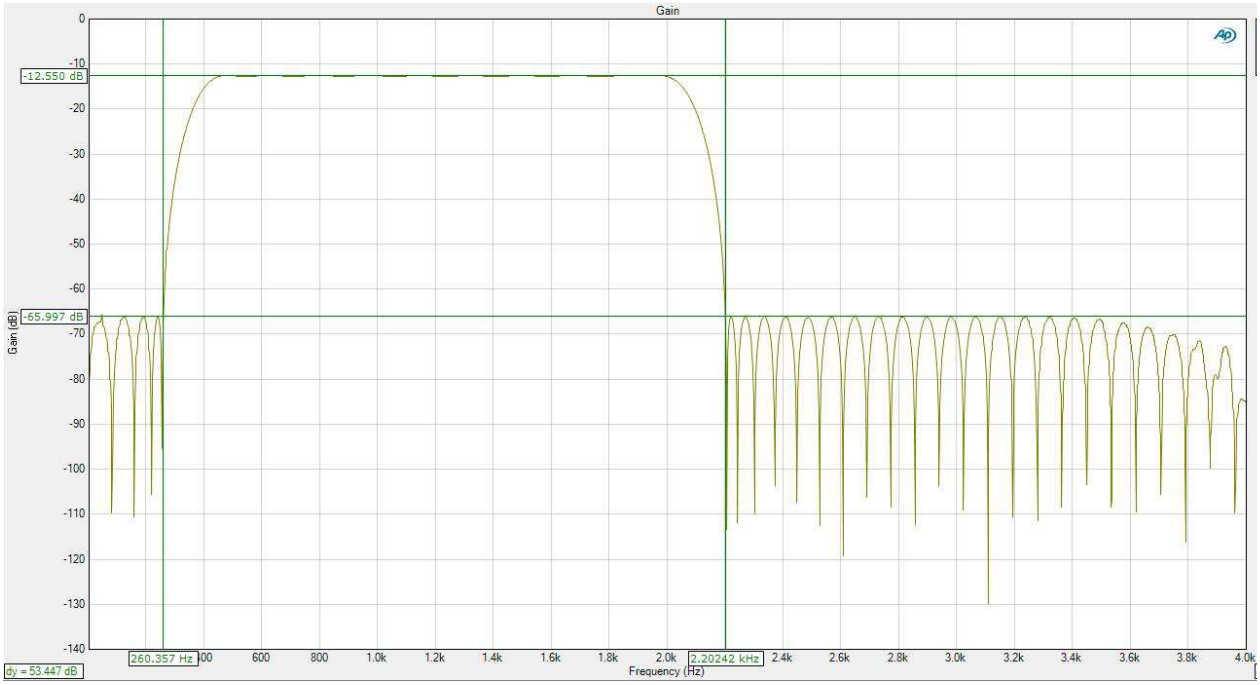


Figure 23. Graph of the frequency response of the assembly code.

Once again the result was almost exactly to specification, with stopband frequencies of 260 Hz and 2.2 kHz measured to 3 s.f. (as against 260 Hz and 2.2kHz specified).

Optimisation of assembly code

The initial implementation of the assembly code did not use all 8 processing units of the DSK, using only one at a time, thus wasting possible cycles by keeping some processing units idle. Additionally no pipelining optimisation was done, with instructions written and expected to be completed linearly by adding NOP operations to avoid any read after write hazards. The optimisations thus focused on performing operations in parallel while also taking advantage of other pipelining optimisations.

A snippet of the resulting code is as follows:

```
        ;*** all function setup completed above
        ;********************************* loop begin *********************************
        LDDW.D2         *B4++, B11:B10  ; load B value so that the first multiply addition can be completed.
        SUB .D2         B0,1,B0         ; (0) b0 - 1 -> b0
        NOP 3
loop:

        ; ************************ INSERT YOUR MAC CODE HERE ****************************
  [B0] B.S2             loop            ; (5) loop back if b0 is not zero
||      LDDW.D2         *B4++,B11:B10   ;load new B value
||      MPYDP.M1        B11:B10,A11:A10,A1:A0 ;multiply b value with value in circular buffer and store in result
||      ADDDP.L1        A15:A14,A1:A0,A15:A14   ;add result to accumulate result
||      SUB .L2         B0,1,B0         ; (0) b0 - 1 -> b0
||      LDDW.D1         *++A5,A11:A10           ;load new value from circular buffer in memory
        NOP 5
        ADDDP.L1 A15:A14,A1:A0,A15:A14  ;add data arriving in the pipeline after exiting the loop
        NOP 5
        ADDDP.L1 A15:A14,A1:A0,A15:A14  ;add data arriving in the pipeline after exiting the loop
        NOP 5
        ;*** function exit operations completed here
```

