

IMPERIAL COLLEGE LONDON

REAL TIME DIGITAL SIGNAL PROCESSING

LAB 4 REPORT

Andrew Zhou, CID: 00938859

Jagannaath Shiva Letchumanan, CID: 00946740

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 In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others.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

Signed: **Andrew Zhou, Jagannaath Shiva Letchumanan**

February 24, 2017

# Contents

<b>1</b>	<b>Background and Introduction</b>	<b>2</b>
<b>2</b>	<b>MATLAB Filter Design</b>	<b>6</b>
<b>3</b>	<b>Optimization Levels</b>	<b>10</b>
<b>4</b>	<b>Non-circular FIR</b>	<b>12</b>
4.1	Theory and code . . . . .	12
4.2	Output signal . . . . .	13
<b>5</b>	<b>Non-circular FIR, symmetric coefficients</b>	<b>16</b>
<b>6</b>	<b>Circular FIR</b>	<b>17</b>
<b>7</b>	<b>Circular FIR, symmetric coefficients</b>	<b>19</b>
7.1	Handling out of bounds with if statements . . . . .	19
7.2	Wrap prediction - no if statements . . . . .	20
<b>8</b>	<b>Circular FIR, Double Buffer</b>	<b>22</b>
<b>9</b>	<b>Further Improvements</b>	<b>23</b>
<b>10</b>	<b>Spectrum Analysis</b>	<b>24</b>
10.1	Improved filter designs . . . . .	26
	<b>Appendices</b>	<b>27</b>
<b>A</b>	<b>MATLAB code for obtaining FIR coefficients</b>	<b>27</b>
<b>B</b>	<b>Non-circular FIR</b>	<b>29</b>
<b>C</b>	<b>Non-circular FIR, symmetric coefficients</b>	<b>33</b>
<b>D</b>	<b>Circular FIR</b>	<b>37</b>
<b>E</b>	<b>Circular FIR, symmetric coefficients</b>	<b>41</b>
<b>F</b>	<b>Circular FIR, symmetric coefficients with wrap prediction and float precision optimisation</b>	<b>46</b>
<b>G</b>	<b>Circular FIR, double buffer with float precision optimisation and inlining</b>	<b>51</b>

# 1 Background and Introduction

**Interrupt Driven Programming:** In Lab 3, interrupt driven programming was introduced to improve upon the polling code used previously. Both of these techniques are used to allow a computer to interact with external devices. Polling checks if the external device requires servicing synchronously within the code whilst interrupts do so asynchronously and send Interrupt Requests (IRQs) to the processor whereby it vectors to an Interrupt Service Routine (ISR). Hence, interrupts save time in cases where the external device does not need to be serviced, as polling will check anyway, and in cases where the external device requires servicing well before polling is due to check for required servicing.

On the other hand, interrupts required extra hardware to save the context of the `main()` function, and reload the context once the interrupt has been serviced.

**Finite Impulse Response Filters:** A finite impulse response (FIR) filter is a filter whose impulse response is zero outside a finite duration. In a way, the impulse response of an FIR filter performs a weighted average of the past few inputs and is hence, referred to as a Moving Average (MA) filter. As analogue filters (with capacitors and inductors) have an infinite memory, FIR filters do not have an analogue equivalent and can hence, only be implemented digitally.

The output of an FIR filter can be represented in many different ways. Firstly as a difference equation:

$$\begin{aligned} y[n] &= b_0x[n] + b_1x[n-1] + \dots b_Nx[n-N] \\ &= \sum_{i=0}^N b_ix[n-i] \end{aligned} \tag{1}$$

Where  $x[n]$  is the input signal,  $y[n]$  is the output, and  $b_i$  are the filter coefficients, equal to the value of the impulse response at the  $i$ 'th instant.  $N$  represents the order of the filter, and also the memory of the system as it is the maximum power of  $z^{-1}$ . The order is finite, differing again from the case of an IIR filter where the memory is infinite.

Physically, as stated earlier, this equation represents a weighted sum of the present and past  $N$  input values, and the computation is a discrete convolution between the filter coefficients and the input values. Each coefficient considered in the calculation,  $b_i$ , is referred to as a tap (based on the structure of a tapped delay line), and as such, a FIR filter of order  $N$  will have  $N + 1$  taps.

