REAL TIME DIGITAL SIGNAL PROCESING

Lab 4 – Real-time Implementation of FIR Filters

Zoe Williamson CID: 00942159
Vikrant Karna Sethia CID: 00939390

Contents

Declaration: We confirm that this submission is my own work. In it I give references and citations whenever I refer to or use the published, or unpublished, work of others. I am aware that this course is bound by penalties as set out in the College examination offences policy.

Signed: Zoe Williamson, Vikrant Karna Sethia

Introduction

In this lab, the objectives are to:

- Learn to design FIR filters using Matlab
- Implement the FIR filter using the C6713 DSK system in real-time in C
- Make the FIR filter operate as fast as possible
- Measure the filter characteristics using a network or spectrum analyzer

The focus of this lab is to design a FIR filter in Matlab using functions such as `firpm, firpmord` and `freqz`. Once the filter has been designed, it will be implemented on the DSK board which allows for real-time filtration of the signal from the Signal Generator. The rest of the lab is centred on speeding up the filter and reducing the clock cycles required to run the ISR (Interrupt Service Routine).

This report will cover how the filter was designed in Matlab, its implementation in Code Composer Studio, and how it was optimised to within three times the order of the filter. An in-depth analysis of experimental measurements will also be presented.

The tools used in this lab include Code Composer Studio, Signal Generator, the Texas Instrument C6713 processor, a Tektronix TDS2012C oscilloscope, and Audio Precision APX520.

Filter Design Using Matlab

Finite Impulse Response Filters

A finite impulse response filter is a filter whose impulse response is of finite duration, meaning it reaches zero in finite time.

For a causal discrete-time FIR filter of order N, each value of the output sequence is a weighted sum of the most recent input values:

$$y[n] = b[0]x[n] + b[1]x[n-1] + b[2]x[n-2] + \cdots + b[N]x[n-N]$$
$$= \sum_{i=0}^{N} b[i]x[n-i]$$

Where

- $x[n]$ is the input signal
- $y[n]$ is the output signal
- N is the order of the filter
- $b[i]$ is the coefficient of the filter at the ith instant

Taking the Z-Transform of the convolution sum the result is as follows:

$$H[z] = b[0] + \frac{b[1]}{z} + \cdots + \frac{b[N]}{z[N]} = \sum_{i=0}^{N} b[i]z^{-i}$$

Which can be visually represented by

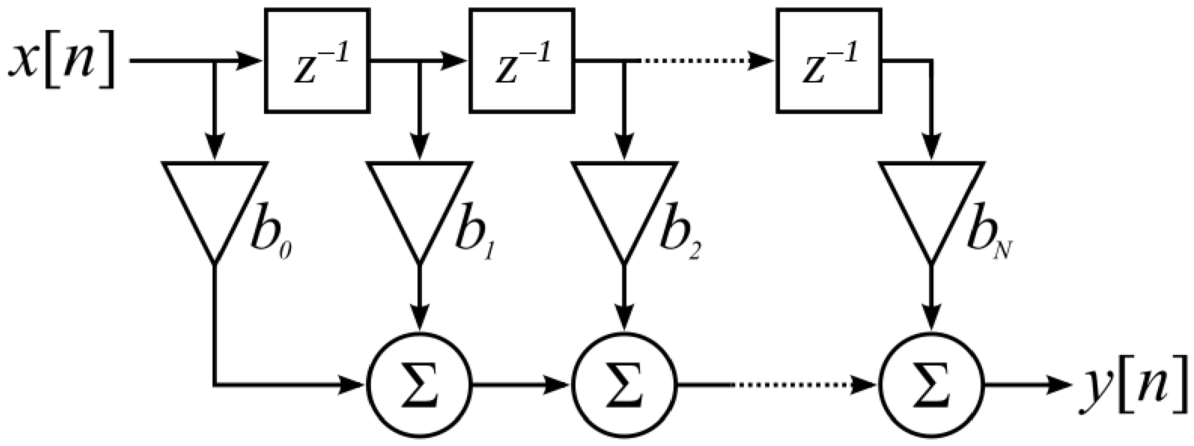


Figure 1 - Visual representation of an FIR filter

FIR filters have some useful properties which will be exploited later on in this report. These include:

- No feedback is required, making implementation easier, and also means that rounding errors are not compounded upon.
- FIR filters are inherently stable, since the output is a sum of a finite number of multiplications.
- They can be designed to be linear phase by making the coefficient sequence symmetric. In particular, this property of the FIR filter will be used to improve the speed of the filter.

FIR design in Matlab

The following specification was provided:

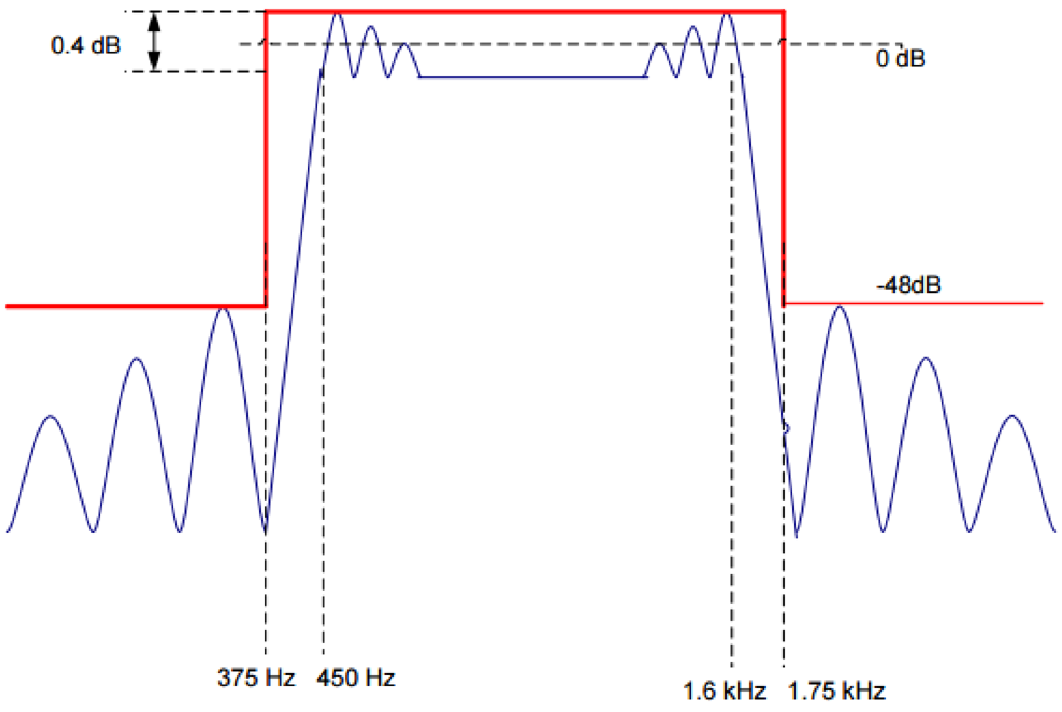


Figure 2 - Specification

This can be summarized as follows:

- Passband – 450Hz to 1600Hz
- Passband ripple – 0.4dB
- Transition band – 375Hz to 450Hz & 1600Hz to 17500Hz
- Stopband ripple - -48dB
- Stopband – 0Hz to 375Hz & f > 1750Hz

Description of Matlab functions used

Firpm

Firpm is the Parks-McClellan optimal FIR filter design. It designs a linear-phase FIR filter using the Parks-McClellan algorithm. The filters are optimal in the sense that the maximum error between the desired frequency response and the actual frequency response is minimized. Filters designed this way exhibit an equiripple behaviour in their frequency responses and are sometimes called equiripple filters.

Firpmord

Firpmord is the Parks-McClellan optimal FIR filter order estimation. In other words, firpmord, finds the approximate order, normalized frequency band edges, frequency band amplitudes, and weights that meet input specifications.

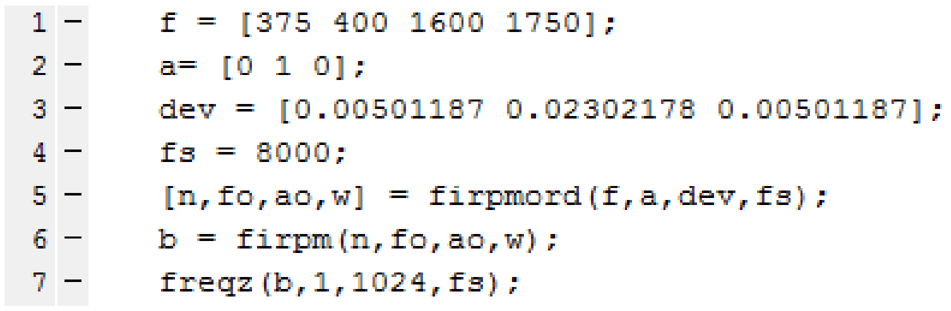
Freqz

Freqz provides the frequency response of the filter based on the filter coefficients.

Implementing the given specifications

When the specification is implemented to design a filter, the frequency response is not symmetrical and is not a particularly good approximation to the required filter.

Below shows the Matlab code used to generate the filter and plot the frequency response.



```
f = [375 400 1600 1750];
a= [0 1 0];
dev = [0.00501187 0.02302178 0.00501187];
fs = 8000;
[n,fo,ao,w] = firpmord(f,a,dev,fs);
b = firpm(n,fo,ao,w);
freqz(b,1,1024,fs);
```

Figure 3 - Matlab code for given specification

The first line of the code initialises an array, f, containing the first stopband frequency, then the first passband frequency, followed by the second passband frequency and finally the second stopband frequency as per specification.

The second line again initialises an array, a, which is used for the gain of the filter.

dev is an array used to store the absolute ripple values given in the specification, ie not in dBs. The stopband ripple (0.00501187), can be calculated using the formula $\frac{V_{out}}{V_{in}} = 10^{-46}/_{20} = 0.00501187$. The passband ripple can be calculated using $\frac{10^{0.4}/_{20} - 1}{10^{0.4}/_{20} + 1} = 0.02302178$.

The given sampling frequency is 8000Hz.

firpmord is then used to calculate the approximate order, normalized frequency band edges, frequency band amplitudes, and weights.

firpm takes as an input these calculated variables and generates the coefficients required to satisfy the specifications, which is stored in b.

freqz is used to plot the magnitude and phase response of the designed filter in order to check the response is as expected.

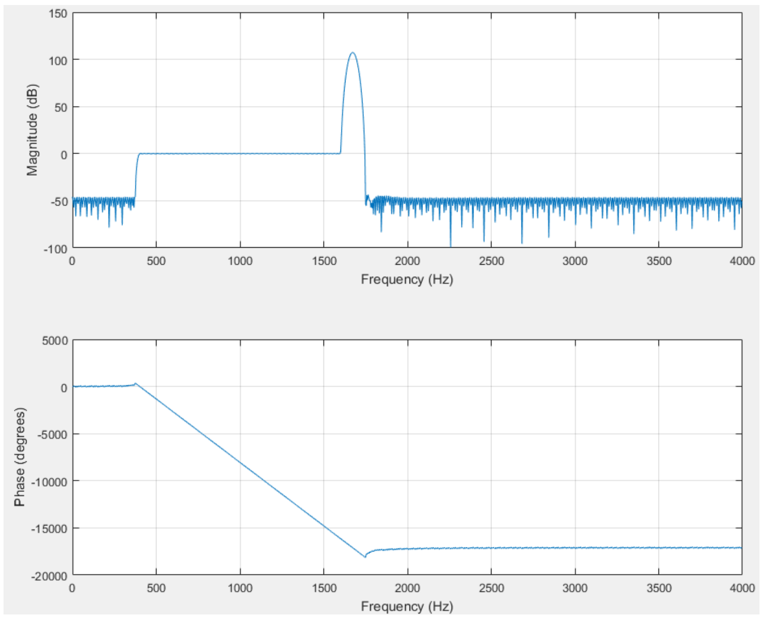


Figure 4 - Frequency and Phase response for given specification

It is clear to see from Figure 4 that the frequency response of this filter is not a good approximation to the required filter.

Currently the specification is too tight to achieve a good approximation to the required filter. In order to reduce this error, there are multiple possible solutions. One way to 'loosen' the specification is to allow more of a passband and or decrease the ripple in the stopband. Whilst this does reduce the error at the end of the passband, this is not ideal at the pass band ripple is more noticeable and could have an effect on the signal. Another way to achieve a better frequency response would be to increase the passband bandwidth.

Instead of the given frequencies $f = [375, 400, 1600, 1750]$, the frequencies used are $f = [375, 410, 1665, 1700]$. This not only removes the error, but also makes the filter symmetrical as the transition periods are of the same bandwidth.