The first optimisation done is to place the branch before the rest of the instructions of the loop. This is due to the branch taking 5 cycles to execute, meaning that the four instructions below it will be executed before a branch is taken into account.

The second optimisation is to perform all load, add and multiply instructions in parallel. However this is more tricky than it seems (it does not involve just adding || in front of each instruction). Attention must be paid to what data is accessed when. This explains the addition of the `LDDW.D2 *B4++,B11:B10` before the loop starts, as it loads the b coefficient into memory such that the first multiply instruction can use it while another coefficient is loaded in parallel.

The first two `||  ADDDP.L1 A15:A14,A1:A0,A15:A14` (add) instructions do not actually write anything to the accumulate buffer as the multiply instruction takes 9 cycles to write back to memory, thus the first add instruction misses the write back from `|| MPYDP.M1 B11:B10,A11:A10,A1:A0` (multiply) by 5 cycles and the second add misses the multiply result by just 1 cycle, thus we are always 2 multiply results behind. This means that two extra add instructions must be added to "catch" the two results incoming in the pipeline as the loop is exited. Thus a NOP 5 must be added after exiting the loop to allow the first result to arrive from the pipeline and another NOP 5 must also be added after for the same reason.

The total number of cycles this code takes if 691 cycles.

Figure 24 allows us to explain the flow of information as well as the unit use by each instruction. As we can see during the main loop some more instructions could be added, such as performing another add and multiplication operation however thus would cause extra complications with memory dependencies. Should such an implementation be done the number of cycles could be halved to theoretically around 350 cycles.

The last optimisation possible would be to unroll the loop which would save cycles, avoiding to go back each loop iteration which wastes cycles.

	Prologue													loop		Epilogue
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
.L1															ADDDP (5)	ADDDP (5)	ADDDP (5)
.L2															SUB		
.S1										ZERO	ZERO						
.S2	MVC	MVK	MVKLH	MVC								MV			B		
.M1															MPYDP		
.M2																	
.D1					LDDW	LDW (4)	STW	STW	STW						LDDW		
.D2													LDDW	SUB (3)	LDDW		

Figure 24. This figure represents the pipeline of the optimized assembly code. Each unit is represented in the left column with the top row representing the cycle it is run in. The instructions are then sequentially represented with the black bubbles representing the NOP operations added to stall the pipeline to avoid any read and write hazards and ensuring the correct data is read.