The z-transform of (1) and signal flow graph (Figure 1) for the direct form of the filter are shown below:

$$\begin{aligned}
H(z) &= \frac{Y(z)}{X(z)} = b[0] + b[1]z^{-1} + \dots + b[N]z^{-N} = \sum_{i=0}^N b_i z^{-i} \\
&= \frac{b[0]z^N + b[1]z^{N-1} + \dots + b[N]}{z^N}
\end{aligned} \tag{2}$$

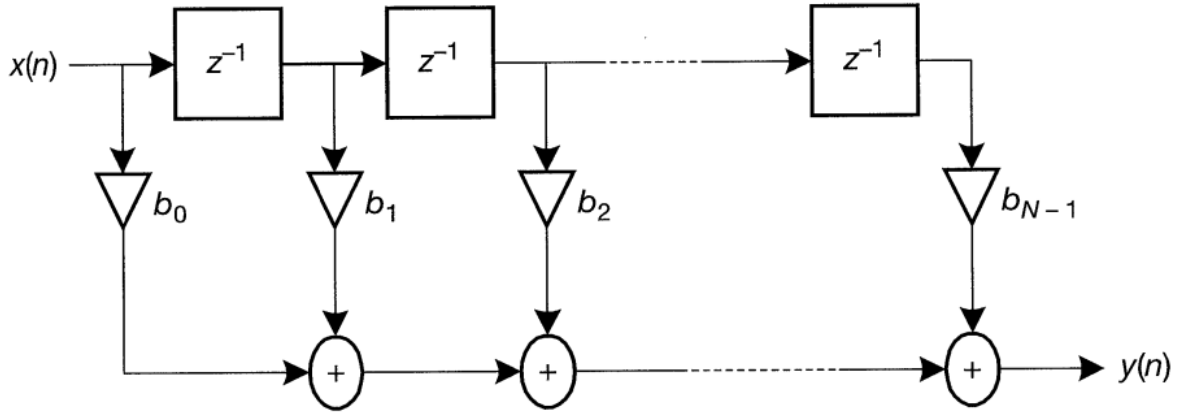


Figure 1: FIR direct form diagram

Thus,  $H(z)$  is a polynomial in  $z$  and it can have zeros anywhere (dependent on the coefficients), but has  $N$  poles, all at the origin ( $z=0$ ). This is again another distinguishing feature between FIR and IIR filters, as IIR filters can have poles anywhere on the  $z$ -plane depending on the coefficients.

The locations of poles and zeros for filters dictate the behaviour of the filter in terms of transfer function and stability. Poles outside the unit circle in the  $z$ -plane will result in an unstable system where the impulse response will grow exponentially to infinity, whereas, a pole placed close to the unit circle will allow sharp attenuation which is often a desirable feature. On the other hand, zeros do not cause instability and zeros on the unit circle cause the frequency response to decay around that frequency.

FIR filters, unlike IIR filters, have no freedom in the position of their poles; they always have  $M$  poles at the origin as evident from (2). The consequence of this is that for accurate frequency selectivity they tend to be of a higher order and thus require considerably more computational power compared to IIR filters for the same sharpness and selectivity.

The positioning of the poles does however mean that FIR filters are inherently stable. In addition to this, the lack of feedback from output to input means that any rounding errors, while compounded by the summed iterations, will not become infinite (as the number of

coefficients is finite). The same relative errors occur in each calculation and hence, as long as the number of coefficients is sufficiently large, the errors can be ignored. This also makes implementation simpler. This fact plays a crucial role in code optimisation in later parts of the laboratory session.

FIR filters can also be made to be linear phase in relevant frequency ranges. The implication of this is that the filter coefficients will be symmetrical i.e.  $b[i] = b[N - i]$ , where  $N$  is the order of the filter. This property can be exploited, allowing efficient implementations of the filter computations where the number of multiplication is reduced from  $N$  to either  $(N + 1)/2$  (even number of taps  $N + 1$ ) or  $(N + 2)/2$  (odd number of taps  $N + 1$ ).