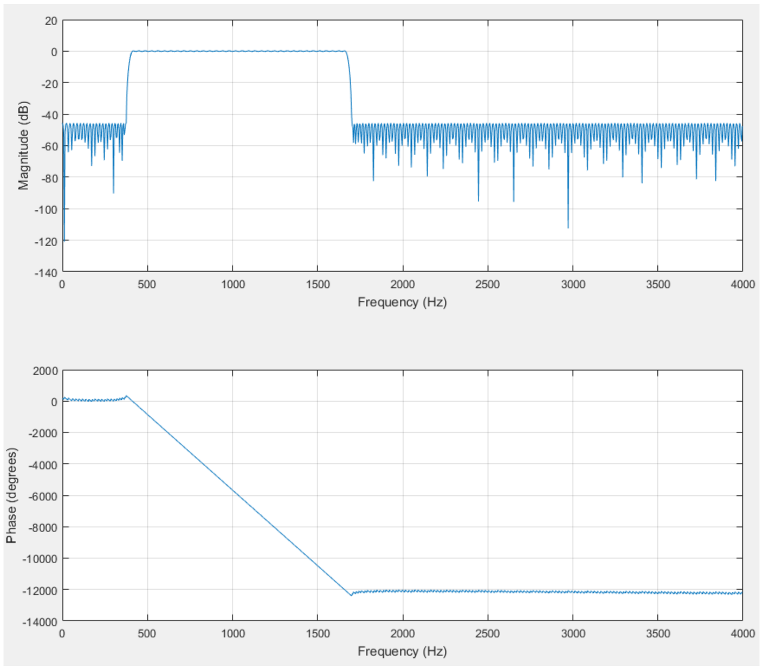
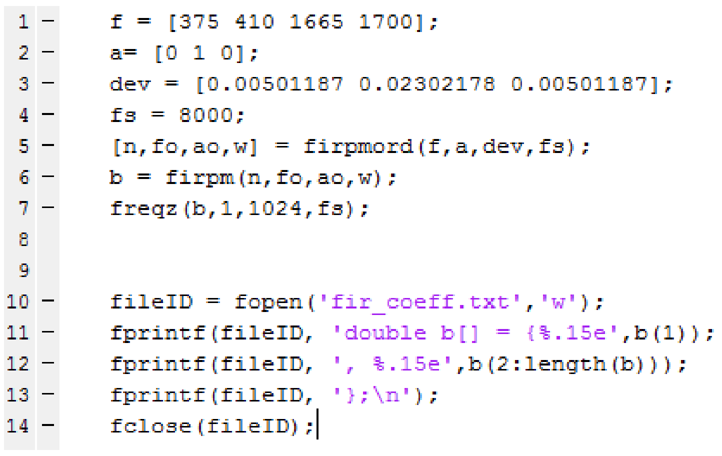


Figure 5 - Frequency response with altered specification



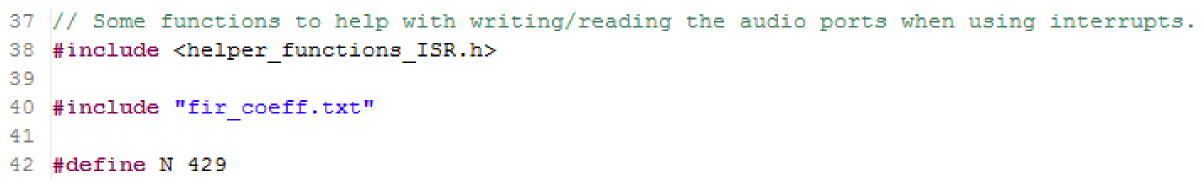
```
f = [375 410 1665 1700];
a= [0 1 0];
dev = [0.00501187 0.02302178 0.00501187];
fs = 8000;
[n,fo,ao,w] = firpmord(f,a,dev,fs);
b = firpm(n,fo,ao,w);
freqz(b,1,1024,fs);


fileID = fopen('fir_coeff.txt','w');
fprintf(fileID, 'double b[] = {%.15e',b(1));
fprintf(fileID, ', %.15e',b(2:length(b)));
fprintf(fileID, '};\n');
fclose(fileID);
```

Figure 6 - Matlab code with altered specification

As seen in Figure 5, the frequency response is much closer to an ideal filter. The code in Figure 6 is used to generate the required the required filter coefficients and also extracts them to a text file to later be included in the C program.

FIR implementation

Once the FIR has been designed in Matlab the coefficients can be included within the inito.c file as shown in Figure 7.



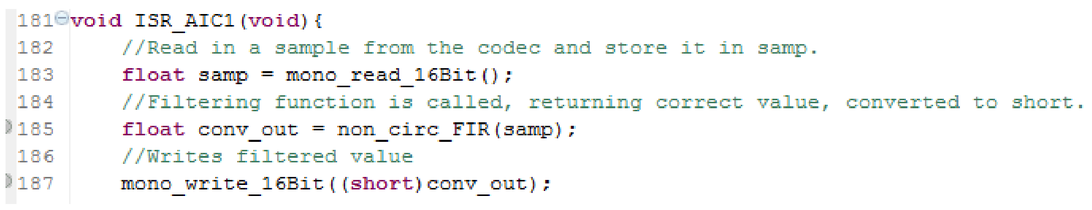
```
// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

#include "fir_coeff.txt"

#define N 429
```

Figure 7 - Including the filter coefficients

In order to implement the filter, there are four tasks that must be done.

1. Read the input sample from the codec
2. Perform the delay operator
3. Perform the convolution function
4. Output y[n] to the codec

ISR Function



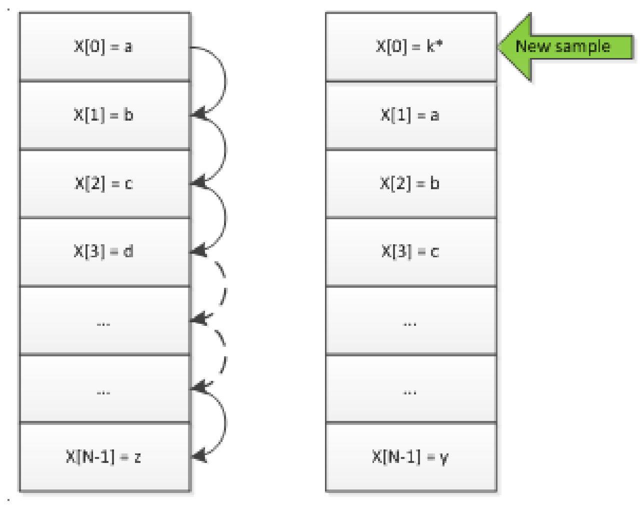
```
181 void ISR_AIC1(void){
182     //Read in a sample from the codec and store it in samp.
183     float samp = mono_read_16Bit();
184     //Filtering function is called, returning correct value, converted to short.
185     float conv_out = non_circ_FIR(samp);
186     //Writes filtered value
187     mono_write_16Bit((short)conv_out);
```

Figure 8 - ISR Function

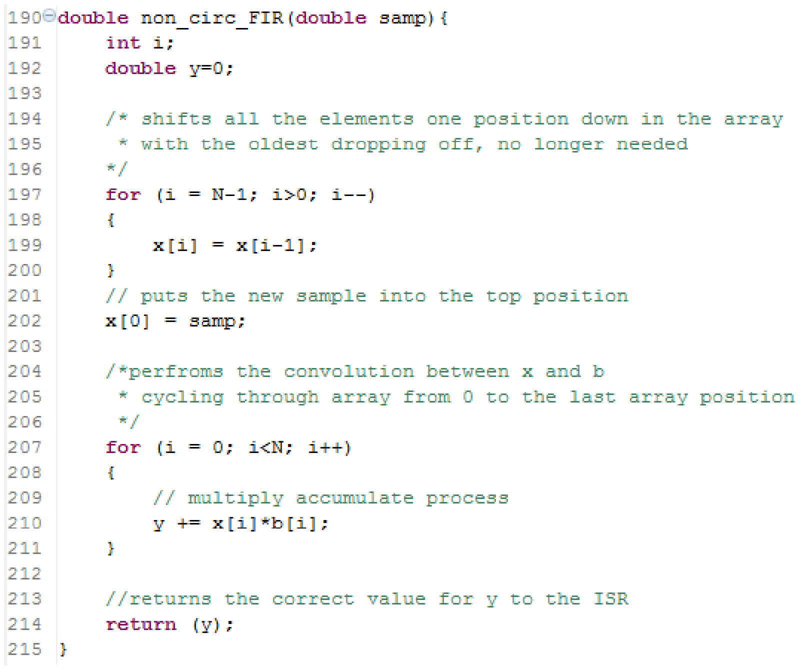
The interrupt service routine is called at the sampling frequency to filter the new input sample. The 16 bit sample is read in and stored in the float ‘`samp`’, this is then passed to the FIR filtering function, which is Figure 8 would be `non_circ_FIR(samp)` (line 185). The FIR filter returns a float value, with the filtered sample value. This is then converted back into a 16 bit ‘`short`’ number from a float. It is then written back to the codec.

Basic non-circular FIR filter in C

Explanation



The first implementation of the code uses a standard buffer of size N. The samples that are read in are stored in the buffer. The buffer is updated by shifting all the values one place down in the array, with the oldest sample that is no longer need dropping off the end. The new sample is then inserted at the top of the array. This array ‘`x`’ containing the samples, corresponds directly with the filter coefficients stored in array ‘`b`’.



```c
double non_circ_FIR(double samp){
    int i;
    double y=0;

    /* shifts all the elements one position down in the array
     * with the oldest dropping off, no longer needed
    */
    for (i = N-1; i>0; i--)
    {
        x[i] = x[i-1];
    }
    // puts the new sample into the top position
    x[0] = samp;

    /*perfroms the convolution between x and b
     * cycling through array from 0 to the last array position
     */
    for (i = 0; i<N; i++)
    {
        // multiply accumulate process
        y += x[i]*b[i];
    }

    //returns the correct value for y to the ISR
    return (y);
}
```

Figure 9 - non_circ_FIR

The code uses a for loop to shift the stored samples one position, and then putting the new sample at the top. A second for loop is then used to perform the convolution between array ‘`x`’ and array ‘`b`’, which consists of N multiply accumulate processes. The same element in both arrays is multiplied together, and then added to the variable `y`.

Once the entire convolution is complete, the variable `y` – which is type double – is returned to the ISR function.