Analysis of different implementations

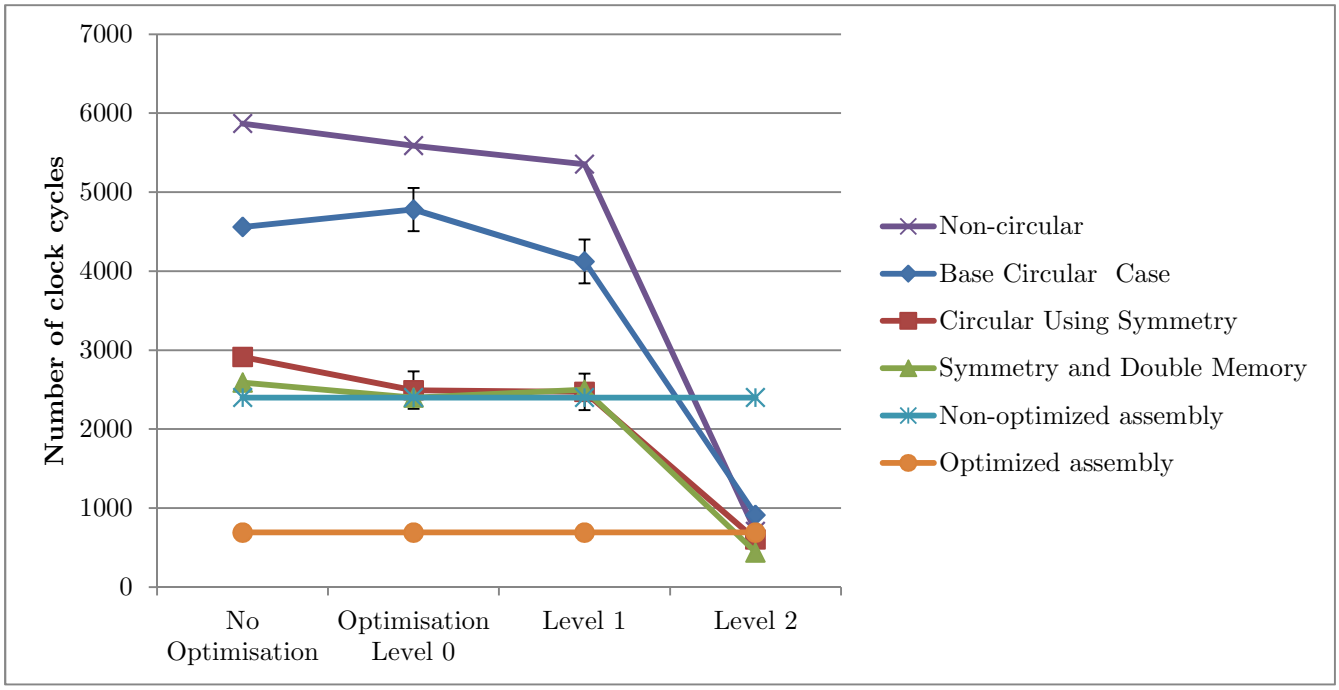


Figure 25. Graph of various implementation clock times. Assembly implementations are not affected by optimisation levels.

The best overall implementation so far is the double memory implementation taking advantage of the FIR symmetry properties.

An explanation of what each optimisation level does can be found earlier on the discussion about the performance of different circular buffer implementations.

The improvements due to optimisation level 0 are mainly attributed to the control flow optimisation as well as the loop rotation. This is because the control flow removes branches or redundant conditions which are predominantly present in the non-circular implementation. As the functions taking advantage of symmetry do not exhibit such behaviour little improvement at this level is expected – in fact it seems to make the base circular case increase its number of cycles (a result replicated multiple times to check if this was not an error). Loop rotations can also marginally help as the compiler evaluates the conditions to exit the loop, which, in many times, can reverse the execution and save one check/cycle. As all functions main operation is composed of a loop optimisation will happen there.

Improvements due to level 2 bring all 4 different optimizable implementations to under 1000 cycles per call. The main reason would be the use of software pipelining which ensure all 8 of the processing units are used. Thus operations with no data dependency are automatically placed in parallel. Loop optimisation uses a similar idea for instructions within a loop and parallelizes them as much as possible – this would have a large impact on performance if a loop has many iterations which is our case. Another point is that our loops rely heavily on arrays which are converted to pointers in optimisation level 2, which saves a few cycles per loop, thus a total of $N \times saved\ time$ cycles. Finally loop unrolling is another important optimisation which saves cycles by not having the processing unit update the program counter to return to the loop start each iteration. Instead a successful branch is only acted

upon to exit the unrolled loop. If the loop iterations are constant (which is our case) a complete unrolling can be done.

Thus the combination of software pipelining, loop optimisation and unrolling explains why our assembly code is not the most optimal and why the compiler manages to achieve better cycle times.

This fits in with Figure 26 found in lecture 2 which details how a C compiler can be more efficient than a human writing assembly code as the effort to unroll and fully utilize all 8 processing units is a tedious process which a compiler systematically going through the instructions can often achieve better.

