*Imperial College London*

Mattin Mir-Tahmasebi (mm5213) and Ahmed Ibrahim (aai13)

February 2016

RTDSP Lab 4 Report

*Real-Time Implementation of FIR Filters*

Contents

[Matlab Filter Design 2](#_Toc444268509)

[Non-Circular FIR Filter 4](#_Toc444268510)

[Circular-Buffer Filters 10](#_Toc444268511)

[Symmetrical FIR Filters 14](#_Toc444268512)

[Code Perfomance 19](#_Toc444268513)

[Spectrum Analyser Further Results 19](#_Toc444268514)

[APPENDIX A: filter\_genterator.m 24](#_Toc444268515)

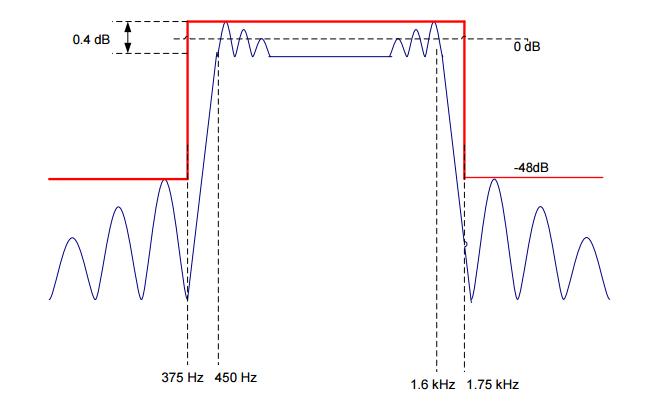
[APPENDIX B: initio.c 26](#_Toc444268516)

# Matlab Filter Design

Our FIR filter is designed using MATLAB. The filter has the following specifications:

* Stop band attenuation of -48dB
* Maximum passband ripple of 0.4dB
* Passband frequencies between 450 and 1750Hz

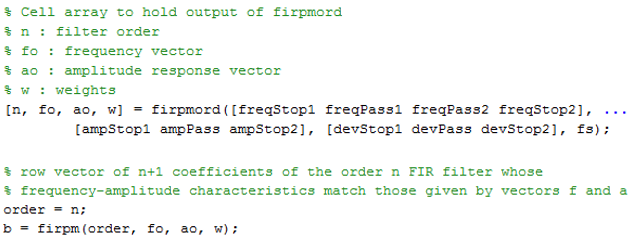
*Figure 1* below highlights the full specifications for the pass band filter.

.

*Figure 1 - The full specifications for the pass band filter [[1]](#footnote-1)*

To implement this filter, the Parks-MClellen algorithm is used. This algorithm uses iterative techniques (Remez algorithm) to design the filter to the specifications. Using a simple rectangular filter will produce large errors at the discontinuities due to the Gibbs phenomenon. The Parks-MClellen algorithm has the advantage that the errors between the ideal and actual magnitude response can be controlled and minimised in each of the frequency bands[[2]](#footnote-2) [[3]](#footnote-3).

The algorithm is implemented in MATLAB via the firpmord and firpm functions. firpmord takes parameters that describe the desired cutoff frequencies and amplitudes of the stopbands/passbands of the filter, the allowed deviation from the desired amplitudes, and the sampling frequency. It returns parameters that can be passed to firpm to produce an array of frequency coefficients.

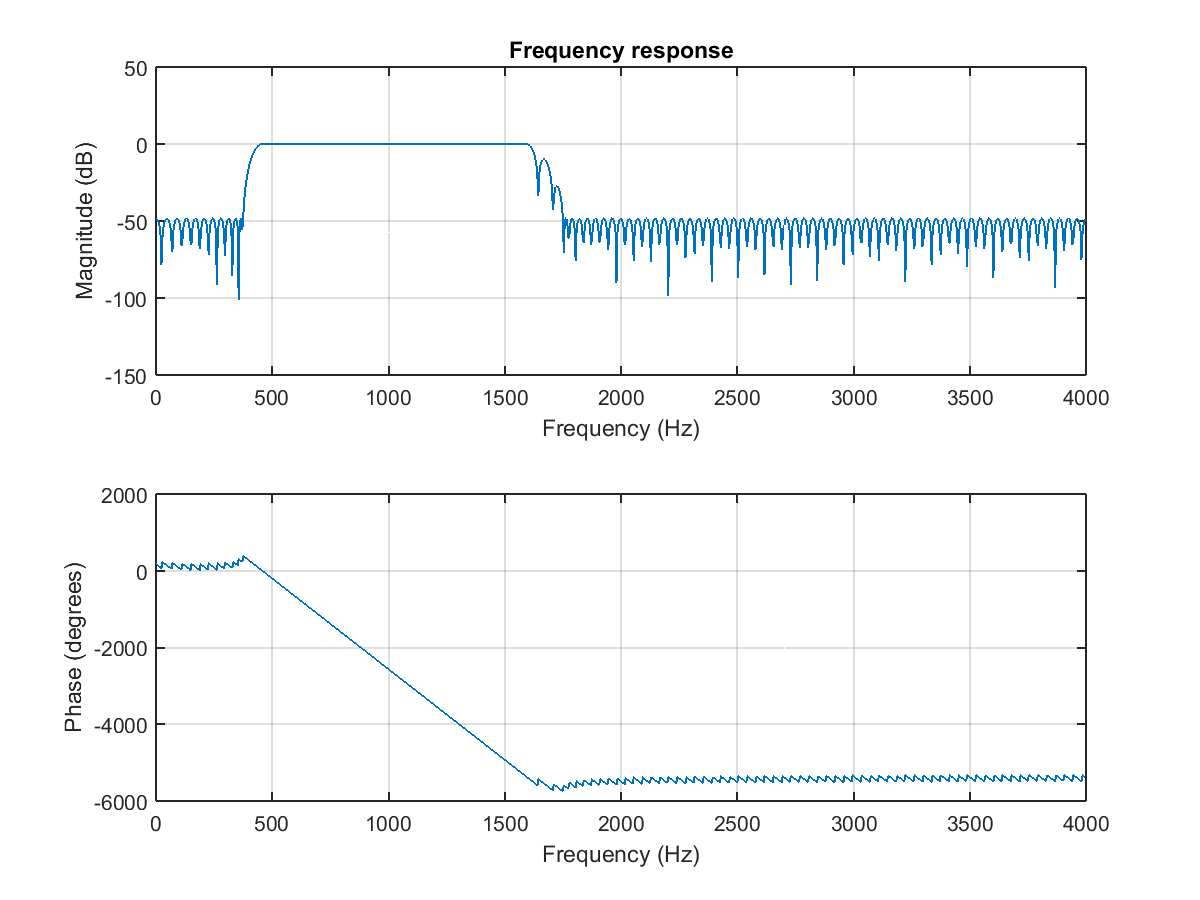


*Figure 2 - The firpmord function passes the passband specification as arguments to firpm, which returns an array of frequency coefficients, b.*

Our first attempt used the given specifications for the ideal magnitude response of the filter as the parameters to firpmord and produced the magnitude and phase response shown in *Figure 3*. The filter order, however, needed to be increased to n+4 due to this filter giving insufficient attenuation on the stop band (-47.54dB). The result of running the MATLAB script gives the specifications found on *Table 1*. So that we can easily refer to it later, the filter with these specifications will be called FIR1.

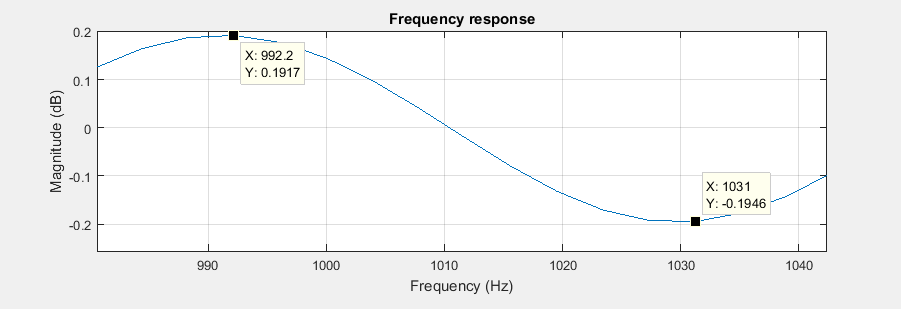
|  |  |
| --- | --- |
| FreqStop1 | 375Hz |
| FreqPass1 | 450Hz |
| freqPass2 | 1600Hz |
| freqStop2 | 1750Hz |
| devStop1 | 0.00251 |
| devPass | 0.02302 |
| devStop2 | 0.00251 |
| order | n+4 |
| No. of coefficients | 211 |

*Table 1 First FIR filter specifications (FIR1)*



*Figure 3 Magnitude and phase responses of the filter using the given specifications*

The new filter now gives good attenuation (-48.27dB) and passband ripple (found to be 0.386dB) as shown in *Figure 4*.



*Figure 4 Pass band ripple of FIR1 within the specified limits*

# Non-Circular FIR Filter

An N-tap FIR filter algorithm must be designed. The output is given by:

The Z-transform can be found:

Giving



*Figure 5 Direct form of an FIR filter [[4]](#footnote-4)*

Essentially, the Direct Form as shown in *Figure 5* must be implemented. To implement the delay line, an array buffer will be used, initialized to 0, as in *Figure 6.* The algorithm facilitates the convolution between the data array and the filter coefficient array, which amounts to performing a dot product between the two arrays, shown in *Figure 7* and *Figure 8*. Each new sample written to the data array is done so to the first position in the array, as demonstrated in *Figure 9*, to ensure it is aligned with the array of coefficients. To make space for the new sample, every item in the array is shifted up one position in memory before reading in data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| New sample x(n) | 0 | 0 | 0 | 0 |

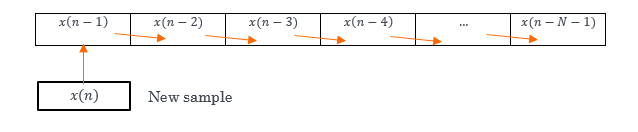
*Figure 6* *Initial empty buffer array*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

*Figure 7 Example array buffer with sampled components x(n)*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