Oscilloscope traces showing expected operation of implemented filter

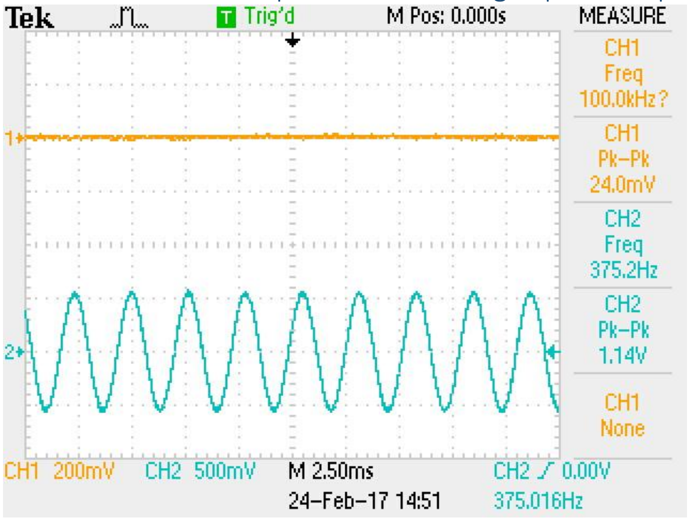


Figure 10 - Scope trace at 375Hz

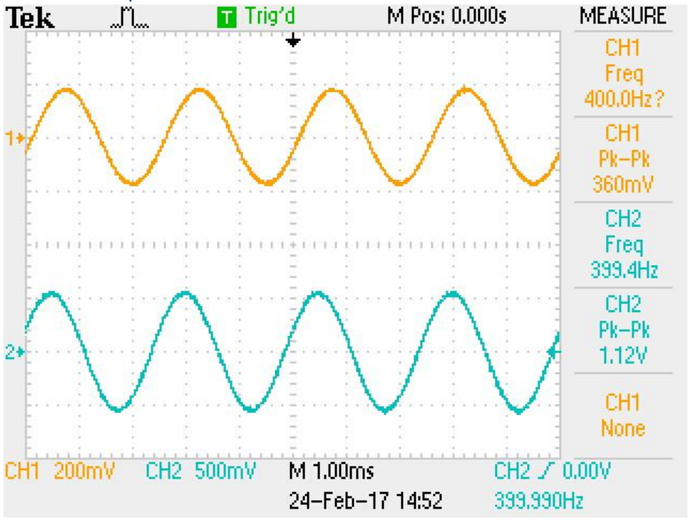


Figure 11 - Scope trace at 400Hz

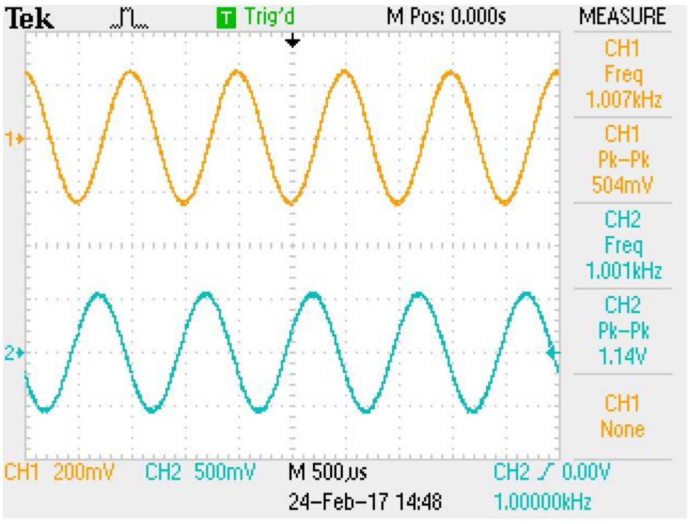


Figure 12 - Scope trace at 1000Hz

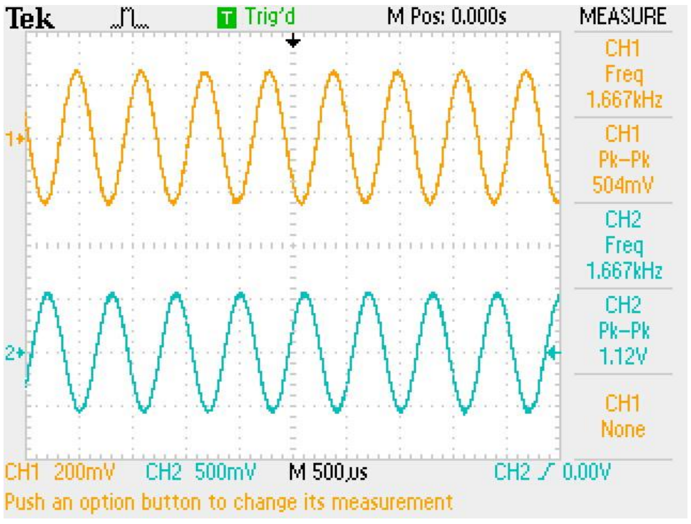


Figure 13 - Scope trace at 1665Hz

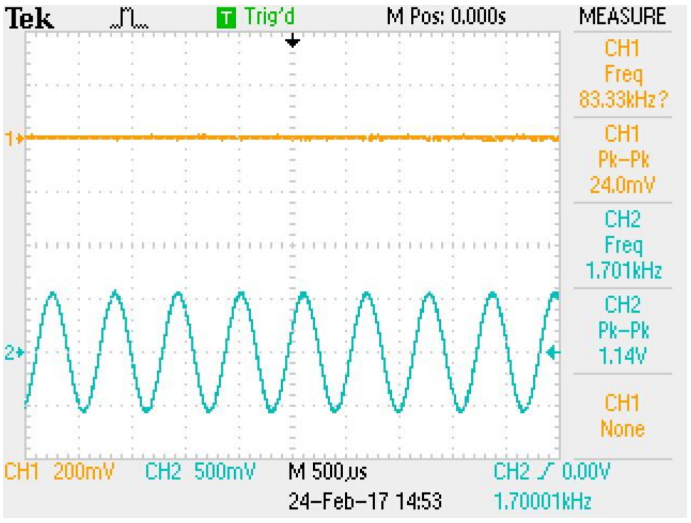


Figure 14 - Scope trace at 1700Hz

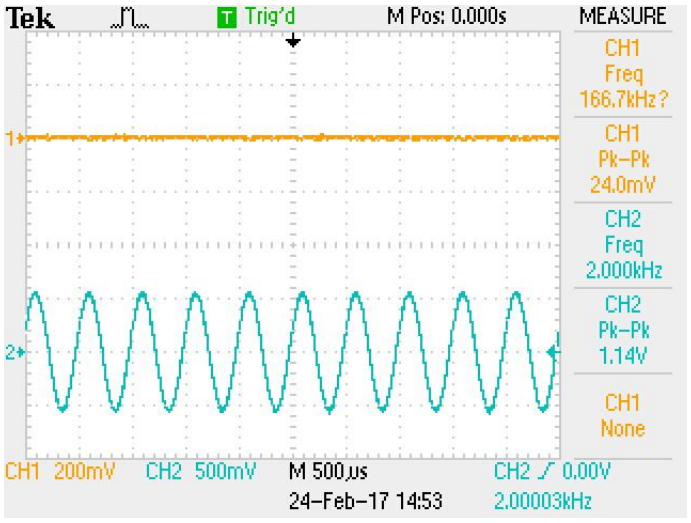


Figure 15 - Scope trace at 2000Hz

As can be seen from the oscilloscope traces provided, the filter operates as expected where the passband is from 410Hz to 1665Hz. The orange line is the filtered output signal and the blue line is the input signal. It should be noted that in Figure 11, with the frequency set to 400Hz, the peak to peak amplitude of the output is less than that of the output signals in Figure 12 and 13. Since 400Hz is in the transition band, the gain is expected to be less than that of the passband. The peak to peak voltage in the passband is approximately half of the input signal, due to the potential divider at the input of the audio chip.

Benchmarking

For the benchmarking, breakpoints were placed in the ISR and using the clock along with various optimisation options. The three optimisation options were:

1. No optimisation
2. Option set to –o0
3. Option set to –o2

non_circ_fir		
no optimisation /clock cycles	-o0 /clock cycles	-o2 /clock cycles
27127	22341	2944
27127	22341	2945
27126	22342	2944
27127	22342	2944
27127	22342	2944
Minimum		
27126	22341	2944

Table 1 - non_circ_FIR benchmarks

Circular FIR filter in C

Explanation

The first optimisation was to replace the standard buffer with a circular buffer. This improves the speed of the code as the for loop to shift values in the array can be removed. The circular buffer works by using a ‘make-shift’ pointer to indicate the newest sample in the array, which indicates where the convolution for loop starts. When the end of the array is reached, it wraps around to the start of the array and continuing through the array until it reaches the oldest value. The red line in the Figures 16 and 17 indicates the split in the buffer between the oldest and the newest samples, starting from the newest sample moving down through the buffer until it reaches the end of the array when it wraps around to the first element in the array continuing to the oldest sample – directly above the red line.

The oldest sample is overwritten by the newest sample when it is updated, as the oldest sample is no longer needed. The pointer to the start value is then updated.

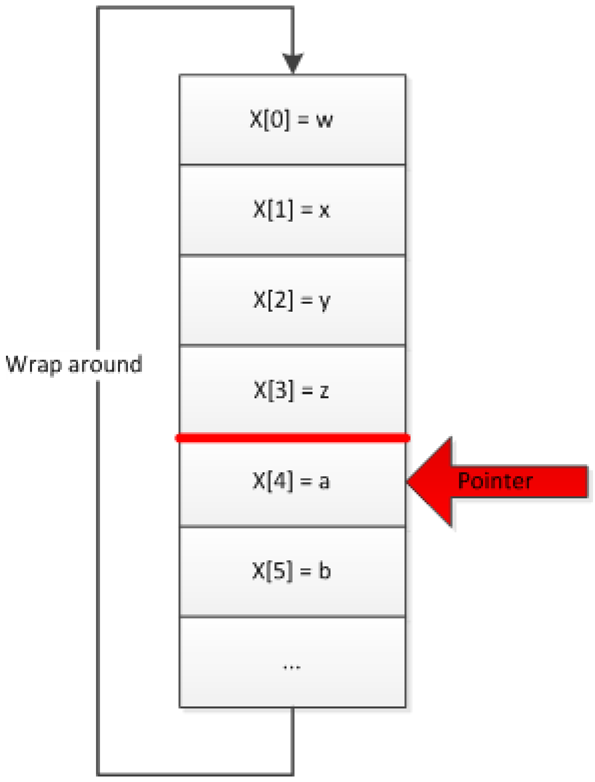


Figure 16 - Circular buffer explanation

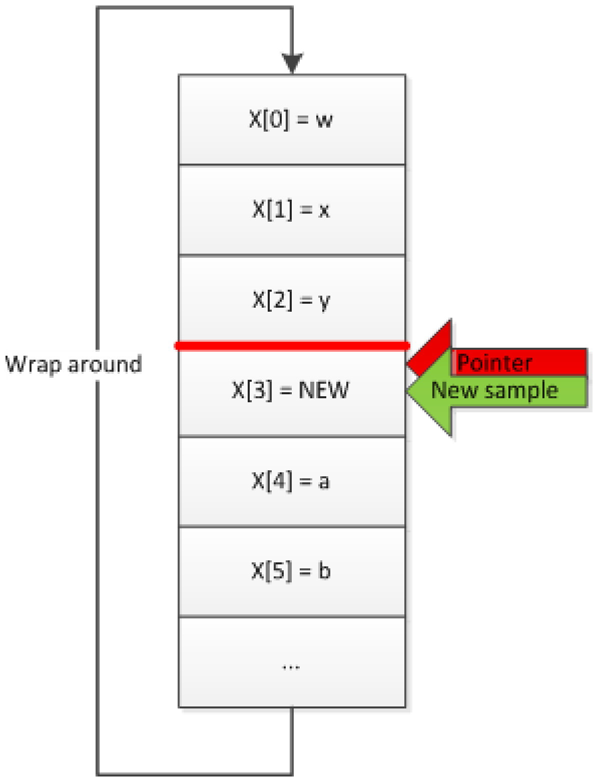
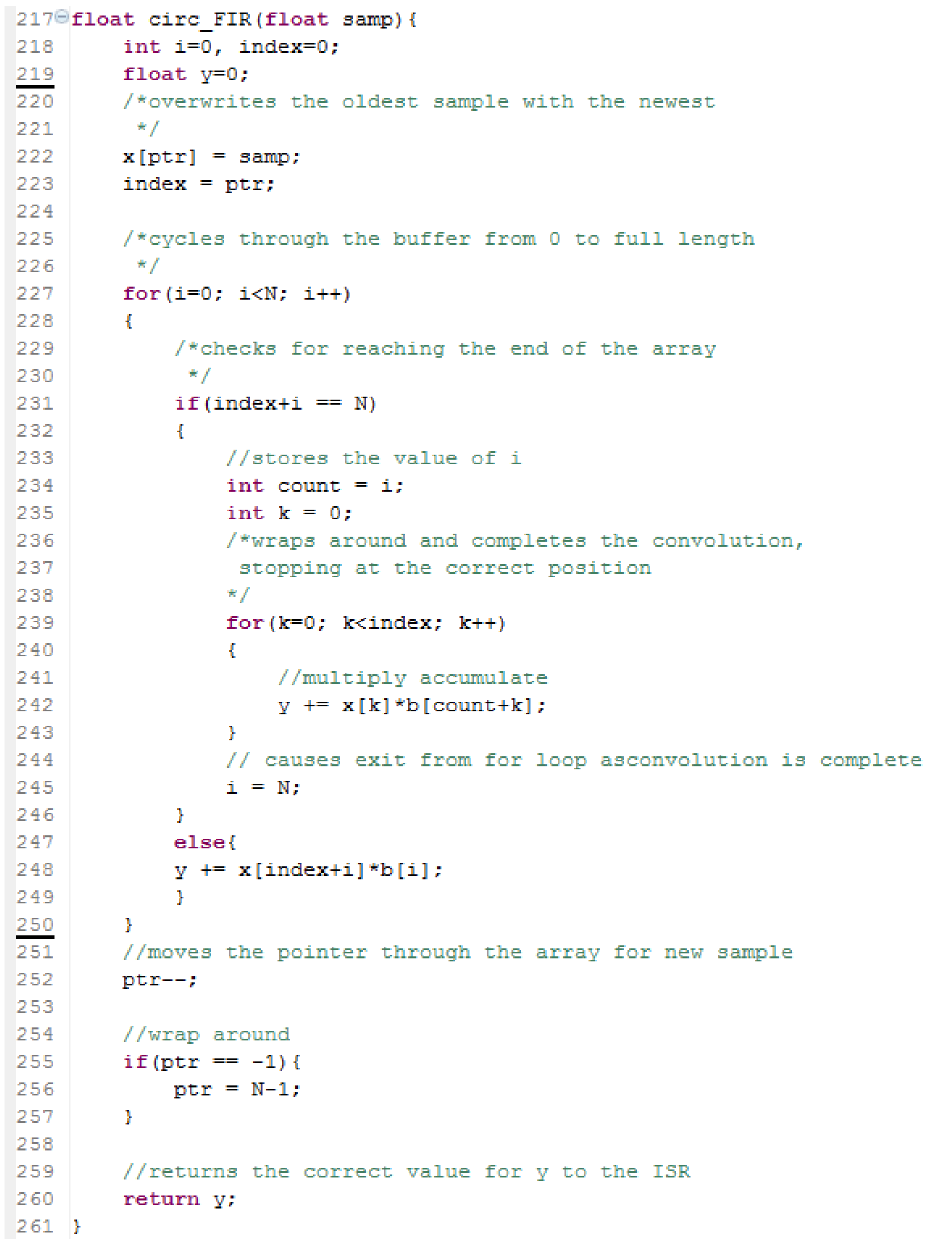


Figure 17 - Circular buffer explanation

The circular buffer function takes the newest read sample, and replaces the oldest value using the global variable '`ptr`' which was updated in the previous call of the function. The value of '`ptr`' is then stored in the variable '`index`'. A for loop is used, with N cycles to perform the convolution. With each iteration of the for loop, the value of '`index+i`' is checked to see if it has reached the end of the array. If it has then it wraps around to the start of the array. The value of '`i`' which is used for the for loop, is moved the variable '`count`'. A second for loop is the entered to complete the convolution from the start of the array to the oldest sample. Variable '`i`' is then set to value N, causing the first for loop to finish. If the last element has not been reached in the array, the if statement is not triggered and the multiply accumulate happens as previously in the standard buffer. Once the for loop for the convolution has completed the pointer for the start element is decremented, ready for the new sample to be written into the array. Again a check must be implemented for wrap around, and if it reaches the top of the array the pointer is moved back to the bottom. The value of '`y`' is returned to the ISR.



```c
float circ_FIR(float samp){
    int i=0, index=0;
    float y=0;
    /*overwrites the oldest sample with the newest
     */
    x[ptr] = samp;
    index = ptr;

    /*cycles through the buffer from 0 to full length
     */
    for(i=0; i<N; i++)
    {
        /*checks for reaching the end of the array
         */
        if(index+i == N)
        {
            //stores the value of i
            int count = i;
            int k = 0;
            /*wraps around and completes the convolution,
             stopping at the correct position
            */
            for(k=0; k<index; k++)
            {
                //multiply accumulate
                y += x[k]*b[count+k];
            }
            // causes exit from for loop asconvolution is complete
            i = N;
        }
        else{
        y += x[index+i]*b[i];
        }
    }
    //moves the pointer through the array for new sample
    ptr--;

    //wrap around
    if(ptr == -1){
        ptr = N-1;
    }

    //returns the correct value for y to the ISR
    return y;
}
```

Figure 18 - circ_FIR

Benchmarking

circ_fir		
no optimisation /clock cycles	-o0 /clock cycles	-o2 /clock cycles
18124	15927	3935
18128	15935	4134
18132	15943	4153
18136	15948	4172
18140	15959	3980
Minimum		

18124	15927	3935

Table 2 - circ_FIR benchmarks

Symmetric, circular FIR filter in C

Explanation

On further inspection of the filter, it can be seen that the filter coefficients from Matlab are mirrored around the centre of the array ‘`b`’. Therefore the newest and oldest element will be multiplied by the same coefficient of ‘`b`’, and the second newest and the second to last oldest by the same coefficient of ‘`b`’ and so on until the middle position is reached. Figure 19 below illustrates this, with the blue line showing where the ‘mirror’ line is.