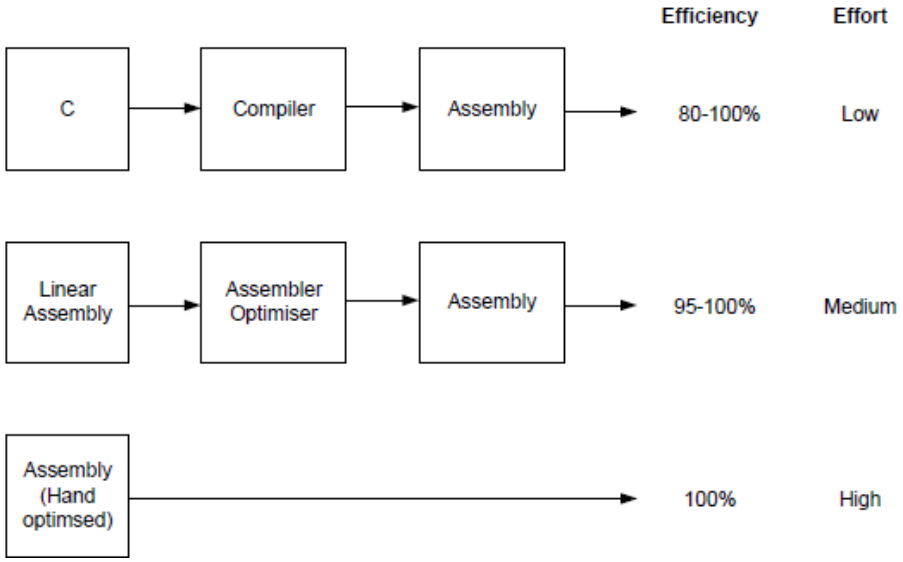


Figure 26. Graph detailing the effort to efficiency of writing software at various levels.