*Figure 8 Example array containing filter coefficients*



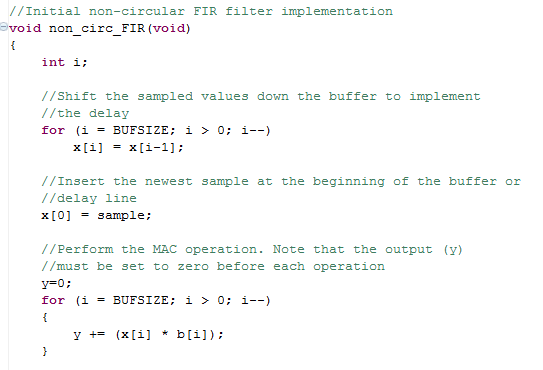
*Figure 9 Shifting data along the buffer before reading a new sample*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *\** | \* | \* | \* | \* | \* |
|  |  |  |  |  |  |

*Figure 10 The values are multiplied together*

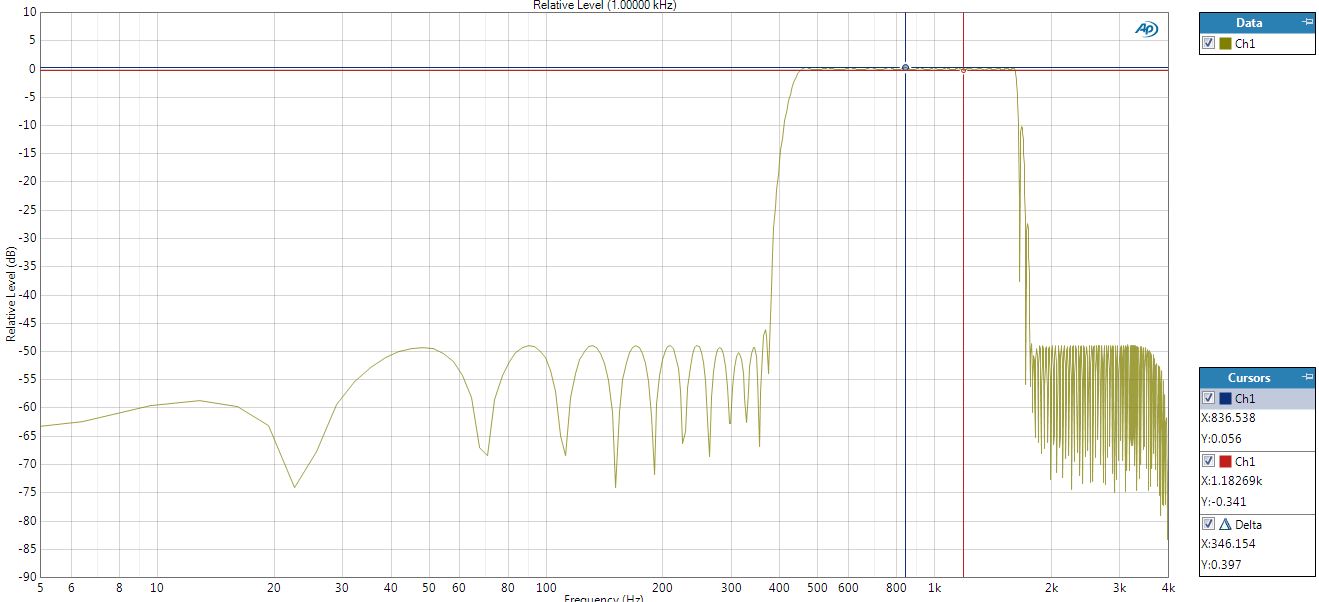
Lastly, each pair of values in the same position in the two arrays is multiplied, as shown in *Figure 10*, the results of which are accumulated and sent to the output. This is the mathematical representation of the process that is being performed:

*Figure 11* shows the implementation of this algorithm used in the C file.

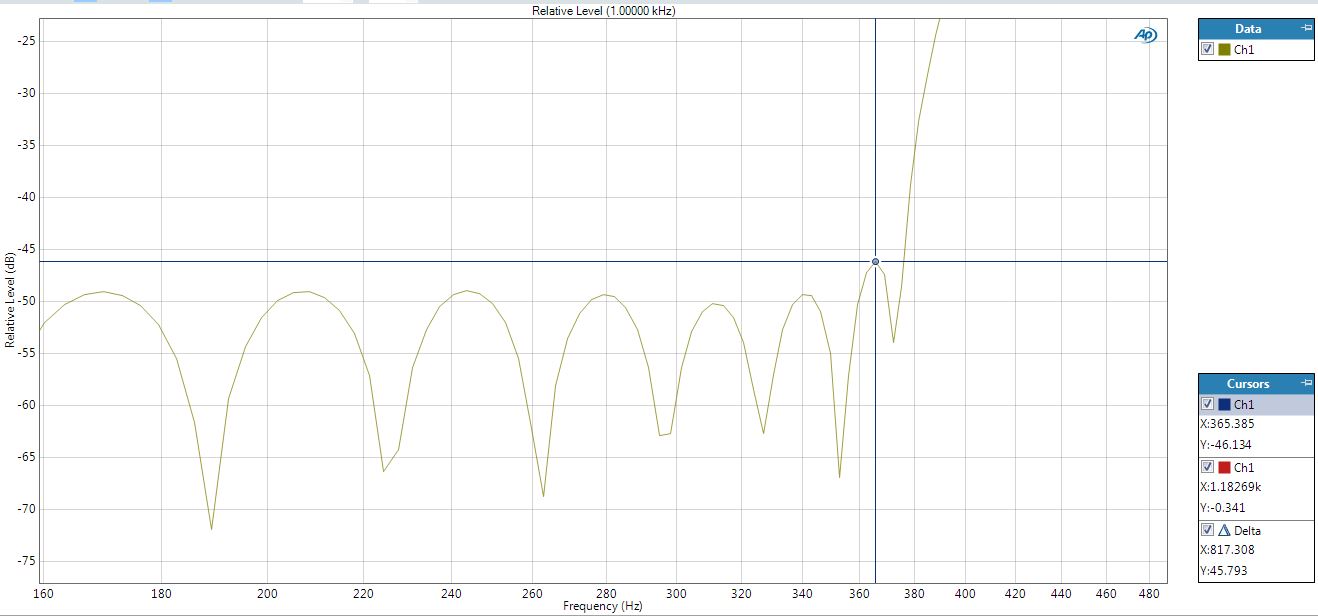


*Figure 11 The non-circular FIR filter implementation*

A frequency sweep can be performed using the APX520 Audio Analyser giving the actual frequency response of the filter. From the trace given in *Figure 12* it can be seen that the response closely matches the MATLAB trace for the magnitude response. However, it is found that the parameters differ slightly from the specification: around the stop band edge of the first transition band of the filter that was previously being produced, one of the lobes was not matching the specification according to the network analyser, even though it was found to be correct on MATLAB. This may be due to delay in the ADC/DAC of the DSP board. *Figure 13* highlights this stop band attenuation (-45.79dB).