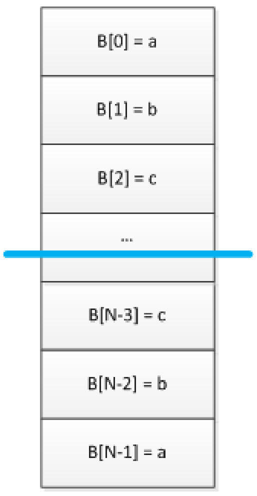
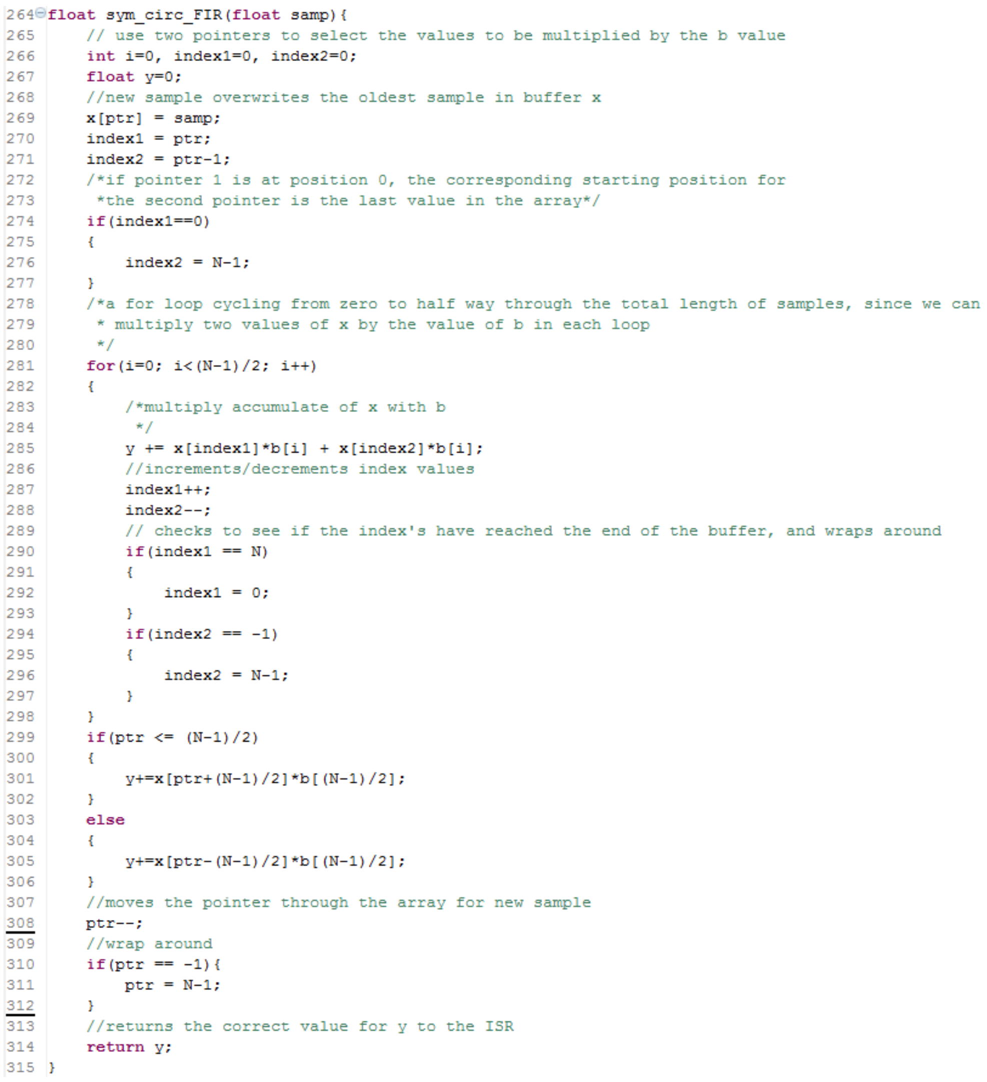


Figure 19 - Symmetric coefficients explanation



```c
float sym_circ_FIR(float samp){
    // use two pointers to select the values to be multiplied by the b value
    int i=0, index1=0, index2=0;
    float y=0;
    //new sample overwrites the oldest sample in buffer x
    x[ptr] = samp;
    index1 = ptr;
    index2 = ptr-1;
    /*if pointer 1 is at position 0, the corresponding starting position for
     *the second pointer is the last value in the array*/
    if(index1==0)
    {
        index2 = N-1;
    }
    /*a for loop cycling from zero to half way through the total length of samples, since we can
     * multiply two values of x by the value of b in each loop
     */
    for(i=0; i<(N-1)/2; i++)
    {
        /*multiply accumulate of x with b
         */
        y += x[index1]*b[i] + x[index2]*b[i];
        //increments/decrements index values
        index1++;
        index2--;
        // checks to see if the index's have reached the end of the buffer, and wraps around
        if(index1 == N)
        {
            index1 = 0;
        }
        if(index2 == -1)
        {
            index2 = N-1;
        }
    }
    if(ptr <= (N-1)/2)
    {
        y+=x[ptr+(N-1)/2]*b[(N-1)/2];
    }
    else
    {
        y+=x[ptr-(N-1)/2]*b[(N-1)/2];
    }
    //moves the pointer through the array for new sample
    ptr--;
    //wrap around
    if(ptr == -1){
        ptr = N-1;
    }
    //returns the correct value for y to the ISR
    return y;
}
```

Figure 20 - sym_circ_FIR

Firstly a few variables are initialised which will be used to keep track of various positions in the buffer. Then the new sample read in from the codec will be placed into the array at the position the pointer is pointing to. ‘`index1`’ is then assigned to the value of ‘`ptr`’ which is the position of the newest sample in the array and ‘`index2`’ is assigned to ‘`ptr-1`,’ which is the oldest sample in the array. The if statement then checks whether '`index1`' is equal to 0, in which case '`index2`' needs to be wrapped around to N-1. The for loop is then started which cycles from 0 to one below halfway through the number of coefficients. Since the number of coefficients is odd, the middle case for the multiply accumulate will be explained later. In the for loop the usual multiply accumulate occurs. After this '`index1`' is incremented by one and '`index2`' is decremented by one in order to keep track of which samples need to be multiplied by the same coefficient. Since this is a circular buffer, a check must be done when the end of the buffer is reached by '`index1`' or when the beginning of the buffer is reached by '`index2`'. When '`index1`' reaches the end of the buffer, it is wrapped around to 0, whilst when '`index2`' reaches the beginning of the buffer it is wrapped around to N-1. This deals with all of the samples except for the middle one. In this implementation, a check must be done to see whether '`ptr`' points to a location in the first or second half of the buffer. If it is the former, the middle case will be in the second half of the buffer and therefore the middle value of the length of the array must be added to the initial '`ptr`' position to receive the middle value for the multiply accumulate. If '`ptr`' points to the second half of the buffer then the middle multiply accumulate will be in the first half of the buffer, therefore half the length of the array needs to be subtracted from '`ptr`.' Once these checks have been completed, the value of '`ptr`' is decremented to allow for the next sample to be placed into the correct position. A check on whether '`ptr`' needs to be wrapped around is then initiated.

Benchmarking

sym_circ_FIR		
no optimisation /clock cycles	-o0 /clock cycles	-o2 /clock cycles
12450	10781	2232
12484	10776	2223
12642	10793	2230
12465	10797	2221
12453	10777	2232
Minimum		
12642	10776	2221

Table 3 - sym_circ_FIR benchmarks

Double buffer, symmetric, circular FIR filter in C

Explanation

To improve the speed further, it is best to reduce the number of processes in the for loop. By creating a ‘double buffer’, a buffer of twice the size, 2N, with the samples repeated – as seen in the Figure 21. As with a circular buffer, there is a pointer indicating the starting position that cycles through. In the double buffer the pointer wraps around when it gets to the N-1 position in the array, so only points to the top half of the array. The red line in the diagram indicates the separator between the newest and oldest sample. As the samples are repeated, in the second half of the double buffer continues in the correct order. This allows for the removal of the check in the for loop for the wrap around. Figure 21 indicates that all the samples are between the two red lines.

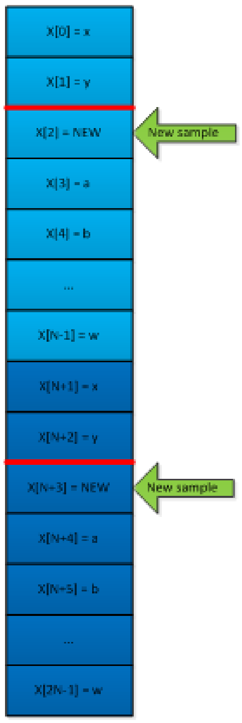
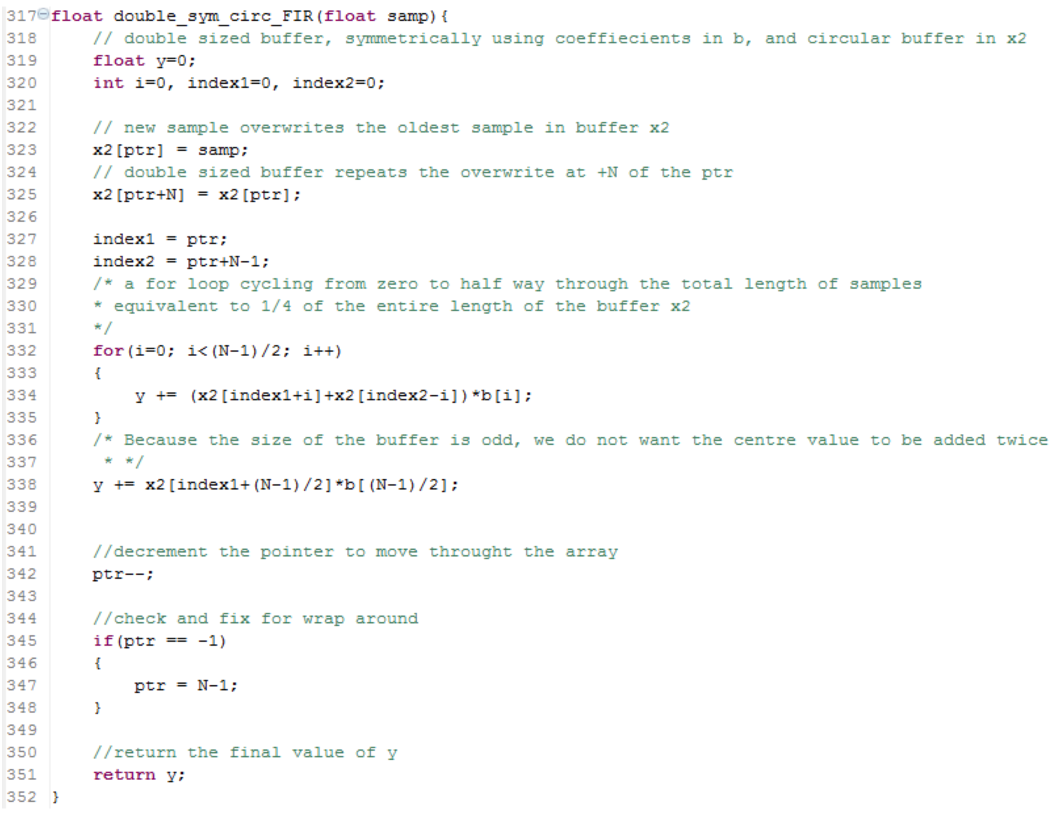


Figure 21 - Double buffer explanation



```c
float double_sym_circ_FIR(float samp){
    // double sized buffer, symmetrically using coeffiecients in b, and circular buffer in x2
    float y=0;
    int i=0, index1=0, index2=0;

    // new sample overwrites the oldest sample in buffer x2
    x2[ptr] = samp;
    // double sized buffer repeats the overwrite at +N of the ptr
    x2[ptr+N] = x2[ptr];

    index1 = ptr;
    index2 = ptr+N-1;
    /* a for loop cycling from zero to half way through the total length of samples
    * equivalent to 1/4 of the entire length of the buffer x2
    */
    for(i=0; i<(N-1)/2; i++)
    {
        y += (x2[index1+i]+x2[index2-i])*b[i];
    }
    /* Because the size of the buffer is odd, we do not want the centre value to be added twice
     * */
    y += x2[index1+(N-1)/2]*b[(N-1)/2];


    //decrement the pointer to move throught the array
    ptr--;

    //check and fix for wrap around
    if(ptr == -1)
    {
        ptr = N-1;
    }

    //return the final value of y
    return y;
}
```

Figure 22 - double_sym_circ_FIR

The ‘`double_sym_circ_FIR(float samp)`’ function works as follows, the new sample writes over the oldest sample in the double buffer at the the ‘`ptr`’ position and ‘`ptr+N-1`’. This inserts the new samples in the double buffer, as seen in Figure 21. The indexes are then initialised with the correct values for the oldest and newest samples. The for loop is then entered using the symmetry of ‘`b`’ to add two values on each iteration as discussed above. As the size of coefficients array ‘`b`’ is odd in this case, it is added outside the for loop so the central value is only added once. Again the ‘`ptr`’ is then updated ready for the next cycle. The correct value for y is then returned to the ISR. A diagram of the for loop can be seen below.