C Code

```
/*********************************************************************************
                        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>
#include <stdio.h>
// math library (trig functions)
#include <math.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"
//include coefficent variables
#include "coef.txt"

// 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>
#define BUFFER_BYTE_SIZE 1024
#define BUFFER_SIZE 128
double x_buffer[BUFFER_SIZE];
double * X_PTR = x_buffer;
#pragma DATA_ALIGN(x_buffer, BUFFER_BYTE_SIZE)
int var;
int rectify = 1;
int sampling_freq = 8000;
/*********************************** 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;
// Holds the value of the current sample
double output;
double sample;
#define N 93
double x[2*N];
int i = N-1;
int a=0;
int index = 0;
int index2 = N;
 /******************************* Function prototypes *********************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC(void);
double non_circ_FIR();
double base_circ_FIR();
double circ_FIR();
double doublesize_circ_FIR();
extern void circ_FIR_DP(double **ptr, double *coef, double *input_samp, double *filtered_samp ,unsigned int
numCoefs);
/********************************* Main routine ***********************************/
void main(){


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

  /* initialize hardware interrupts */
  init_HWI();

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

}


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

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

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

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

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


}

/********************************* init_HWI() ************************************/
void init_HWI(void)
{
        IRQ_globalDisable();                    // Globally disables interrupts
        IRQ_nmiEnable();                        // Enables the NMI interrupt (used by the debugger)
        IRQ_map(IRQ_EVT_RINT1,4);               // Maps an event to a physical interrupt
        IRQ_enable(IRQ_EVT_RINT1);              // Enables the event
        IRQ_globalEnable();                     // Globally enables interrupts

}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
/********************************* ISR_AIC ***************************************/
void ISR_AIC(){
        //uncomment to sue assembly code
        /*sample = (double)mono_read_16Bit();
        circ_FIR_DP(&X_PTR,&b[0],&sample,&output, N);
        mono_write_16Bit(output);*/
        mono_write_16Bit(circ_FIR());
        /*
        for (i = N-1;i>0;i--)
        {
                x[i]=x[i-1]; //move data along buffer from lower
        } // element to next higher
        x[0] = sample; // put new sample into buffer */
}
/*************************************************************************/
double non_circ_FIR(){
        //this fucntions performs convolution of an input sample with
        //coefficient values and past inputs in a non-circular buffer
        //implementation
        //reset output, a global variable, to zera
        output = 0;
        x[0] = mono_read_16Bit(); // place new sample into buffer
        //perform convolution by looping through all past samples and mutliple them
        //by their respective coefficients
        for (i=0;i<N;i++){
                //accumulate each multiply result to output
                output += b[i]*x[i];
        }
        //shift all elements such that input value can be placed in zero.
        for (i = N-1;i>0;i--)
        {
                x[i]=x[i-1]; //move data along buffer from lower
        } // element to next higher
        return output; //return output value

}

/*************************************************************************/
double circ_FIR(){
        x[index] = mono_read_16Bit(); //read input
        output = 0; //reset output
        //check where index is to determine which values will overflow
        if (index <= (N-1)/2){
                //handle the case where index -1 - i may overflow.
                //first loop which handles all values without overflow
                for (i=0;i<index;i++){
                        //performs factorized convolution
```

```
                        output += (x[index-1-i]+x[index + i])*b[i];
                }
                //second loop handling the overflow
                for (i=index;i<(N-1)/2;i++){
                        output += (x[index-1+N-i]+x[index + i])*b[i];
                }
                //handles the middle coefficient seperate to avoid doubling
                //a component of the output. This is only necessary for odd
                //N values.
                output += (x[index + (N-1)/2])*b[(N-1)/2];
        }else
        {
                //handle the case where index + i may overflow.
                //first loop
                for (i=0;i<N-index;i++){
                        //performs factorized convolution
                        output += (x[index-1-i]+x[index + i])*b[i];
                }
                //second loop
                for (i=N-index;i<(N-1)/2;i++){
                        //performs factorized convolution
                        output += (x[index-1-i]+x[index - N +i])*b[i];

                }
                //handle odd case
                output += (x[index-(N-1)/2 - 1])*b[(N-1)/2];
        }
        index--;//decrease index
        if (index<0){index=N-1;}//handle overflow of index
        return output; //return output
}
/*************************************************************************/
double base_circ_FIR(){
        x[index] = mono_read_16Bit(); //store in x[index] the input
        output = 0;                                             //reset output
        //loop from 0 to N-index which is the point just before
        //which overflow occurs. This avoids an
        //if function inside a for loop
        for(i = 0;i<N-index;i++){
                //simply accumulate the convolution result
                output+=x[index+i]*b[i];
        }
        //loop from N-index to N which when
        //overflow occurs. This handles the overflow.
        for(i = N-index;i<N;i++){
                output+=x[index+i-N]*b[i];//simply accumulate result
        }
        index++; //increase index
        if (index==N){index=0;} //check for overflow
        return output; //return value
}
/*************************************************************************/
double doublesize_circ_FIR(){
   //read input and place in memory
   //this is more efficient than reading input twice
   sample = (double)mono_read_16Bit();
   x[index] = sample; //store sample in its first index
   x[index2] = sample; //store sample in its second index
   output = 0; //reset output
   //loop from 0 to N-1/2
   for(i=0;i<(N-1)/2;i++){
        //use symmetry properties and perform convolution
        output += b[i]*(x[index+i]+x[index2-i-1]);
   }
   //use for the case where N is odd
   output += b[(N-1)/2]*x[index+(N-1)/2];
   index++; //increase index
```

```
    if (index==N){index=0;} //handle overflow
    index2 = index+N; //make index2 follow index
    return output; //return output value.
}
```