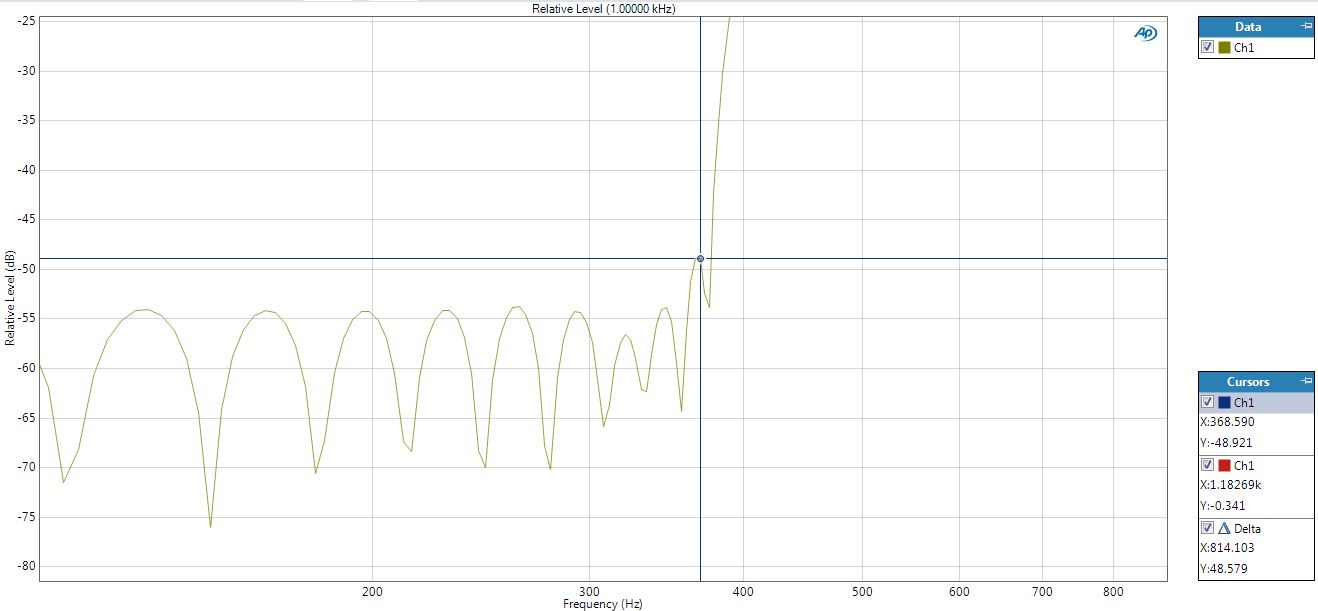


*Figure 12 Magnitude response of the initial filter*



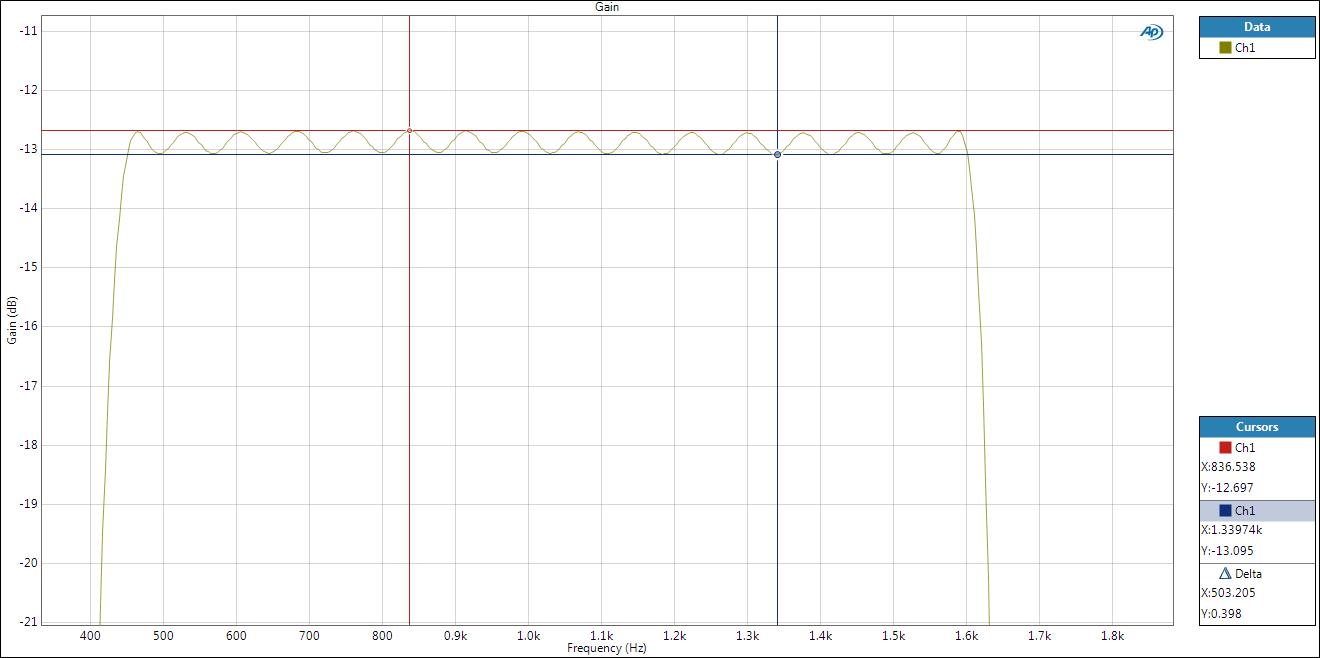
*Figure 13 Stop band attenuation of the initial filter*

To correct for this, a new filter must be designed through MATLAB. As the process for designing the filter using the Parks-MClellen algorithm is complex, trial and error is used to achieve a more correct result. The new filter’s frequency response can be found in *Figure 16*. The improved stop band attenuation of this new filter can be seen in *Figure 14*.

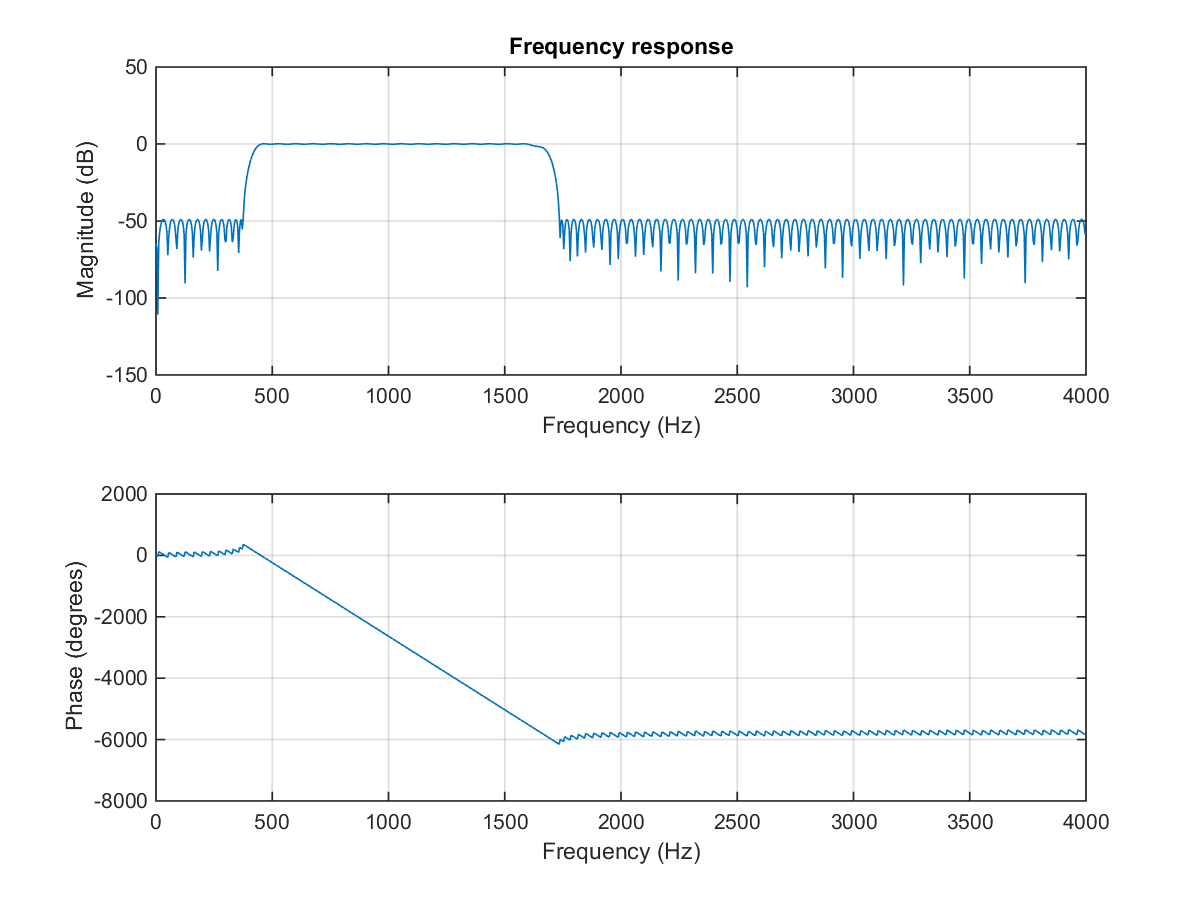


*Figure 14 Improved stop band attenuation*

The new filter, which we will refer to as FIR2, has the specifications summarized in Table 2. However, since the number of coefficients has now significantly increased, the performance of any FIR filter function will be hindered. Taking into consideration that this error is relatively small, to improve performance the filter FIR1 is used. The response for filter FIR2 is seen in *Figure 16,* and its specifications are listed in *Table 2.* The pass band ripple for FIR1 is seen in *Figure 15*, showing it is within the specified limits.



*Figure 15 Pass-band ripple for FIR1*



*Figure 16 Frequency response for the improved FIR2*

|  |  |
| --- | --- |
| freqStop1 | 375Hz |
| freqPass1 | 450Hz |
| freqPass2 | 1600Hz |
| freqStop2 | 1700Hz |
| devStop1 | 0.00251 |
| devPass | 0.02072 |
| devStop2 | 0.00251 |
| order | n+8 |
| No. of coefficients | 231 |

Table Specifications for improved filter (FIR2)

The performance of this FIR filter using the non-circular algorithm can be now found. The function is tested at different compiler optimization levels. An explanation of the different optimization levels is summarised in *Table 3*.

|  |  |
| --- | --- |
| Compiler optimisation level | Effect/Operation |
| 0 | This is the lowest optimization level. Code that is unused is eliminated and all variables are allocated to registers. Expressions and statements are also simplified. Using this level can sometimes even have a detrimental effect on the number of clock cycles used. |
| 1 | Includes all effects of optimisation level 0 and also eliminates local common expressions. Variables are checked to see if they can be turned into constants by performing local copy or constant propagation. |
| 2 | Includes all effects of optimisation level 1. At this level the compiler also attempts to generate software pipeline loops. Loops are unrolled and array references in these loops are converted to pointer form, drastically decreasing the number of cycles used in loop and array intensive functions |

*Table 3 Compiler optimization level effects [[5]](#footnote-5)*

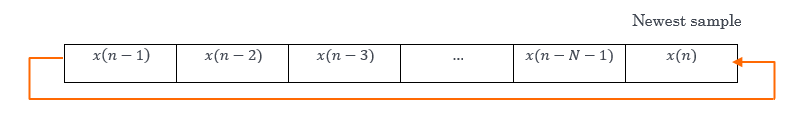
It can be expected that the non-circular implementation takes a significant number of cycles for each of the optimization levels. This implementation is slow since there are two for loops each iterating over the length of the full data buffer, performing several expensive memory read-write operations per iteration. To improve performance, a circular-buffer filter (see next section) can be used. *Table 4* highlights the number of cycles used for each optimisation level. It can be seen that the number of cycles significantly decreases when optimization level is increased.

|  |  |
| --- | --- |
| Optimisation | Number of cycles |
| None | 13409 |
| O0 | 11286 |
| O2 | 973 |

*Table 4 Number of cycles for each optimisation level for the non-circular algorithm*

# Circular-Buffer Filters

To improve performance, a circular buffer can be used. Unlike the previous case the samples are now stored in a circular buffer.

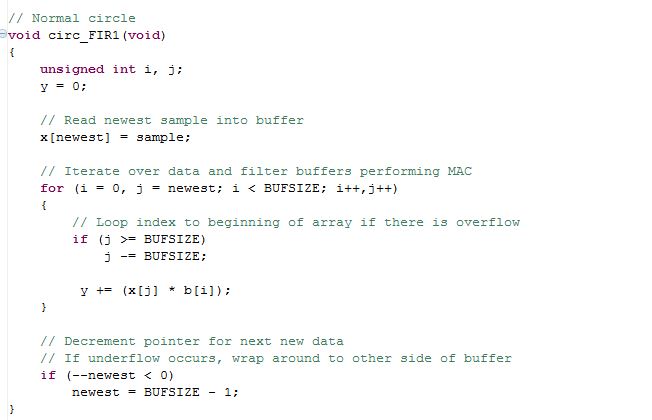


*Figure 17 Index is reset to end of array to prevent underflow*

The operation that is now being performed is

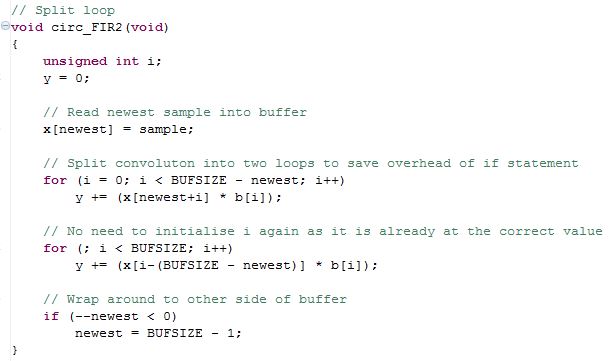
The function for the simple circular buffer can be seen in *Figure 18*. Unlike the non-circular algorithm, a circular buffer works by changing which position in the buffer each new piece of data is written to, rather than writing to the first position and shuffling all the data up one space. This means we need a variable which keeps track of which position the newest piece of data was written to – in our program, this is the newest variable. newest is initialised to the last index of the buffer, and is decremented to the previous position each time data is read in, meaning data is stored in descending order of array index. When newest reaches the start of the array, it is explicitly set to the last position again, to prevent underflow. This is demonstrated in *Figure 17*. In our program, BUFSIZE is a constant value that holds the size of the buffer.