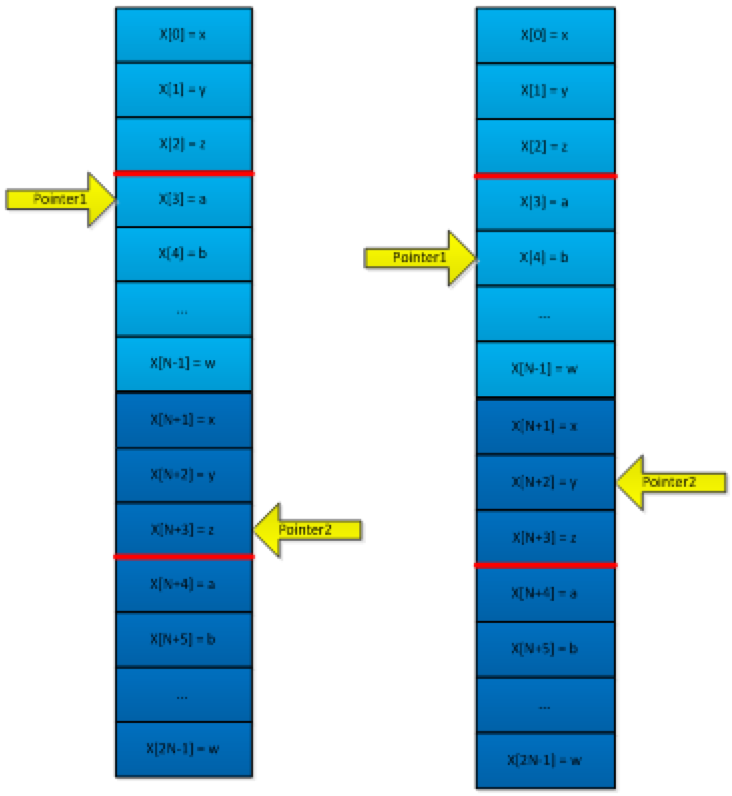


Figure 23 - Visual representation of for loop

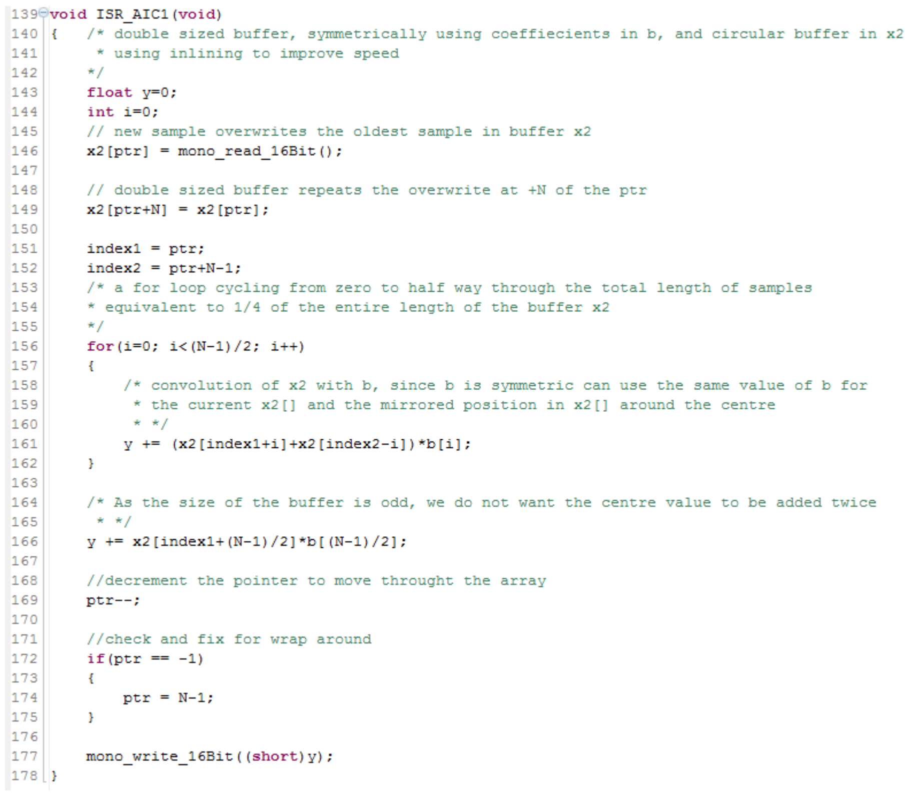
Benchmarking

No inlining

double_sym_circ_FIR		
no optimisation /clock cycles	-o0 /clock cycles	-o2 /clock cycles
8255	6693	531
8255	6693	531
8255	6693	530
8255	6693	536
8259	6693	536
Minimum		
8255	6693	530

Table 4 - double_sym_circ_FIR benchmarks

With inlining



```
void ISR_AIC1(void)
{   /* double sized buffer, symmetrically using coeffiecients in b, and circular buffer in x2
     * using inlining to improve speed
    */
    float y=0;
    int i=0;
    // new sample overwrites the oldest sample in buffer x2
    x2[ptr] = mono_read_16Bit();

    // double sized buffer repeats the overwrite at +N of the ptr
    x2[ptr+N] = x2[ptr];

    index1 = ptr;
    index2 = ptr+N-1;
    /* a for loop cycling from zero to half way through the total length of samples
    * equivalent to 1/4 of the entire length of the buffer x2
    */
    for(i=0; i<(N-1)/2; i++)
    {
        /* convolution of x2 with b, since b is symmetric can use the same value of b for
         * the current x2[] and the mirrored position in x2[] around the centre
         * */
        y += (x2[index1+i]+x2[index2-i])*b[i];
    }

    /* As the size of the buffer is odd, we do not want the centre value to be added twice
     * */
    y += x2[index1+(N-1)/2]*b[(N-1)/2];

    //decrement the pointer to move throught the array
    ptr--;

    //check and fix for wrap around
    if(ptr == -1)
    {
        ptr = N-1;
    }

    mono_write_16Bit((short)y);
}
```

Figure 24 – Inlined code for least clock cycles

Inlining is an optimisation that replaces the function call with the body of the function itself, therefore improving the performance of the program by bypassing the function call.

double_sym_circ_FIR		
no optimisation /clock cycles	-o0 /clock cycles	-o2 /clock cycles
8542	7437	501
8541	7437	501
8541	7437	501
8541	7437	502
8545	7441	501
Minimum		
8541	7437	**501**

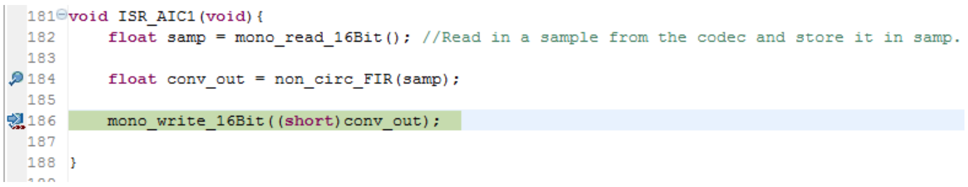
Table 5 - double_sym_circ_FIR with inlining benchmarks

This method of implementing the FIR filter is the most optimised technique in C and brings the clock cycles to 1.17 times the number of filter coefficients. In order to obtain significantly fewer clock cycles, one would have to implement the filter in assembly.

Double vs. Float

After implementing the non circular buffer for the FIR filter, it was realised that since the processor being used is a floating point processor, another optimisation that could be implemented would be to change the variables in the program to be of type float. Since the hardware implements float only, emulating double in software costs time. Therefore using type float saves a significant number of clock cycles.

This can be shown via the following figure.



```
void ISR_AIC1(void){
    float samp = mono_read_16Bit(); //Read in a sample from the codec and store it in samp.

    float conv_out = non_circ_FIR(samp);

    mono_write_16Bit((short)conv_out);

}
```

Figure 25 - ISR for non_circ_FIR with float implementation

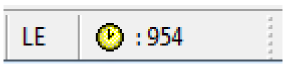
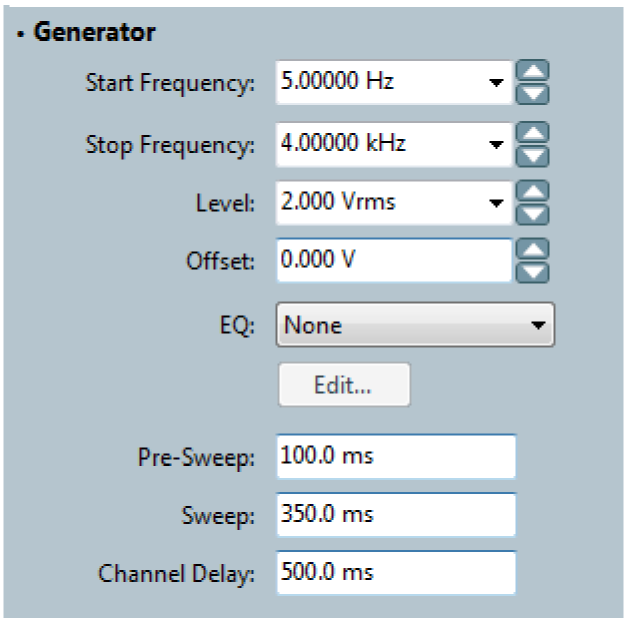


Figure 26 - Clock cycles

As seen in Figure 26, 954 clock cycles are being achieved with -o2 optimisation compared to the 2994 seen in Table 1, a significant improvement.

Frequency response from network analyser

In this part of the report the Audio Precision APX520 spectrum analyser is used to measure the frequency response of the working system in real-time.



Generator

Start Frequency: 5.00000 Hz
Stop Frequency: 4.00000 kHz
Level: 2.000 Vrms
Offset: 0.000 V
EQ: None
Edit...
Pre-Sweep: 100.0 ms
Sweep: 350.0 ms
Channel Delay: 500.0 ms

Figure 27 - Setup of network analyser

Figure 27 shows the setup variables used as per specification.

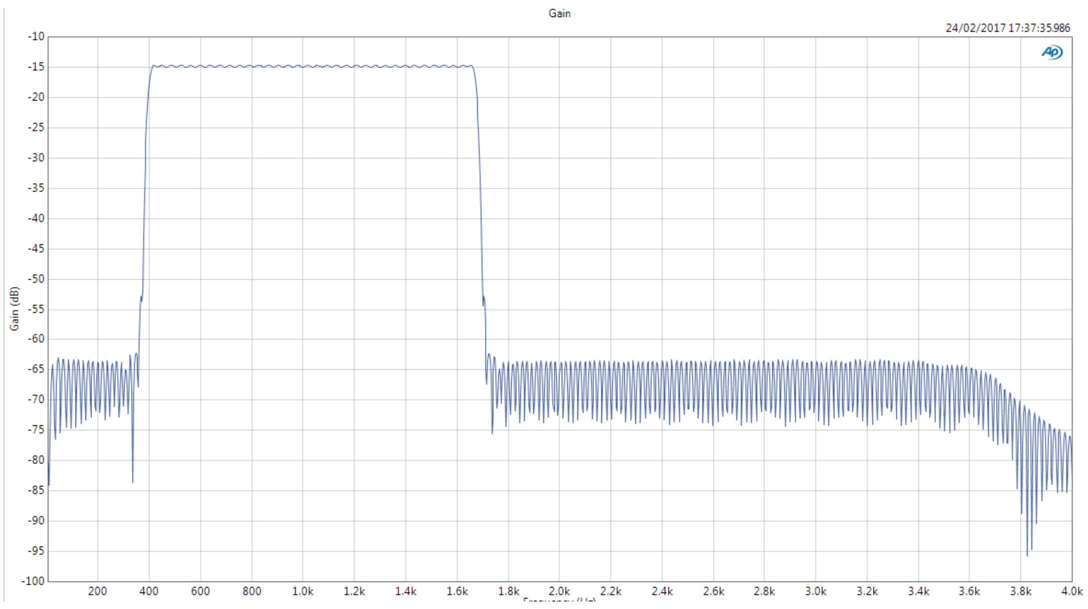


Figure 28 - Gain plot of fastest filter from network analyser

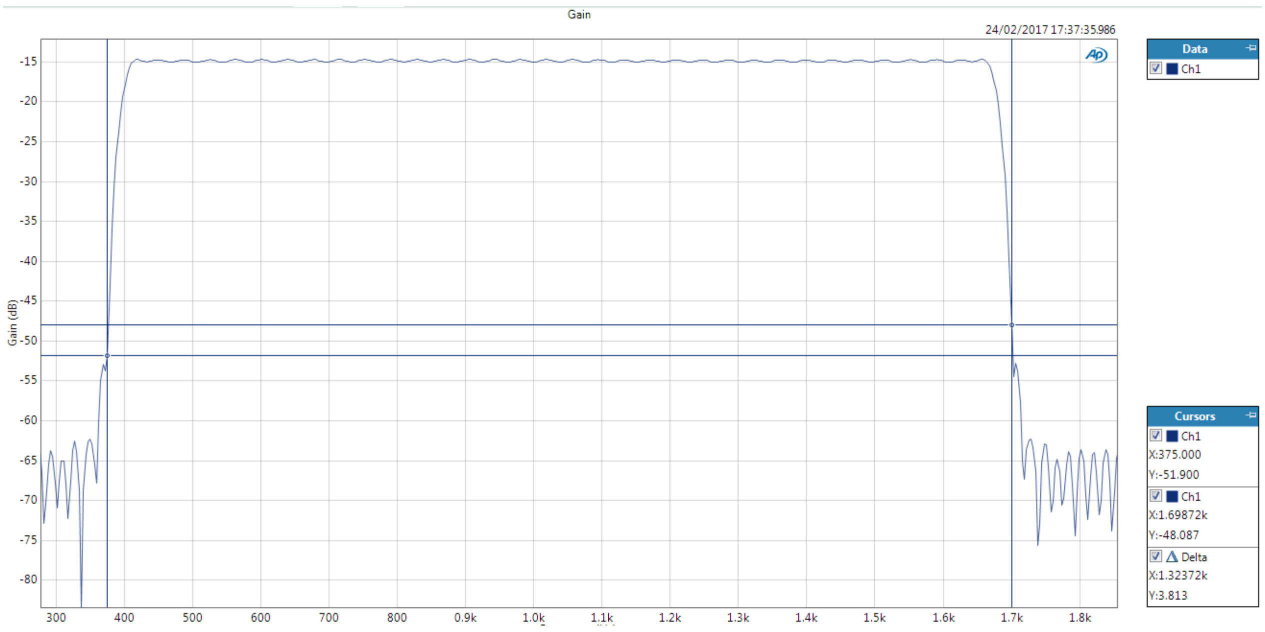


Figure 29 - Close up of gain plot of fastest filter from network analyser

From Figure 28 and 29, it is clear to see that the FIR filter is being implemented correctly as it is similar to the plot generated in Matlab (Figure 5). Figure 29 shows a detailed view of the significant parts of the gain plot. Cursors were placed at important parts of the filter, 375Hz and 1700Hz, where it can be seen that the attenuation is at approximately -50dB. The passband attenuation is close to -14dB, where it should be -12dB. This is to do with the potential dividers at the input stage of the DSK unit as shown in Figure 30. Since $20\log(0.25) \approx -12dB$.