Assembly Code (optimized)

```
;*******************************************************************************
;                       DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
;                                  IMPERIAL COLLEGE LONDON
;
;                            EE 3.19: Real Time Digital Signal Processing
;                                    Course  by: Dr Paul Mitcheson
;
;                       LAB 4: Double precision FIR using Circular Buffer Hardware
;
;                                 ********** circ_FIR_DP.ASM **********
;
;*******************************************************************************
;                                Written by D. Harvey: 18 Jan 2010
;
;*******************************************************************************
;
        .global _circ_FIR_DP

        .text
;
;*************************** _circ_FIR_DP description ******************************
;
;           The input delay buffer has a data length of (size in bytes)/(data type length).
;                       The buffer you create must have a power of 2 size in bytes
;     i.e its length in bytes must equal 2^X bytes (where X is integer between 1 and 32).
;
;                Also ensure that its data length (size in bytes/8) is longer than the
;                coefficient array data length. The buffer will need to be data aligned
;                using  #pragma DATA_ALIGN(delay_buff_name, B) before it is defined
;                            where B is your chosen delay buffer size in bytes.
;
; circ_FIR_DP function call in C;
;
; circ_FIR_DP( &circ_ptr, &coef[0], &read_samp, &filtered_samp, N );
;
;*************************** Register Assignments ***********************************

; A0 LSB Multiplication  result                     B0 Loop Counter
; A1 MSB "                    "                     B1
; A2                                                B2 Used to set AMR to circular mode
; A3                                                B3 Return to C Address
; A4 &circ_ptr                                      B4 &coef[k]
; A5 circ_ptr                                       B5
; A6 &read_samp                                     B6 &filtered_samp
; A7                                                B7
; A8 Number of Coefs (N)                            B8
; A9                                                B9
; A10 LSB delay_circ[j]                             B10 LSB coef[k]
; A11 MSB   "                                       B11 MSB  "
; A12                                               B12
; A13                                               B13 Temp Store for previous AMR register value
; A14 MSB Accumulator                               B14
; A15 LSB   "                                       B15
;  See Real Time Digital Signal Processing by Nasser Kehtarnavaz (page 146) for more
;  info on mixing C and Assembly.
;*******************************************************************************

_circ_FIR_DP:
            ; set circular mode using the AMR

            MVC.S2                  AMR,B13           ;(0) Save contents of AMR reg to B13
            MVK.S2                  00000101B,B2;(0)
            MVKLH.S2                00001001B,B2;(0)
            MVC.S2                  B2,AMR            ;(0) set AMR reg

            ; get the data passed from C

            LDDW.D1                 *A6,A11:A10 ;(4) Get the 32 bit data for read_samp put it in A11:A10
            LDW.D1                  *A4,A5;(4) Get the address of the circ_ptr, dereference then place in A5
            NOP 4                                ; A5 now holds address pointing into delay_circ

            STW.D1                  A11,*--A5   ;(0) Store new input sample (MSB) to delay_circ array
            STW.D1                  A10,*--A5   ;(0) Store new input sample (LSB) to delay_circ array


            STW.D1                  A5,*A4 ;(0) write back the decremented pointer to circ_ptr
                                          ; this points to the end of the MSB of where the next sample
                                           ; will be stored on the next call to this function
```

```
            ZERO.S1                 A14             ;(0) zero accumulator LSB
            ZERO.S1                 A15             ;(0) zero accumulator MSB

        MV.S2X              A8, B0      ;(0) move parameter (numCoefs) passed from C into b0

            ;********************************* loop begin *********************************
            LDDW                *B4++, B11:B10
            SUB                 B0,1,B0             ; (0) b0 - 1 -> b0
            NOP 3
loop:


            ; ************************ INSERT YOUR MAC CODE HERE ***************************
   [B0] B.S2                    loop                ; (5) loop back if b0 is not zero
||          LDDW                *B4++,B11:B10       ;load new B value
||          MPYDP               B11:B10,A11:A10,A1:A0 ;multiply b value with value in circular buffer and store in result
||          ADDDP               A15:A14,A1:A0,A15:A14   ;add result to accumulate result
||      SUB                     B0,1,B0             ; (0) b0 - 1 -> b0
||          LDDW                *++A5,A11:A10               ;load new value from circular buffer in memory
            NOP 5
            ADDDP A15:A14,A1:A0,A15:A14
            NOP 5
            ADDDP A15:A14,A1:A0,A15:A14
            NOP 5



            ; MAC must use 64 bit IEEE double floating point data obtained from arrays defined in C




            ; ******************************************************************************

            ; manage loop


;   [B0] B.S2                   loop                ; (5) loop back if b0 is not zero
;        NOP                    5

            ;********************************* loop end **********************************

            ; send the result of MAC back to C

            STW.D2                  A14,*B6             ;(0) Write accumulator (LSB) into filtered_samp
            STW.D2                  A15,*+B6[1] ;(0) Write accumulator (MSB) into filtered_samp

            ; restore previous buffering mode

            MVC.S2                  B13,AMR             ;(0) restore  AMR reg to previous contents

            ; return to C code

lend:   B.S2                    B3                  ; (5) branch to b3 (register b3 holds the return address)
        NOP                     5

        .end
```