The reason we need this variable is to keep the data buffer aligned with the filter coefficient buffer, to make sure we are multiplying the correct pairs of values. To achieve this, the convolution loop initialises the data buffer index, j, to the index of the newest piece of data, and the filter buffer index, i, to 0. The convolution continues similarly to the non-circular algorithm, with both indices incrementing, except that since j started mid-way through the array, at some point it will overflow, so an if statement is added to check for this overflow and correct it by returning j to the start of the array.



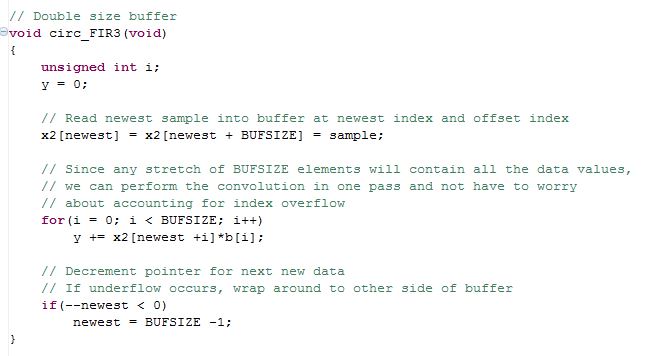
*Figure 18 circ\_FIR1 – The first circular FIR filter*

The if statement in the for loop of the previous algorithm only ever evaluates to true once for the duration of the for loop, however it is run every loop, causing unnecessary extra cycles to be used. We can remove it by pre-calculating the point at which the overflow will occur, and splitting the function into two for loops instead, producing the function in Figure *19*. This way, the same operations are executed, without the overhead of the if statement.

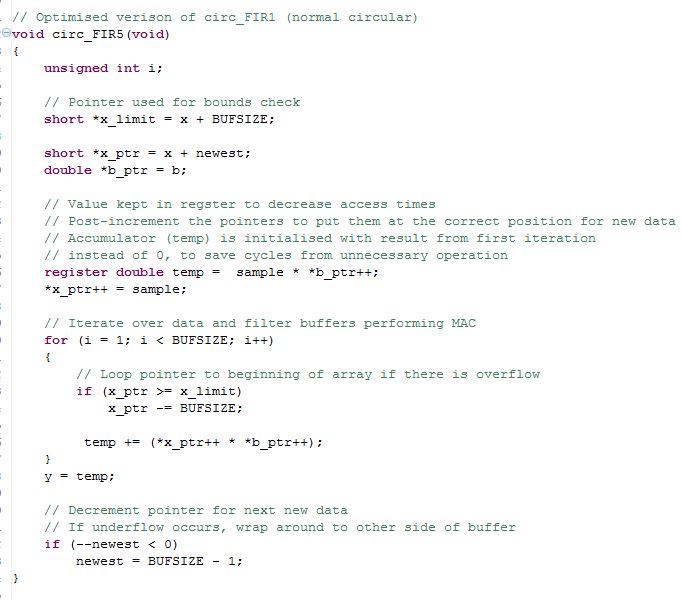


*Figure 19 circ\_FIR2 – Split loop*

Another method of removing the if statement is using a data buffer of twice the length, and storing each sample at the index pointed to by newest, but also at the index newest + BUFSIZE. This way, we can start our convolution iteration over BUFSIZE elements without having to worry about overflowing the array, and also being confident that each of the data values is accounted for, as any stretch of BUFSIZE elements starting at any point will contain all the data samples. If we start at newest, then we can be sure the array is aligned properly also. The function, shown in *Figure 20*, implements this.



*Figure 20 circ\_FIR3 – Double buffer size*



*Figure 21 circ\_FIR5 – An improved version of the normal circular algorithm*

circ\_FIR5 is an improved version of circ\_FIR1 – instead of array indexes, pointers are used. To understand why this is more efficient, we must realise that an array index is composed of two parts: a pointer to the start of the array (the name of the array) and an offset from the start of the array (the number in square brackets). When the compiler evaluates an array index, it has to compute the position by adding the offset to the pointer, which costs cycles. What we do, instead, is to post-increment the pointer every time it is used, ensuring that it is already at the correct position when it is needed, which cuts the time that was being used for the index evaluation. Additionally, we make use of the register keyword: memory accesses take a lot of time, and the register keyword forces the variable to be kept in a register so that it is always easily accessible by the processor. This is especially useful when applied to the variable storing the accumulated result of the convolution, as it is accessed every iteration of the loop. This function is shown in *Figure 21*.

### 

### Symmetrical FIR Filters

FIR filters usually have linear phase. This property gives a filter response that is linear when taken as a function of frequency. For an FIR filter to be linear phase, it must be symmetrical. A symmetrical filter means that the first and last coefficients are equal, the second and second-last coefficients are equal, and so on. i.e.

The proof of the linear phase property can be found through finding the Fourier transform[[6]](#footnote-6) of:

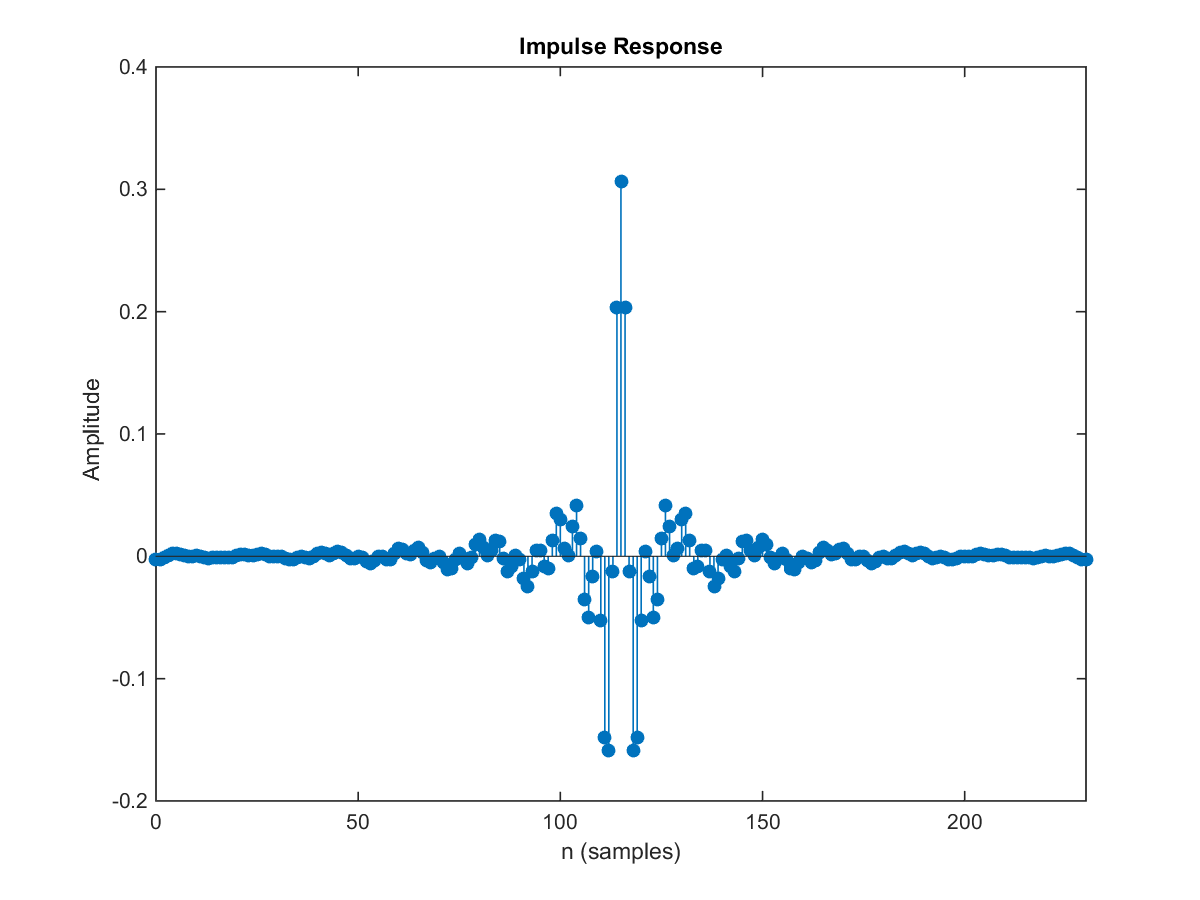
Taking

Assuming N is even

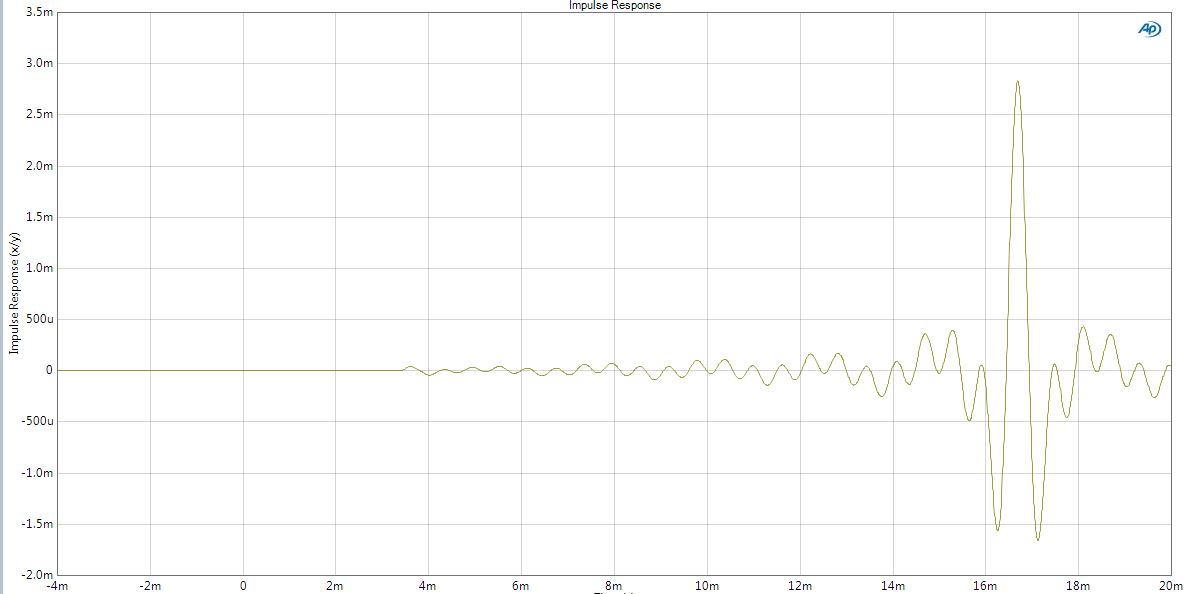
Substituting

For

Since is real, the overall phase is , giving linear phase. Taking the impulse response of the filter in MATLAB (*Figure 22*), it can be seen that it is symmetrical. The impulse response can also be seen using the spectrum analyser (*Figure 23*).