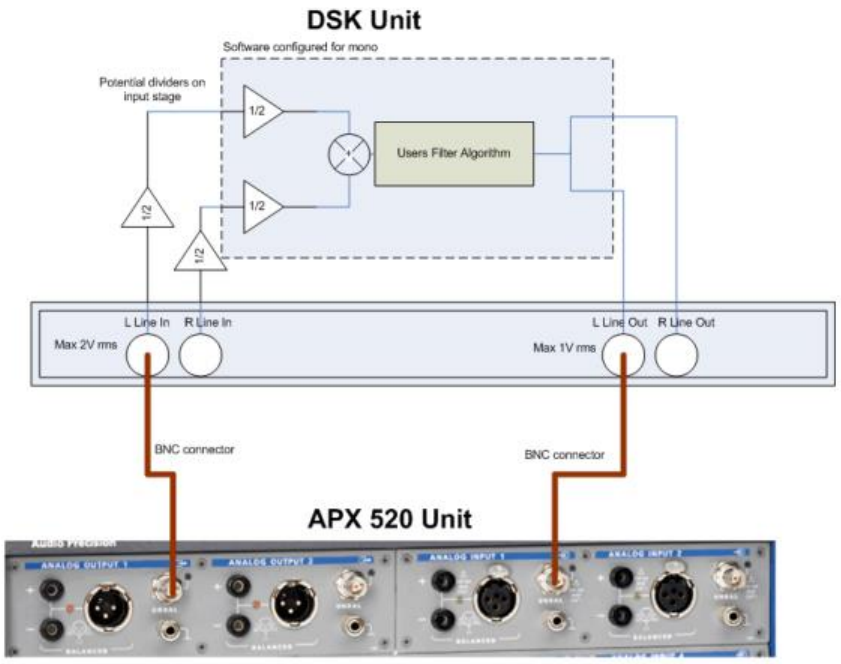


Figure 30 - APX520 connections showing potential dividers

Whilst this is not within the specification given, methods in which to alter the design of the filter were described in the ‘FIR design in Matlab’ section. One way in which to accomplish this would be to reduce the stopband attenuation to a number far less than -48dB, for example -100dB, and make alterations on the stopband and passband frequencies. The passband ripple cannot be altered however since the maximum of 0.4dB is already being used. In doing this, the number of coefficients greatly increases and at a certain point when there are too many coefficients the frequency response from the network analyser is worse. This is because the complier takes too long to calculate all the values associated with the coefficients therefore distorting the output to the network analyser. It is for this reason the decision was made to maintain the same filter coefficients throughout the experiment.

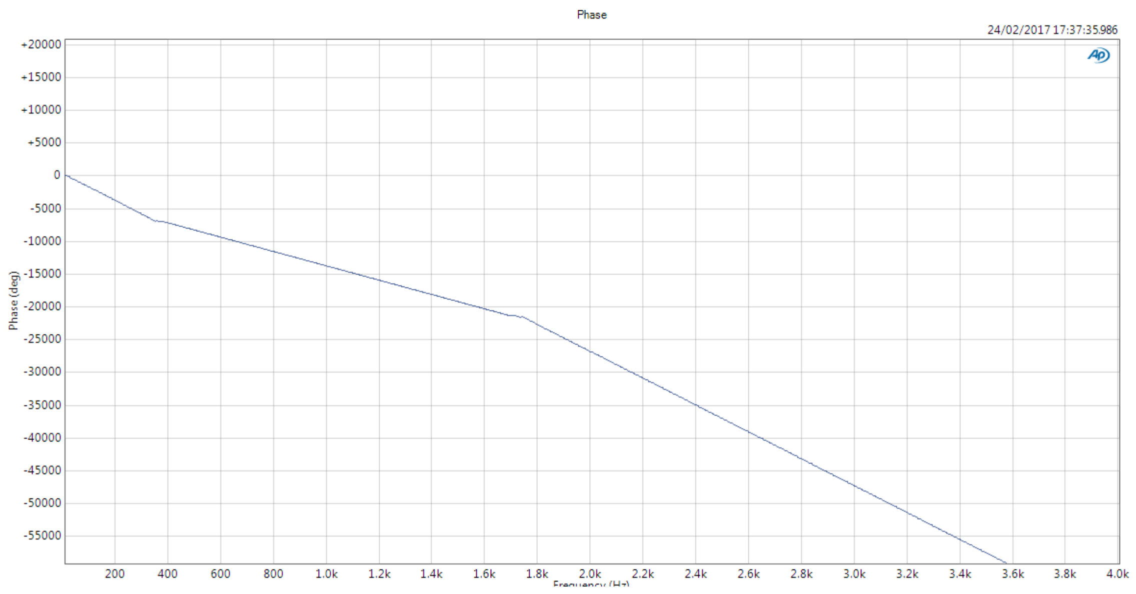


Figure 31 - Phase plot from network analyser

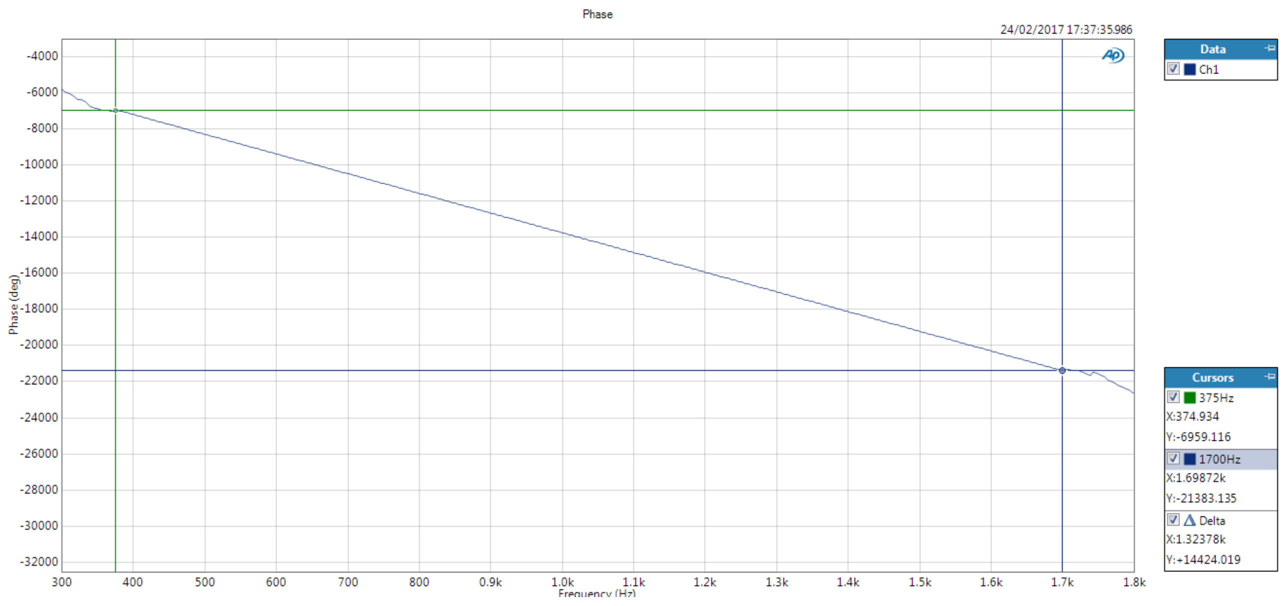


Figure 32 - Close up of phase plot from network analyser

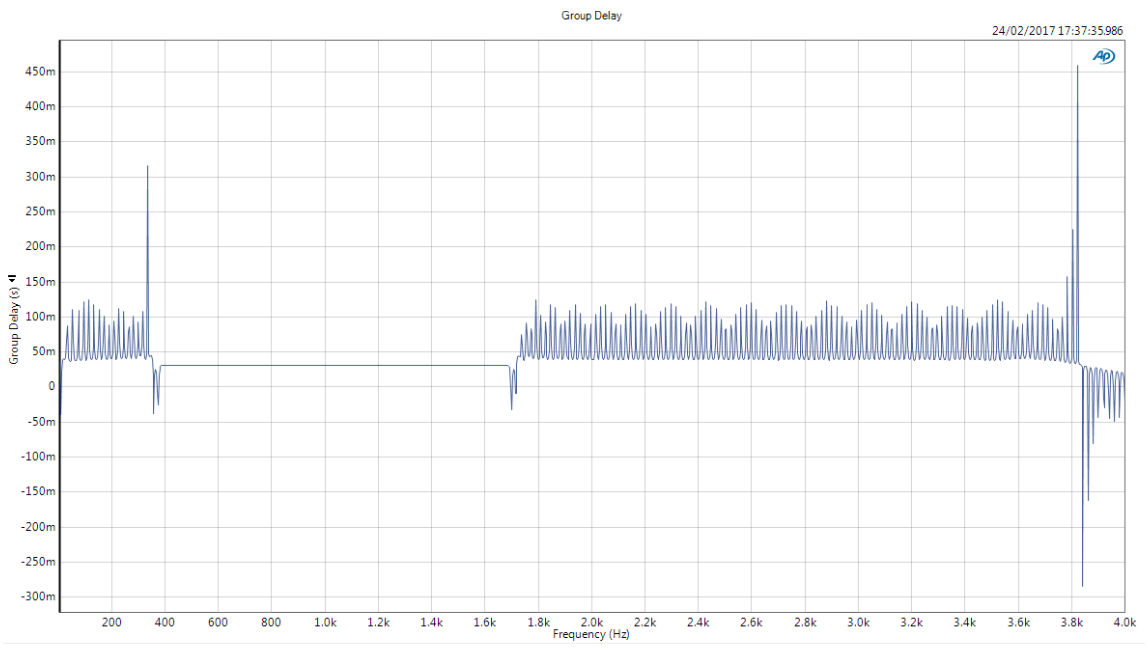


Figure 33 - Group delay showing linear phase

Figures 31, 32 and 33 all show the phase of the FIR filter. From these diagrams, it can be seen that the phase is indeed linear, which it what is expected. Figure 32 in particular shows the linearity of the filter

Conclusion

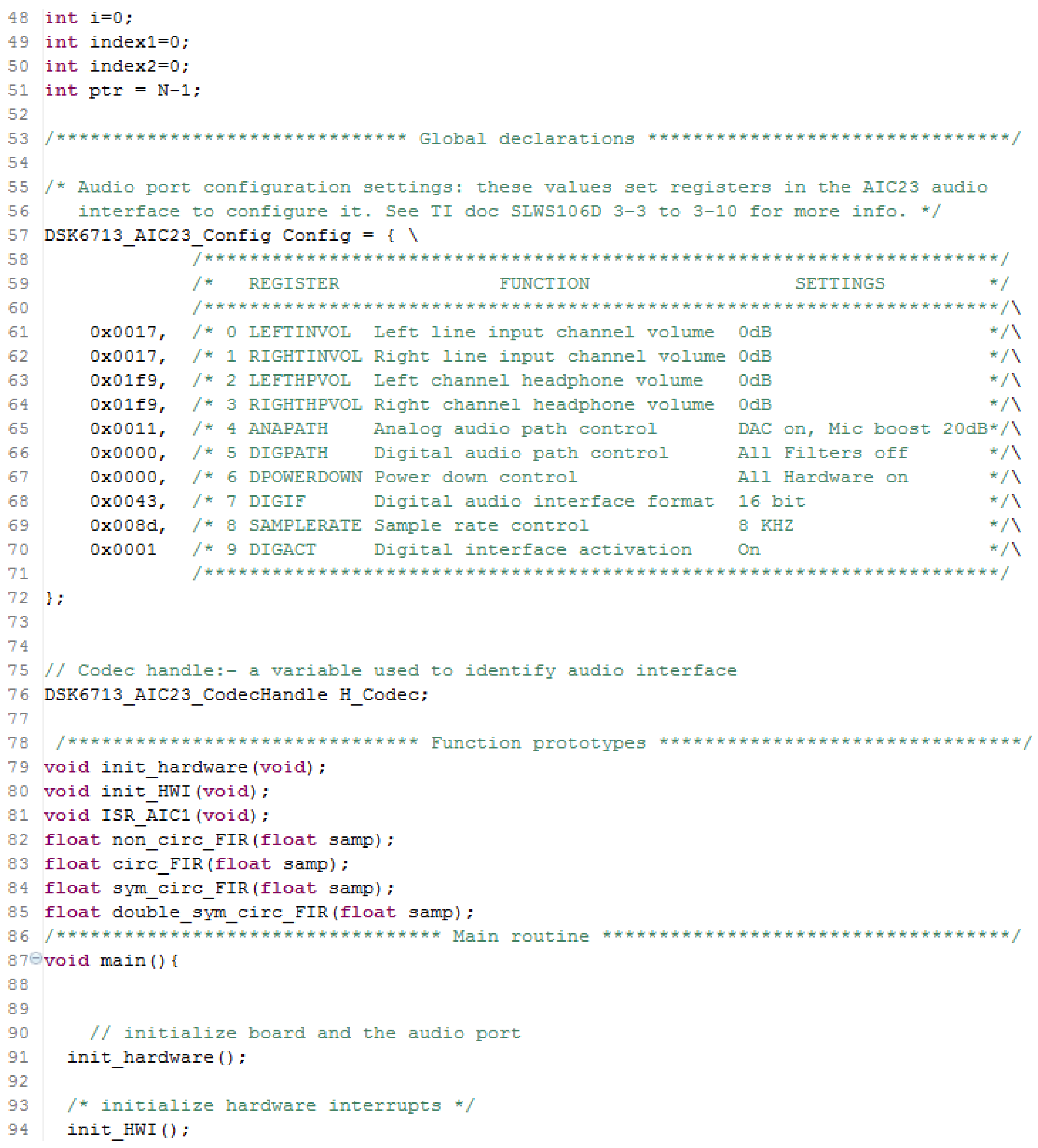
In conclusion, FIR filters can be designed in Matlab, with the use of some helpful functions and then implemented with varying amounts of efficiency in C. Whilst the exact specification was not met from the frequency analyser, the general behaviour of the filter was close to the one designed in Matlab.

During this report, multiple optimisation algorithms to improve the speed of the FIR filter were discussed. It was found that the method by which to maximize the optimisation was to implement a double sized circular buffer and to exploit the symmetrical property of linear-phase FIR filters. Other optimizations tools used were adapting the `type` from `double` to `float` and inlining the code in the ISR. Even though it wasn’t implemented in this report, an assembly implementation of the FIR filter would result in less clock cycles.