*Figure 22 Impulse response of filter in MATLAB, showing symmetry*

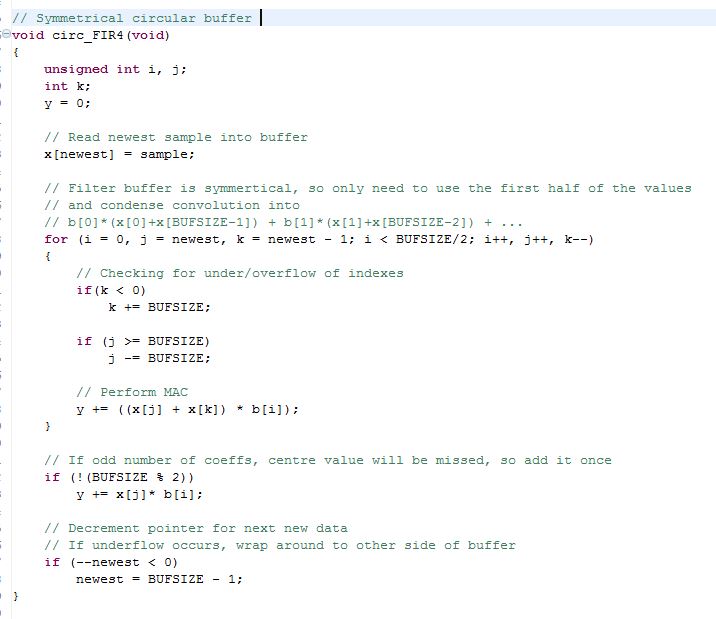


*Figure 23 Impulse response as seen from the spectrum analyser, showing symmetry*

This shows that the coefficients at and therefore this property can be used to produce a more efficient algorithm. Since the first/last coefficients are equal, as well as the second/second last and so on, we only need to perform the half as many multiplications, as we can factorise x[0]\*b[0] + x[1]\*b[1] + ... x[BUFSIZE-2]\*b[BUFSIZE-2] + x[BUFSIZE-1]\*b[BUFZSIZE-1] into b[0]\*(x[0]+x[BUFSIZE-1]) + b[1]\*(x[1]+x[BUFSIZE-2]) + ...

This is achieved by initialising an index, j, to start at the newest data value, and another index, k, to start at the oldest data value. Every iteration, j will increment and k will decrement, and the convolution is carried out as described by the factorisation above. Again there is the issue of under/overflow, which is mitigated by two if statements that work similarly to the one in circ\_FIR1.

If the number of coefficients is odd, then j and k will not reach the middle element, since the condition of the for loop specifies to break before it is reached. Due to this, we manually add the last value to the result. It would be equally correct to allow the middle element to be added twice and then subtract it once afterwards, but this would require an extra unnecessary calculation, so the former method was chosen. The implementation of this algorithm is shown below in *Figure 24*.

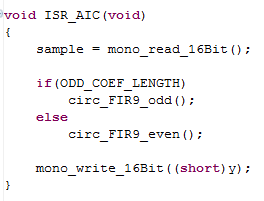


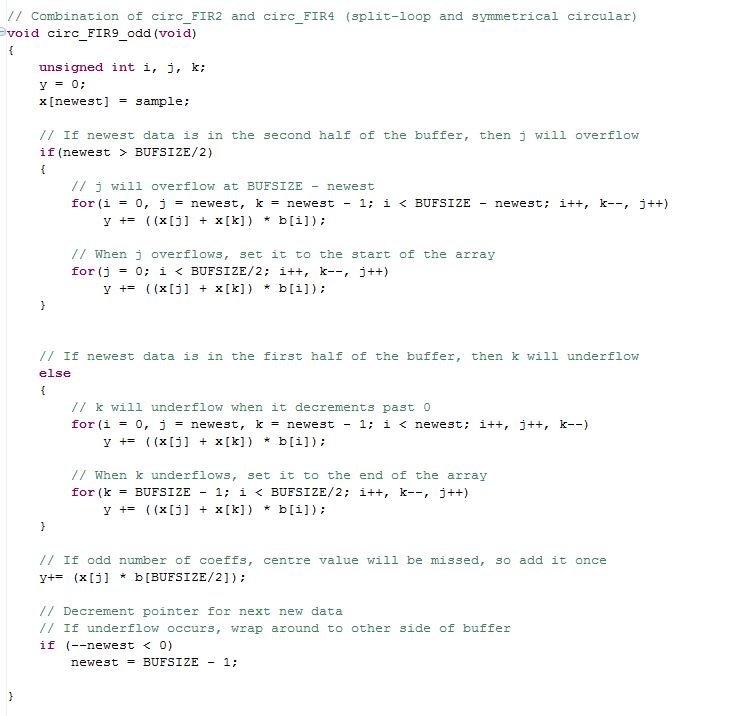
*Figure 24 circ\_FIR4 – Symmetrical circular buffer*

The algorithm shown in *Figure 26* combines the split loop and symmetrical circular buffer algorithms described earlier to benefit from the speed increases they both bring. Unlike the original split loop algorithm, there are now two pointers to keep from under/overflowing, so the first operation the function undergoes is checking to see what value newest holds, since j and k are set based on its position. If newest is in the second half of the buffer, then j will overflow, as it is initialised to newest and is incremented BUFSIZE/2 times. Conversely, if newest is in the first half of the buffer, k will underflow, as it is decremented the same number of times. Depending on which of those two possibilities occur, when j or k reach the position at which they would over/underflow in the first loop, they are set to the start/end of the array, respectively, in the second loop.

circ\_FIR4 had an if statement checking whether or not the length of the coefficient array was odd or even, in order to know if an extra value should be added to the result. This added overhead, and even included a modulo operator which is especially slow. Instead, we have split circ\_FIR9 into an odd version and an even version in order to remove that check. To implement this, we determine whether or not the coefficient buffer has an odd or even length in the pre-processor, then use that value to determine which of the two functions will be executed (*Figure 25*). If the odd function is chosen, then the extra value is added explicitly, and if it is the even, then that statement is removed, ensuring that extra cycles are not wasted if the filter has an even number of coefficients.

*Figure 25 Determining parity of the coefficient array (left) and choosing a function based on the parity (right)*





*Figure 26 circ\_FIR9\_odd algorithm*

# Code Perfomance

The functions were all tested at different optimisation levels, giving some interesting results summarised in *Table 5*.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Optimisation\Algorithm | Non\_circ | Circ\_FIR1 | Circ\_FIR2 | Circ\_FIR3 | Circ\_FIR4 | Circ\_FIR5 | Circ\_FIR9\_odd |
| None | 13952 | 13141 | 10840 | 10429 | 6898 | 13409 | 5701 |
| O0 | 11286 | 10184 | 9777 | 9582 | 5308 | 4560 | 4560 |
| O2 | 973 | 1342 | 1594 | 2418 | 3311 | 810 | 860 |

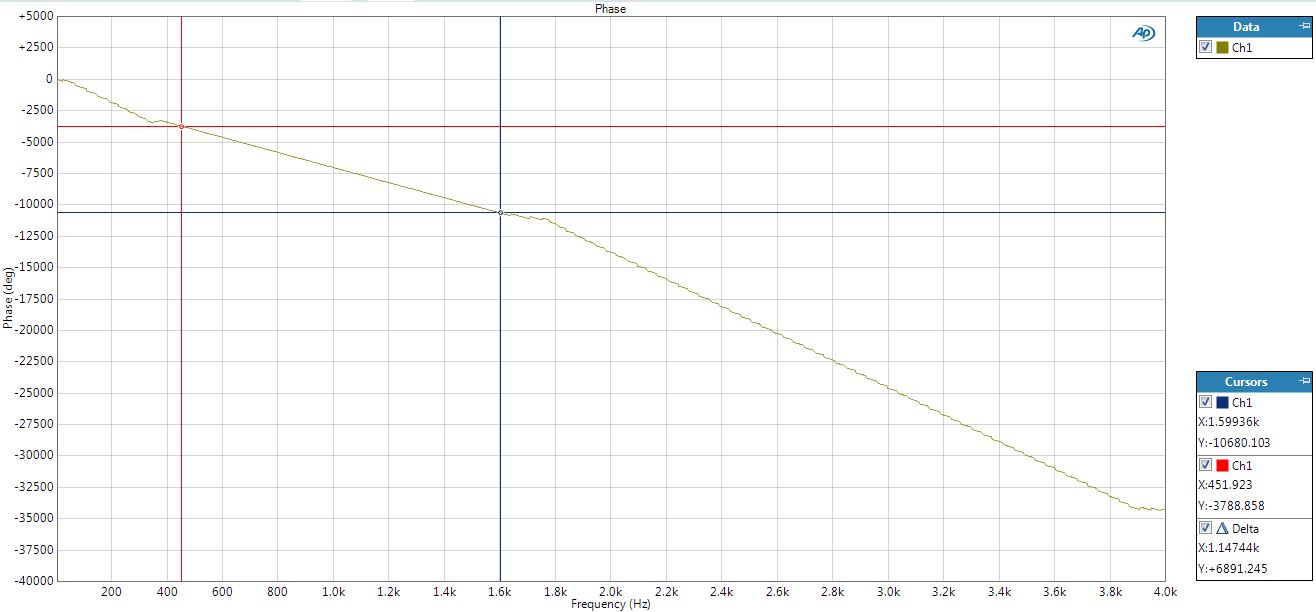
*Table 5 Number of cycles used for different algorithms with each optimization level*

Without compiler optimisations, the fastest algorithm is circ\_FIR9\_odd, as seen in *Table 5* above. This is logical, as it is the algorithm with the most advanced speed-increasing techniques, combining the benefits of the split-loop and symmetrical circular algorithms. However, when -o2 is applied, it is no longer the quickest, and is second to circ\_FIR5, the improved version of the basic circular buffer algorithm. This is counter-intuitive, but is determined by the subtleties of the specific compiler we are using, and how easily it is able to infer that we are trying to implement a multiply-accumulate algorithm. If it successfully does so, it will be able to arrange the assembly code into an efficient structure, giving us the best results. It is difficult to know precisely why circ\_FIR9\_odd is better optimized by the compiler, but a possible reason is the extra control flow statements may make the function’s purpose less obvious. It is evident, however, that with no optimisations, the non-circular algorithm is the slowest. This is due to the excessive memory accesses.