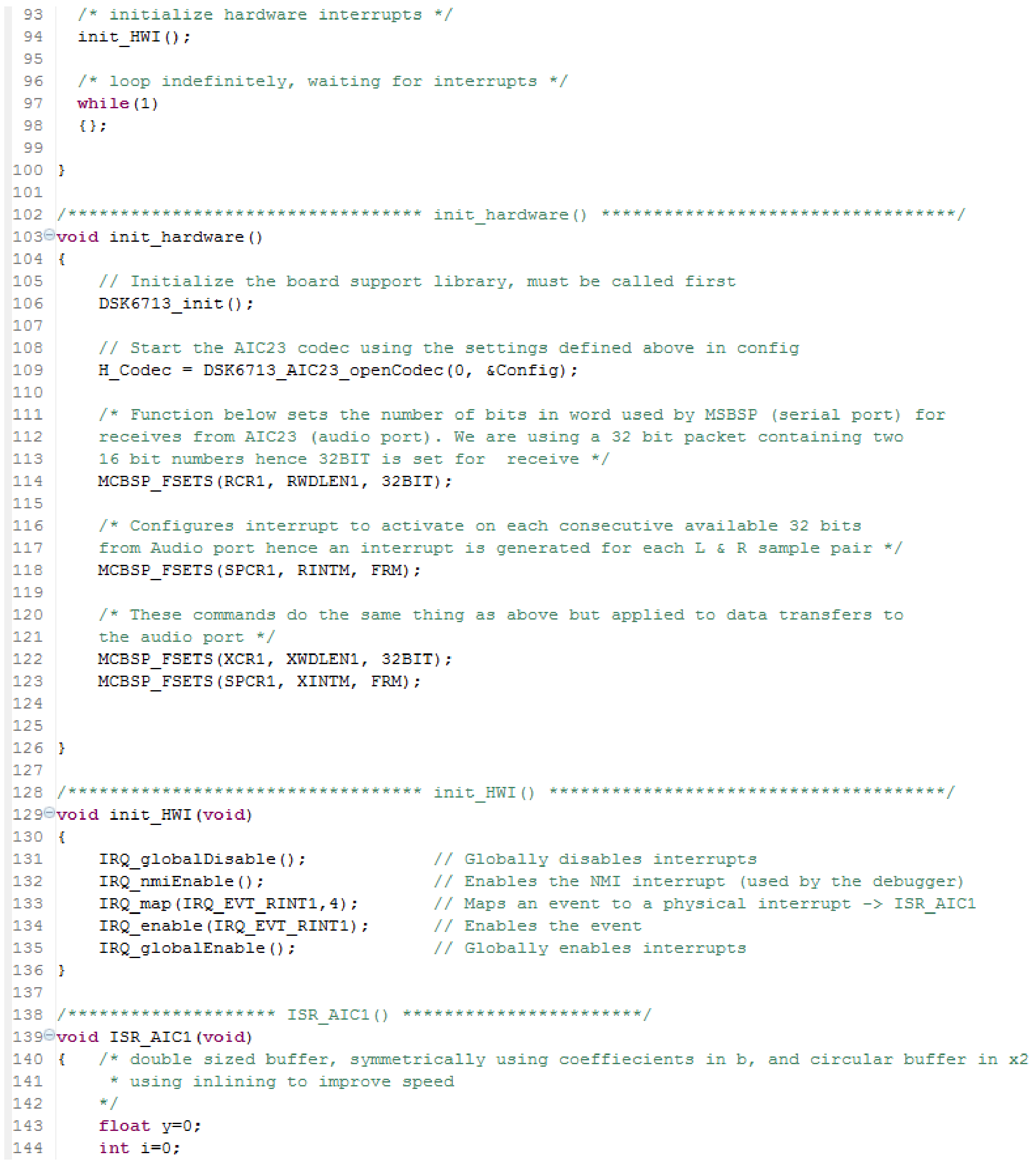
Appendix

Full readable code

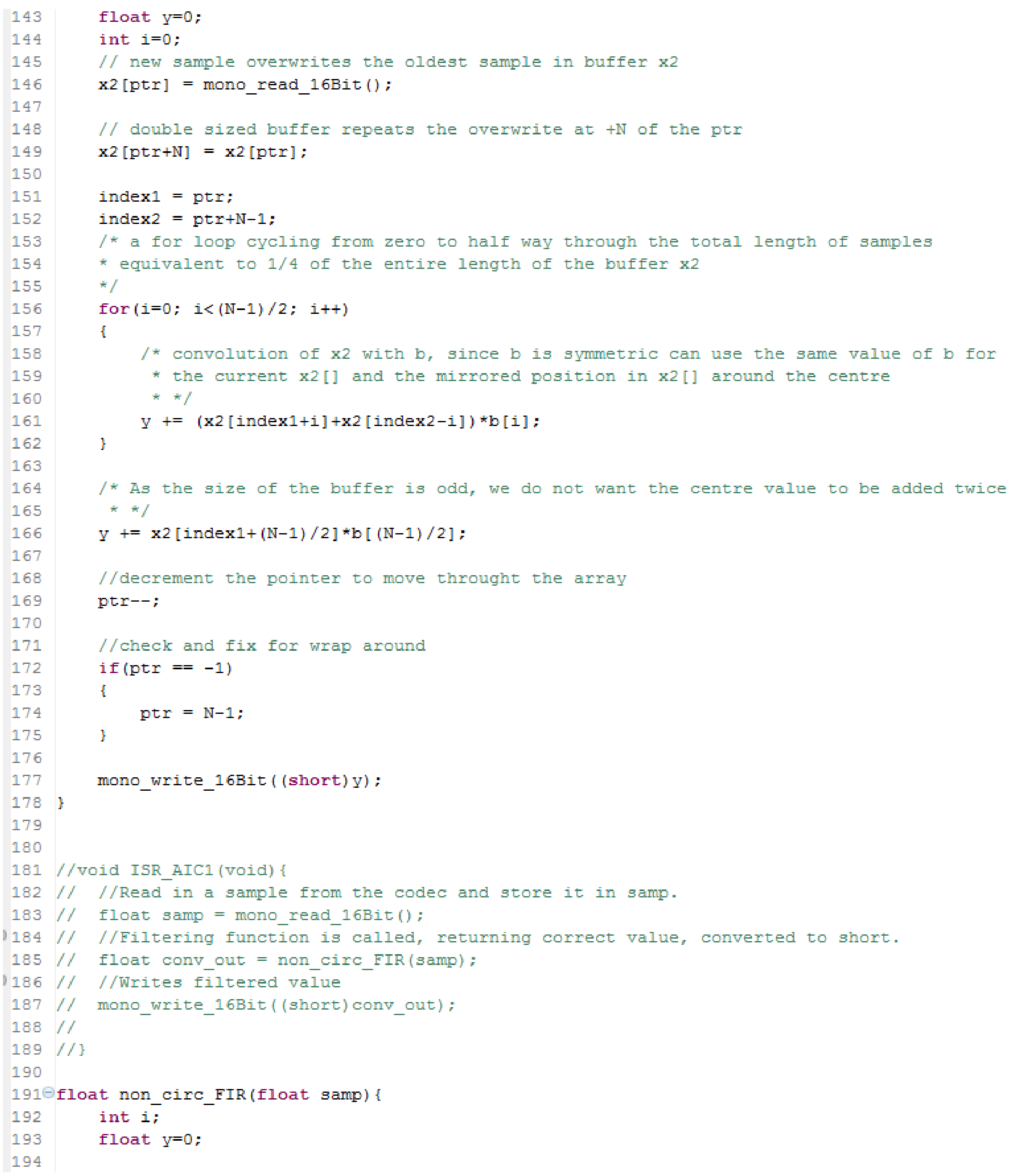
```
/*************************************************************************************
                   DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                                 IMPERIAL COLLEGE LONDON

                      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>
//  Included so program can make use of DSP/BIOS configuration tool.
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This
   example also includes dsk6713_aic23.h because it uses the
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)


// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>

#include "fir_coeff.txt"

#define N 429

float x[N] = {0};
float x2[N*2] = {0};
float conv_out = 0;

int i=0;
int index1=0;
int index2=0;
int ptr = N-1;
```



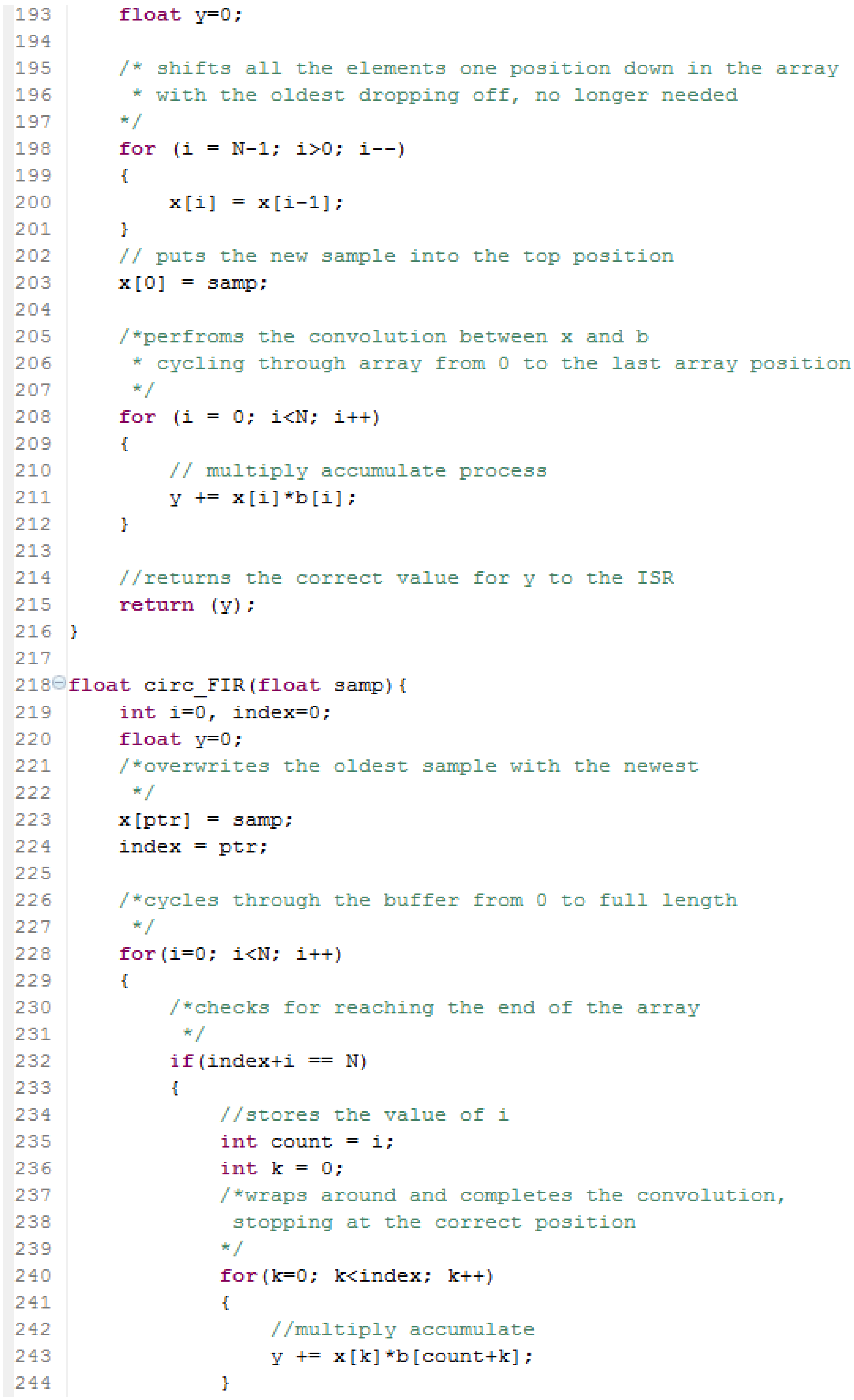
```c
int i=0;
int index1=0;
int index2=0;
int ptr = N-1;

/******************************** Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
             /*********************************************************************/
             /*   REGISTER              FUNCTION                  SETTINGS        */
             /*********************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                 */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                 */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                 */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                 */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off     */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on     */\
    0x0043,  /* 7 DIGIF      Digital audio interface format  16 bit              */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ               */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                  */\
             /*********************************************************************/
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

 /******************************** Function prototypes *********************************/
void init_hardware(void);
void init_HWI(void);
void ISR_AIC1(void);
float non_circ_FIR(float samp);
float circ_FIR(float samp);
float sym_circ_FIR(float samp);
float double_sym_circ_FIR(float samp);
/*********************************** Main routine ************************************/
void main(){


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

  /* initialize hardware interrupts */
  init_HWI();
```



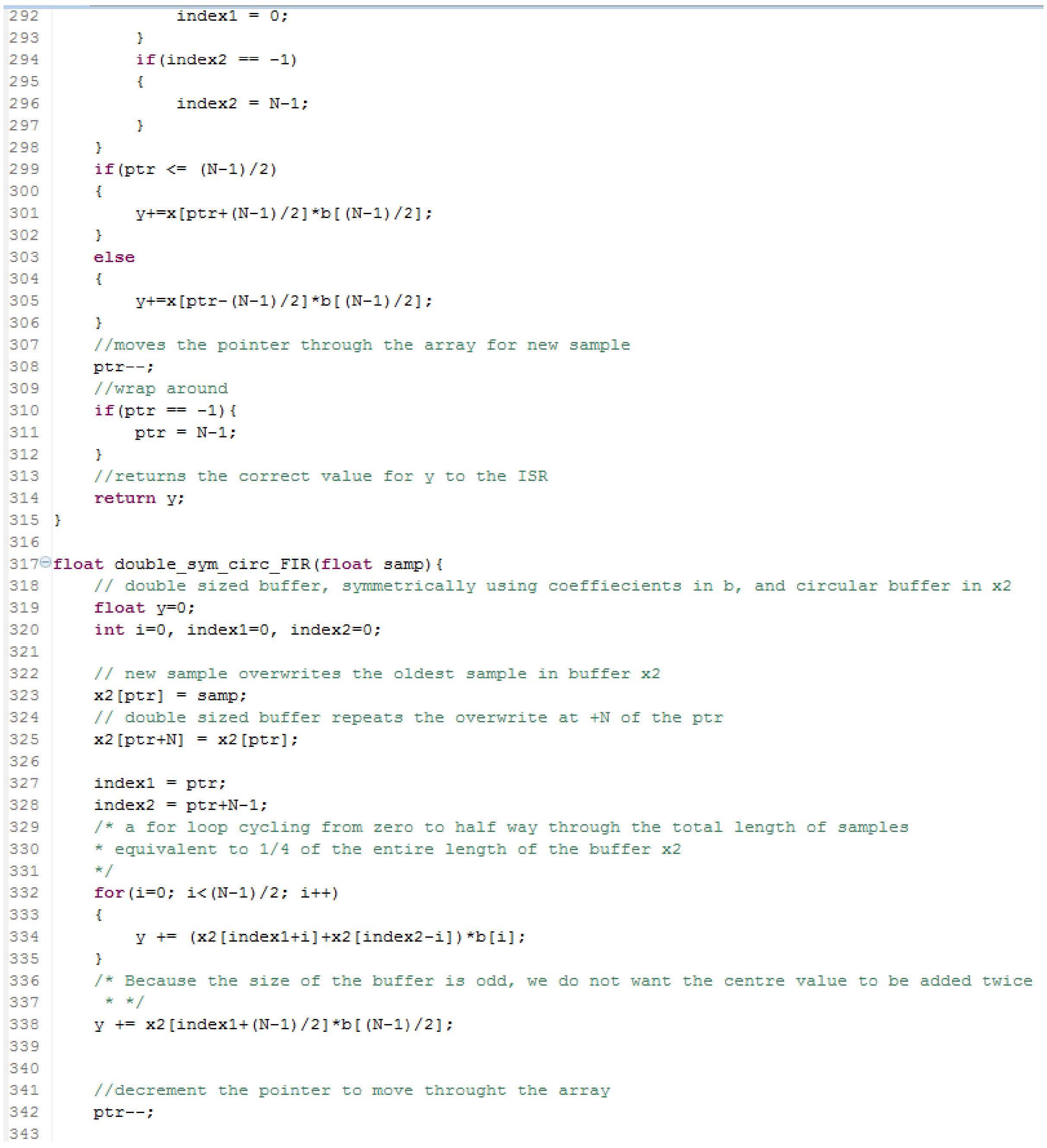
```c
  /* initialize hardware interrupts */
  init_HWI();

  /* 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 -> ISR_AIC1
    IRQ_enable(IRQ_EVT_RINT1);      // Enables the event
    IRQ_globalEnable();             // Globally enables interrupts
}

/******************** ISR_AIC1() **********************/
void ISR_AIC1(void)
{   /* double sized buffer, symmetrically using coeffieciencts in b, and circular buffer in x2
     * using inlining to improve speed
    */
    float y=0;
    int i=0;
```



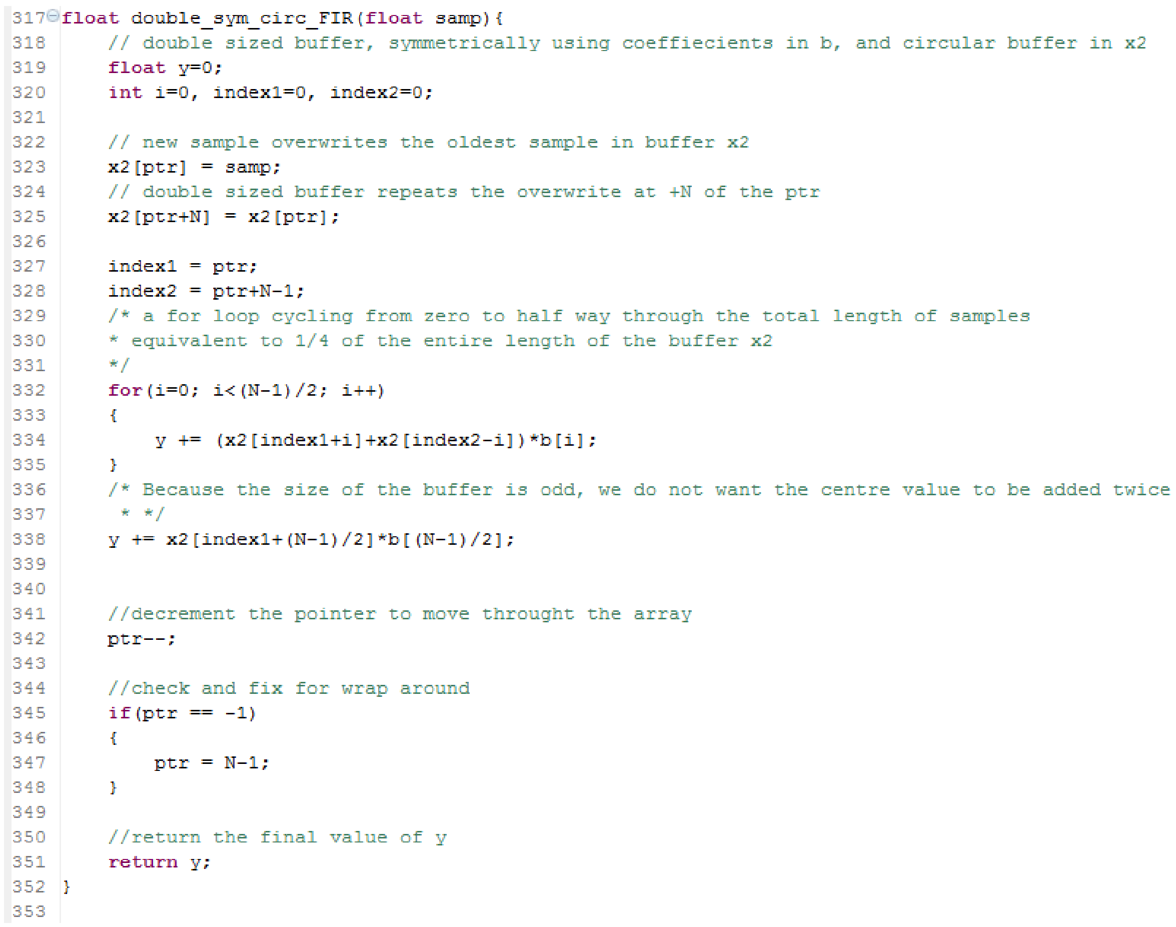
```c
    float y=0;
    int i=0;
    // new sample overwrites the oldest sample in buffer x2
    x2[ptr] = mono_read_16Bit();

    // double sized buffer repeats the overwrite at +N of the ptr
    x2[ptr+N] = x2[ptr];

    index1 = ptr;
    index2 = ptr+N-1;
    /* a for loop cycling from zero to half way through the total length of samples
    * equivalent to 1/4 of the entire length of the buffer x2
    */
    for(i=0; i<(N-1)/2; i++)
    {
        /* convolution of x2 with b, since b is symmetric can use the same value of b for
         * the current x2[] and the mirrored position in x2[] around the centre
         * */
        y += (x2[index1+i]+x2[index2-i])*b[i];
    }

    /* As the size of the buffer is odd, we do not want the centre value to be added twice
     * */
    y += x2[index1+(N-1)/2]*b[(N-1)/2];

    //decrement the pointer to move throught the array
    ptr--;

    //check and fix for wrap around
    if(ptr == -1)
    {
        ptr = N-1;
    }

    mono_write_16Bit((short)y);
}


//void ISR_AIC1(void){
//  //Read in a sample from the codec and store it in samp.
//  float samp = mono_read_16Bit();
//  //Filtering function is called, returning correct value, converted to short.
//  float conv_out = non_circ_FIR(samp);
//  //Writes filtered value
//  mono_write_16Bit((short)conv_out);
//
//}

float non_circ_FIR(float samp){
    int i;
    float y=0;

```



```c
    float y=0;

    /* shifts all the elements one position down in the array
     * with the oldest dropping off, no longer needed
    */
    for (i = N-1; i>0; i--)
    {
        x[i] = x[i-1];
    }
    // puts the new sample into the top position
    x[0] = samp;

    /*perfroms the convolution between x and b
     * cycling through array from 0 to the last array position
     */
    for (i = 0; i<N; i++)
    {
        // multiply accumulate process
        y += x[i]*b[i];
    }

    //returns the correct value for y to the ISR
    return (y);
}

float circ_FIR(float samp){
    int i=0, index=0;
    float y=0;
    /*overwrites the oldest sample with the newest
     */
    x[ptr] = samp;
    index = ptr;

    /*cycles through the buffer from 0 to full length
     */
    for(i=0; i<N; i++)
    {
        /*checks for reaching the end of the array
         */
        if(index+i == N)
        {
            //stores the value of i
            int count = i;
            int k = 0;
            /*wraps around and completes the convolution,
             stopping at the correct position
            */
            for(k=0; k<index; k++)
            {
                //multiply accumulate
                y += x[k]*b[count+k];
            }
            }
            // causes exit from for loop asconvolution is complete
            i = N;
        }
        else{
        y += x[index+i]*b[i];
        }
    }
    //moves the pointer through the array for new sample
    ptr--;

    //wrap around
    if(ptr == -1){
        ptr = N-1;
    }

    //returns the correct value for y to the ISR
    return y;
}

float sym_circ_FIR(float samp){
    // use two pointers to select the values to be multiplied by the b value
    int i=0, index1=0, index2=0;
    float y=0;
    //new sample overwrites the oldest sample in buffer x
    x[ptr] = samp;
    index1 = ptr;
    index2 = ptr-1;
    /*if pointer 1 is at position 0, the corresponding starting position for
     *the second pointer is the last value in the array*/
    if(index1==0)
    {
        index2 = N-1;
    }
    /*a for loop cycling from zero to half way through the total length of samples, since we can
     * multiply two values of x by the value of b in each loop
     */
    for(i=0; i<(N-1)/2; i++)
    {
        /*multiply accumulate of x with b
         */
        y += x[index1]*b[i] + x[index2]*b[i];
        //increments/decrements index values
        index1++;
        index2--;
        // checks to see if the index's have reached the end of the buffer, and wraps around
        if(index1 == N)
        {
            index1 = 0;
        }
```



```c
            index1 = 0;
        }
        if(index2 == -1)
        {
            index2 = N-1;
        }
    }
    if(ptr <= (N-1)/2)
    {
        y+=x[ptr+(N-1)/2]*b[(N-1)/2];
    }
    else
    {
        y+=x[ptr-(N-1)/2]*b[(N-1)/2];
    }
    //moves the pointer through the array for new sample
    ptr--;
    //wrap around
    if(ptr == -1){
        ptr = N-1;
    }
    //returns the correct value for y to the ISR
    return y;
}

float double_sym_circ_FIR(float samp){
    // double sized buffer, symmetrically using coeffiecients in b, and circular buffer in x2
    float y=0;
    int i=0, index1=0, index2=0;

    // new sample overwrites the oldest sample in buffer x2
    x2[ptr] = samp;
    // double sized buffer repeats the overwrite at +N of the ptr
    x2[ptr+N] = x2[ptr];

    index1 = ptr;
    index2 = ptr+N-1;
    /* a for loop cycling from zero to half way through the total length of samples
    * equivalent to 1/4 of the entire length of the buffer x2
    */
    for(i=0; i<(N-1)/2; i++)
    {
        y += (x2[index1+i]+x2[index2-i])*b[i];
    }
    /* Because the size of the buffer is odd, we do not want the centre value to be added twice
     * */
    y += x2[index1+(N-1)/2]*b[(N-1)/2];


    //decrement the pointer to move throught the array
    ptr--;

```



```c
float double_sym_circ_FIR(float samp){
    // double sized buffer, symmetrically using coeffiecients in b, and circular buffer in x2
    float y=0;
    int i=0, index1=0, index2=0;

    // new sample overwrites the oldest sample in buffer x2
    x2[ptr] = samp;
    // double sized buffer repeats the overwrite at +N of the ptr
    x2[ptr+N] = x2[ptr];

    index1 = ptr;
    index2 = ptr+N-1;
    /* a for loop cycling from zero to half way through the total length of samples
    * equivalent to 1/4 of the entire length of the buffer x2
    */
    for(i=0; i<(N-1)/2; i++)
    {
        y += (x2[index1+i]+x2[index2-i])*b[i];
    }
    /* Because the size of the buffer is odd, we do not want the centre value to be added twice
     * */
    y += x2[index1+(N-1)/2]*b[(N-1)/2];


    //decrement the pointer to move throught the array
    ptr--;

    //check and fix for wrap around
    if(ptr == -1)
    {
        ptr = N-1;
    }

    //return the final value of y
    return y;
}
```

References

1. (No Date) Available at: https://bb.imperial.ac.uk/bbcswebdav/pid-885309-dt-content-rid-3073226_1/courses/DSS-EE3_19-16_17/DSS-EE3_19-16_17_ImportedContent_20160826101620/lab3.pdf (Accessed: 08 February 2017).

2. Inline expansion (2016) in Wikipedia. Available at: https://en.wikipedia.org/wiki/Inline_expansion (Accessed: 20 February 2017).

3. butter (1994) Firpm. Available at: http://uk.mathworks.com/help/signal/ref/firpm.html?s_tid=gn_loc_drop (Accessed: 19 February 2017).

4. buttord (1994) Firpmord. Available at: https://uk.mathworks.com/help/signal/ref/firpmord.html (Accessed: 19 February 2017).

5. Freqz (1994) Available at: https://uk.mathworks.com/help/signal/ref/freqz.html (Accessed: 19 February 2017).

6. Circular buffer (2016) in Wikipedia. Available at: https://en.wikipedia.org/wiki/Circular_buffer (Accessed: 22 February 2017).

7. Double or float, which is faster? (2017) Available at: http://stackoverflow.com/questions/4584637/double-or-float-which-is-faster (Accessed: 23 February 2017).