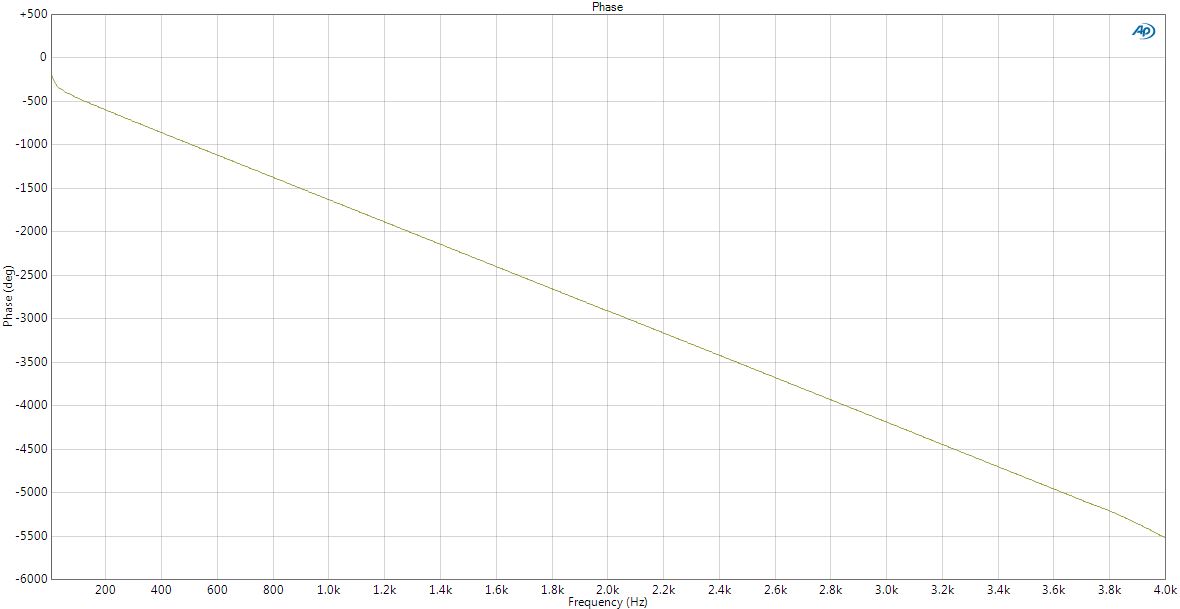
# Spectrum Analyser Further Results

The initial spectrum analyser results were already completed in the non-circular buffer section. Since the frequency response is the same with both filters, the magnitude and phase plots are identical. In this section the responses for the fastest FIR implementation are used, and the origin of some characteristics of the responses are explained. The linear phase property of FIR filters is confirmed in *Figure 27,* where the filter gives a linear phase in the pass-band.

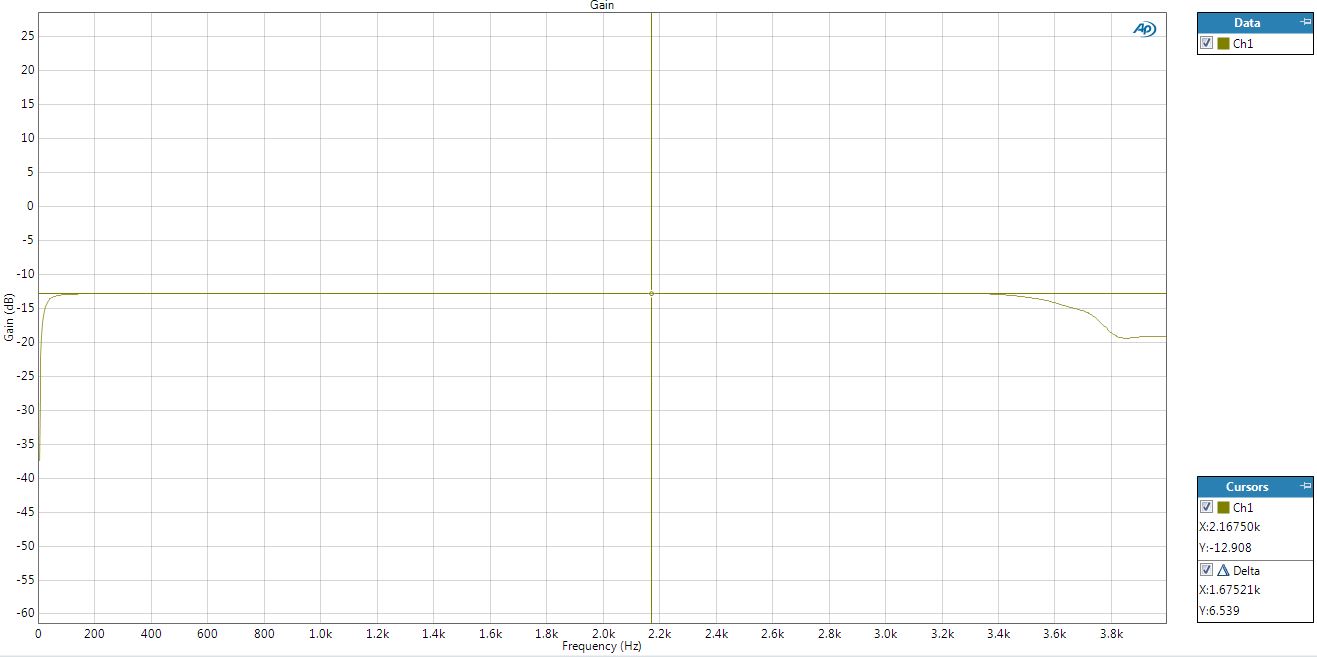
However, the phase response of the faster filter given in Figure 27, shows a more significant phase shift than is expected (using the MATLAB results). The response seen in *Figure 16* shows the expected phase shift from MATLAB. The reason is due to the output filter of the AIC23 Audio Chip (*Figure 32*) gives this additional phase shift. Running the program without using a filter and finding the phase response using the spectrum analyser gives the trace found in Figure 28.



*Figure 27 Phase response of the FIR filter*



*Figure 28 Phase response without using an FIR filter*

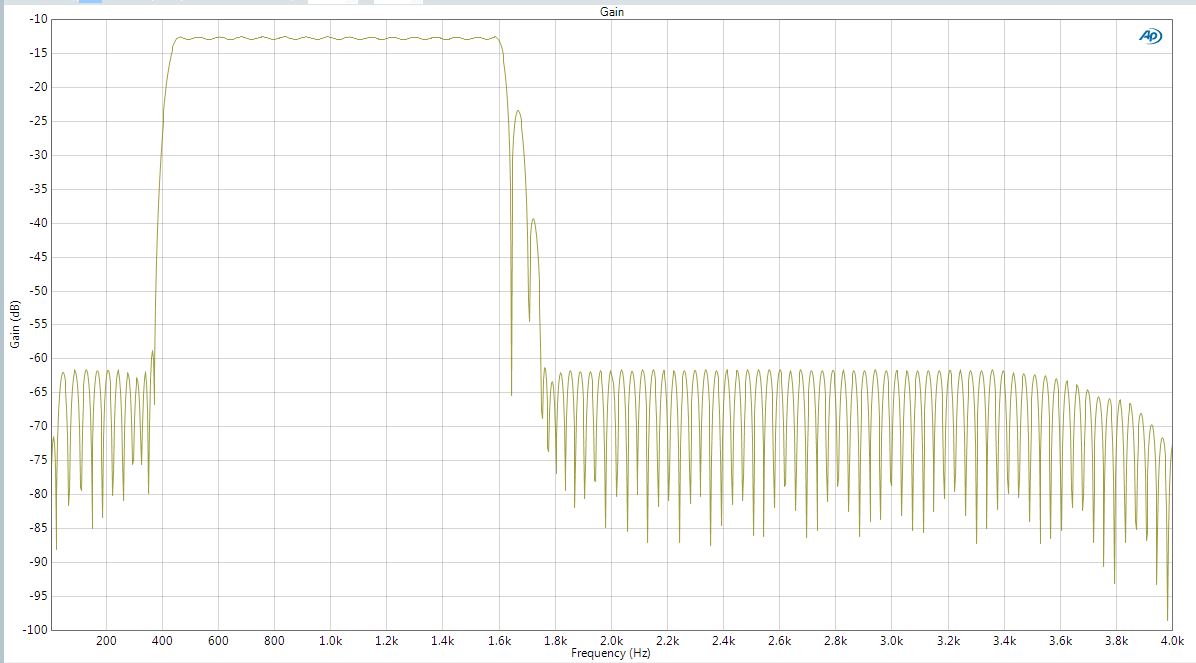


*Figure 29 No FIR filter magnitude response, note the less than -12dB attenuation*

Another effect that can be seen is the roughly -12dB attenuation shown through the actual magnitude response given by the spectrum analyser. This is highlighted in *Figure 29* where the magnitude response without using a filter shows a roughly 1-2dB attenuation. This attenuation occurs due to two reasons:

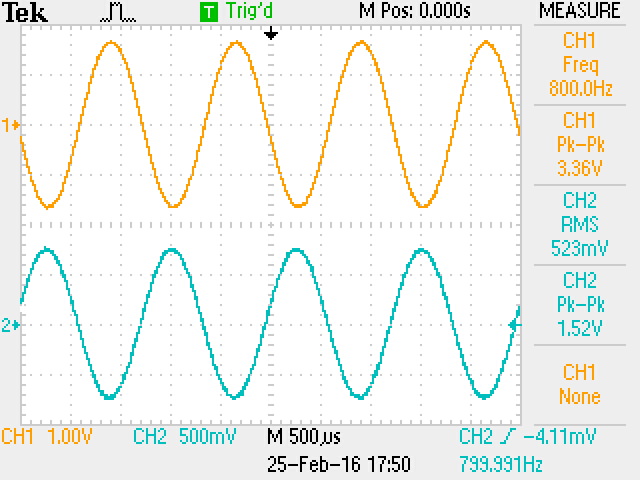
* The input potential divider of the AIC23 Audio Chip (*Figure 32*).
* The left and right line-in’s are averaged (only one line-in is used in this case)

The output is therefore of the input, giving the 12dB attenuation. For simplicity, the spectrum analyser traces are shown with the gain at a relative level to help with the validation of the response (checking to see if the response meets the specification). An unadjusted magnitude response trace showing this attenuation is given in *Figure 30*.

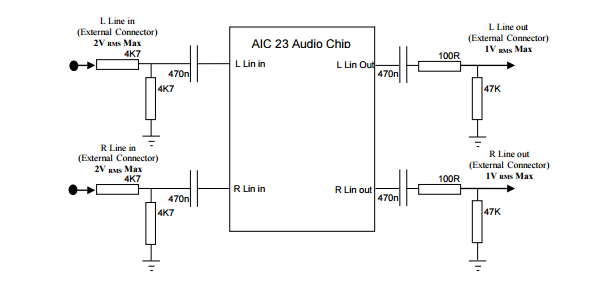


*Figure 30 No relative gain, full response showing the attenuation*

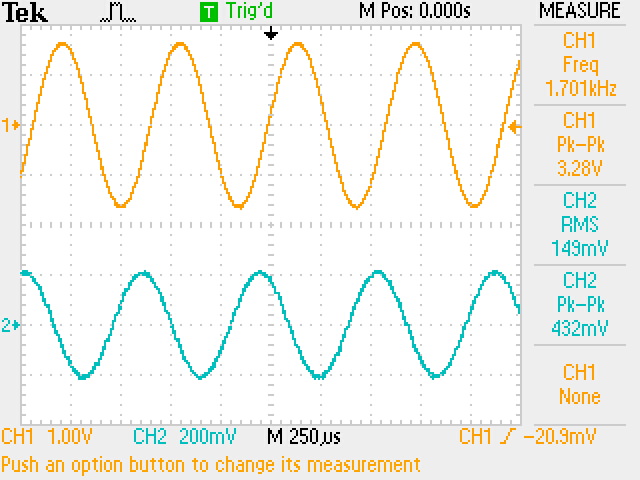
Using the filter and the non-circular implementation of the function, the output of the board can be seen in the scope trace. The input frequency used is 800Hz, which is well within the passband. As can be seen from *Figure 31* the output is given is as expected. There is a significant phase shift as expected. However, as can be seen, the output is roughly a half of the input.



*Figure 31 Scope trace for an input frequency of 800Hz (Input = Yellow)*

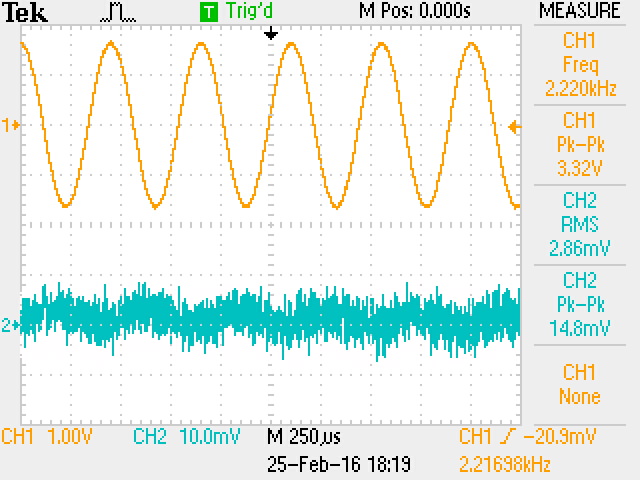


*Figure 32 AIC23 Audio chip [[7]](#footnote-7)*



*Figure 33 Scope trace in the transition band at 1700Hz giving a clear reduced amplitude (Output in Blue)*

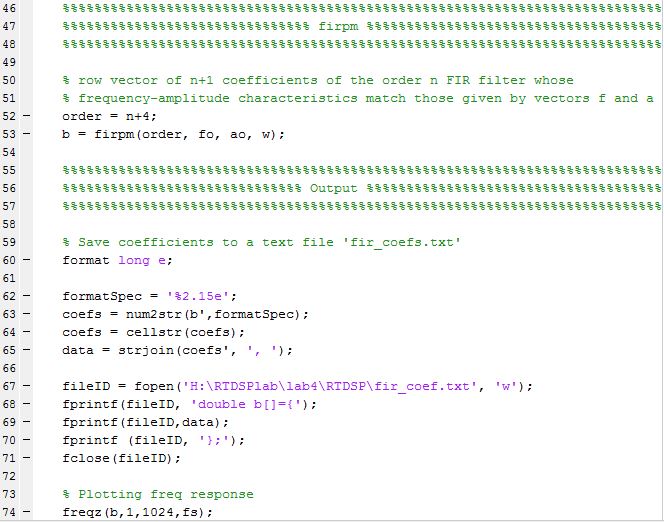
*Figure 33* and *Figure 34* show the outputs clearly in the transition and stop bands. With *Figure 33* the amplitude is clearly reduced as expected within the transition. With *Figure 34* the input frequency is 2200Hz – deep within the stop band. There is very large attenuation giving a very small amplitude noisy output.



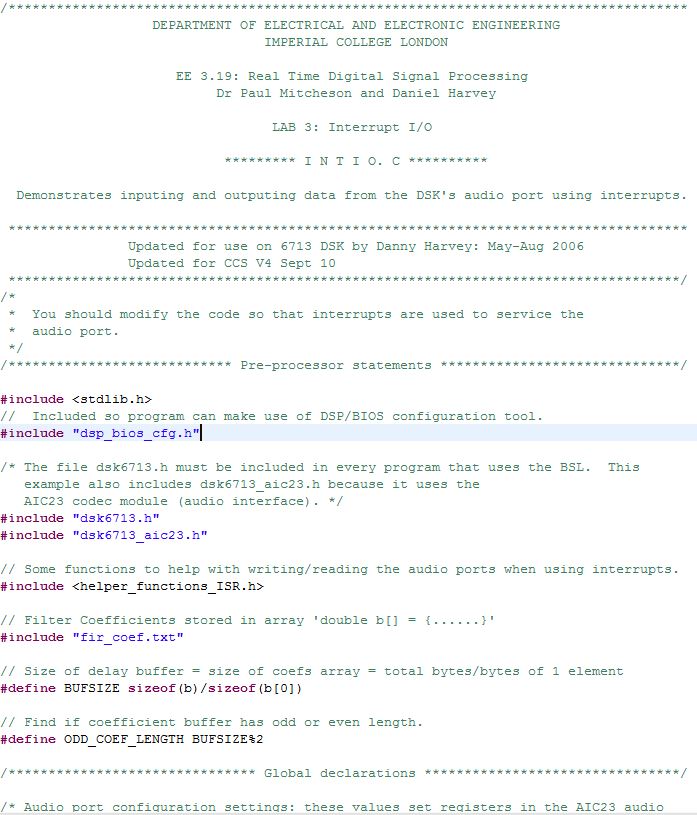
*Figure 34 Scope trace with an input frequency deep within the stop band giving a noisy very low amplitude output (Output in blue)*

# APPENDIX A: filter\_genterator.m

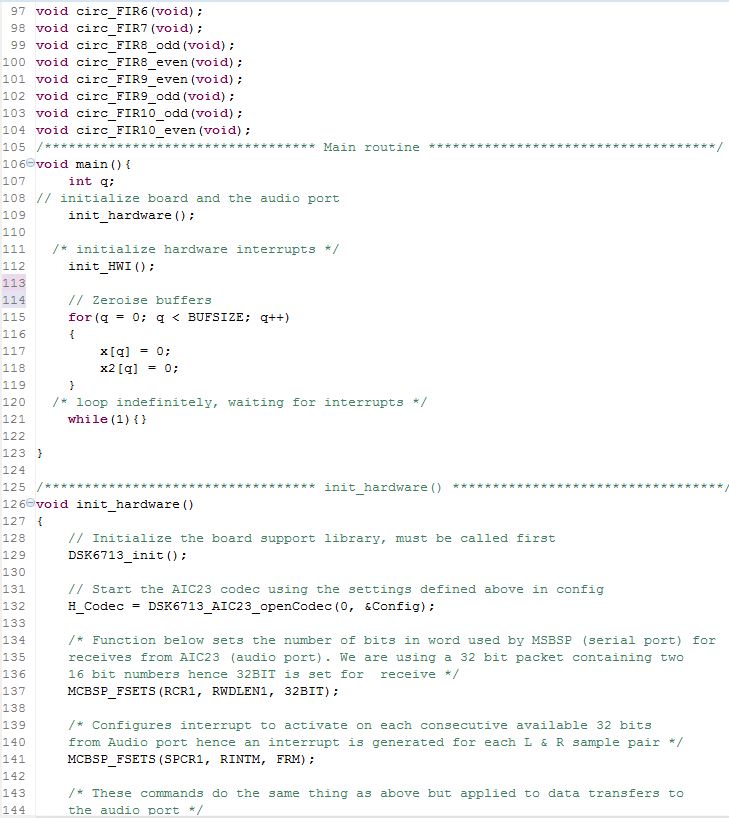


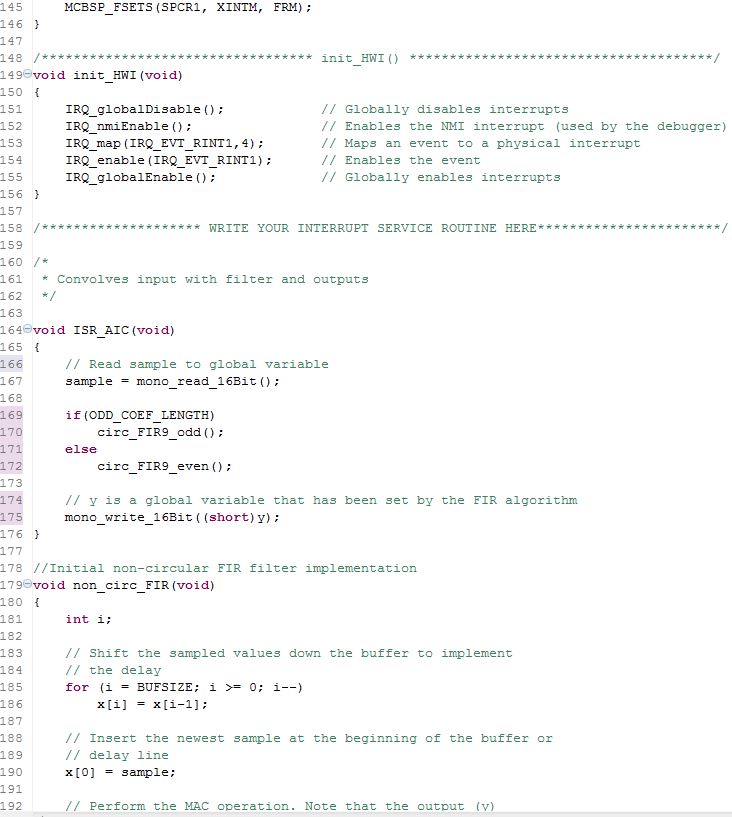


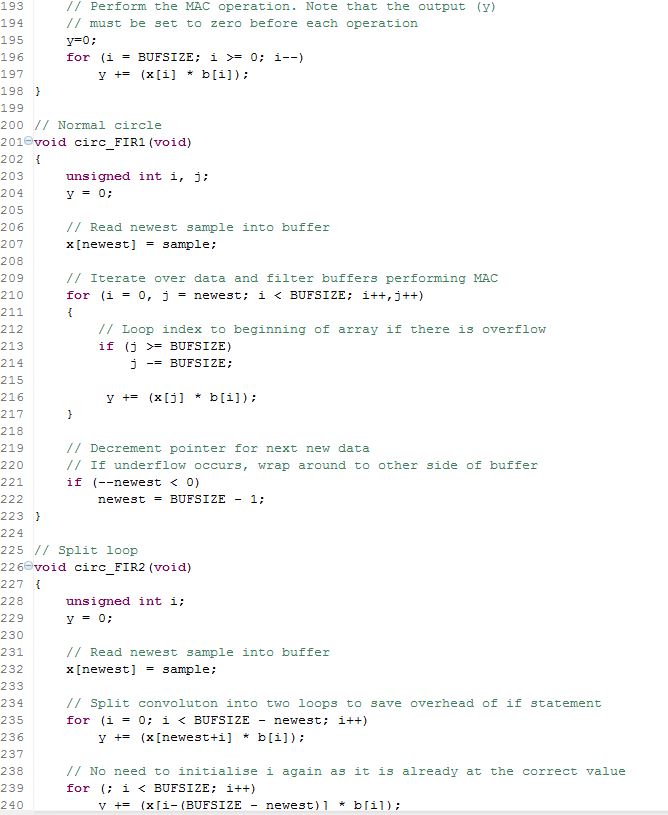
# APPENDIX B: initio.c

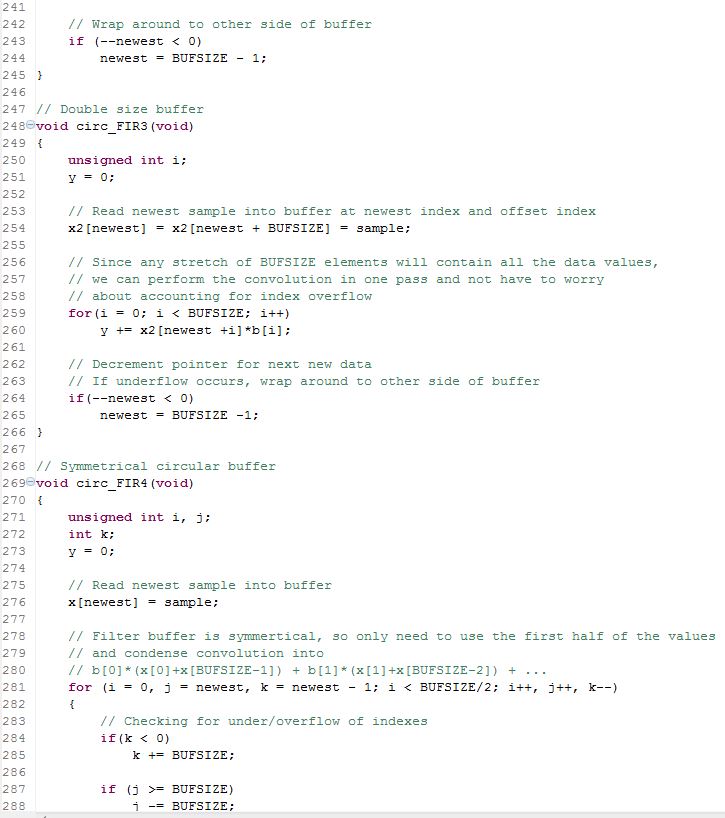


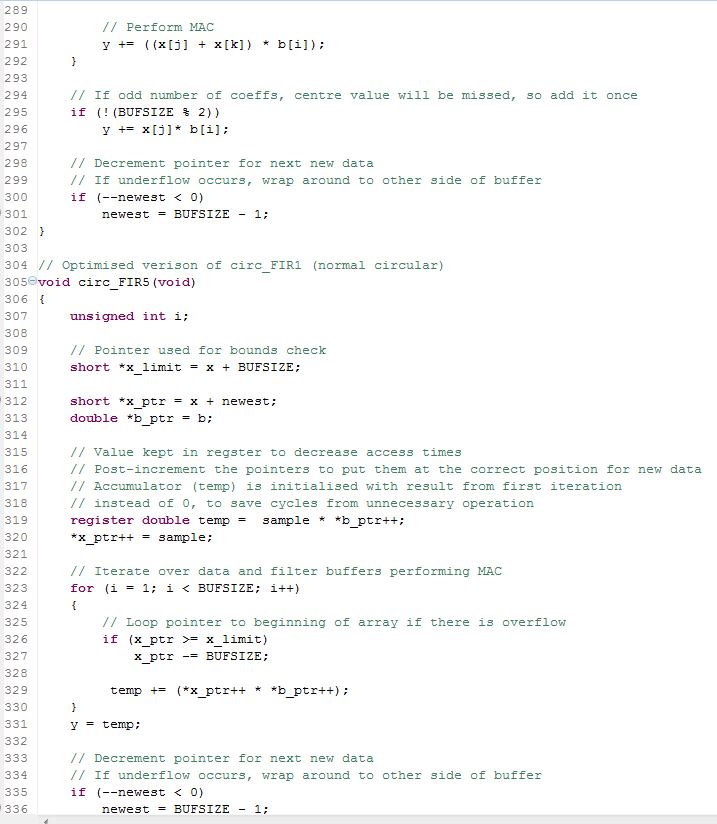


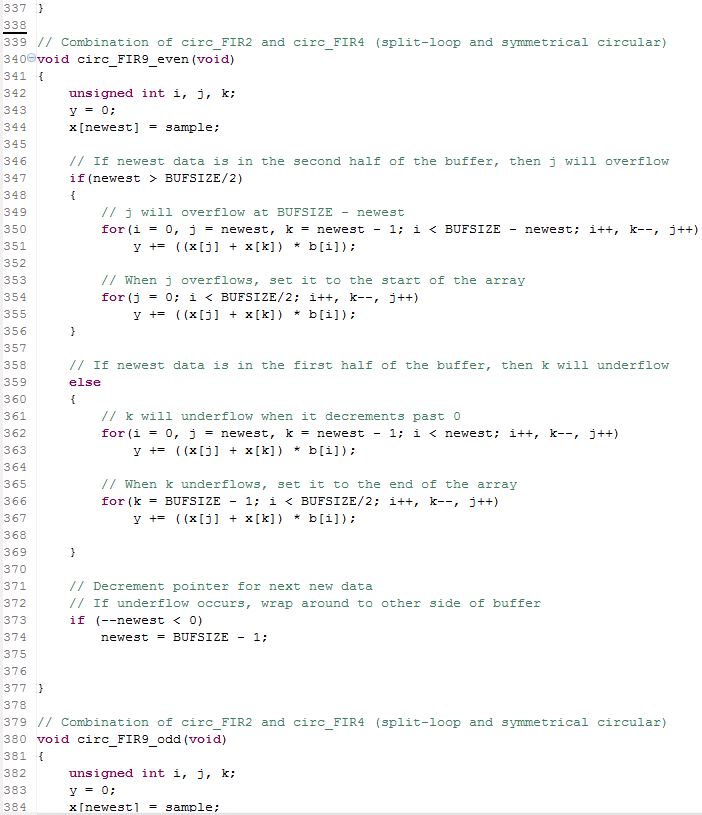


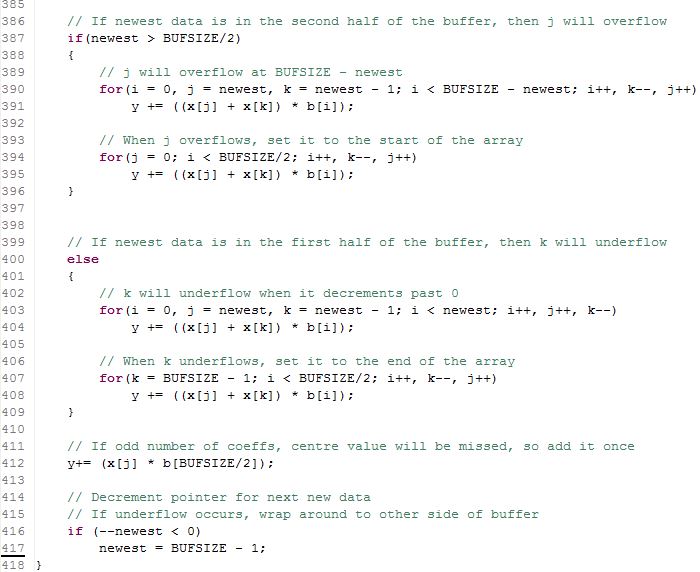












Newest sample

1. RTDSP Lab4 Coursework Guidelines [↑](#footnote-ref-1)
2. http://www.ni.com/example/30772/en/ [↑](#footnote-ref-2)
3. http://signal.ece.utexas.edu/~arslan/courses/dsp/lecture18.ppt [↑](#footnote-ref-3)
4. https://en.wikipedia.org/wiki/Finite\_impulse\_response#/media/File:FIR\_Filter.svg [↑](#footnote-ref-4)
5. TMS320C6000 Optimizing Compiler v7.4 User's Guide [↑](#footnote-ref-5)
6. http://melodi.ee.washington.edu/courses/ee518/notes/lec15.pdf [↑](#footnote-ref-6)
7. *AIC23 Audio chip external components adapted from TMS320C6713 Technical ref (page A-14, 2003 revision A* [↑](#footnote-ref-7)