$$\begin{aligned}
H(z) &= \sum_{n=0}^N h(n)z^{-n} \\
&= \sum_{n=0}^{((N+1)/2)-1} h(n)[z^{-n} + z^{-(N-n)}], N+1 \text{ even} \\
&= \sum_{n=0}^{[(N)/2]-1} h(n)[z^{-n} + z^{-(N-n)}] + h(N/2)z^{-[N/2]}, N+1 \text{ odd}
\end{aligned} \tag{3}$$

and hence

$$\begin{aligned}
H(e^{j\omega}) &= e^{-j\omega[N/2]} \left\{ \sum_{n=0}^{((N+1)/2)-1} 2h(n)\cos\left[\omega\left(n - \frac{N}{2}\right)\right] \right\}, N+1 \text{ even} \\
&= e^{-j\omega[N/2]} \left\{ h\left(\frac{N}{2}\right) + \sum_{n=0}^{(N-2)/2} 2h(n)\cos\left[\omega\left(n - \frac{N}{2}\right)\right] \right\}, N+1 \text{ odd}
\end{aligned} \tag{4}$$

Considering only the case of an odd number of filter coefficients, the magnitude and phase responses are:

$$\begin{aligned}
|H(e^{j\omega})| &= h\left(\frac{N}{2}\right) + \sum_{n=0}^{(N-2)/2} 2h(n)\cos\left[\omega\left(n - \frac{N}{2}\right)\right] \\
\angle H(e^{j\omega}) &= -\omega\frac{N}{2}
\end{aligned}$$

Linear phase FIR filters have another interesting property in which the group delay can be expressed as:

$$\text{Group Delay} = -\frac{d\phi}{d\omega} = \frac{N}{2} \text{ samples}$$

In addition to this, the zeros of a linear phase FIR appear in mirror image pairs across the unit circle i.e. if  $z_0$  is a zero, then  $\frac{1}{z_0^*}$  is also a zero.

Taking all of this into account, the calculations can be simplified as can be seen in Figure 2 below.

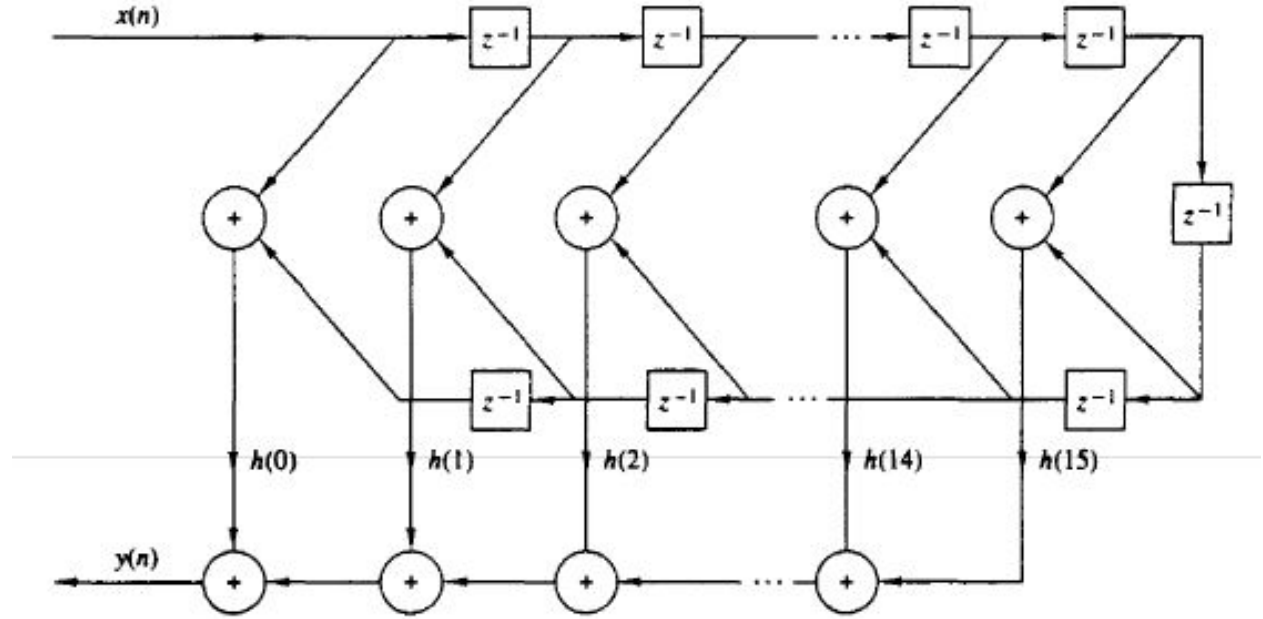


Figure 2: Linear Phase FIR signal flow graph [1]

This was used to optimise the filter's performance as can be seen in section 5 later.

## 2 MATLAB Filter Design

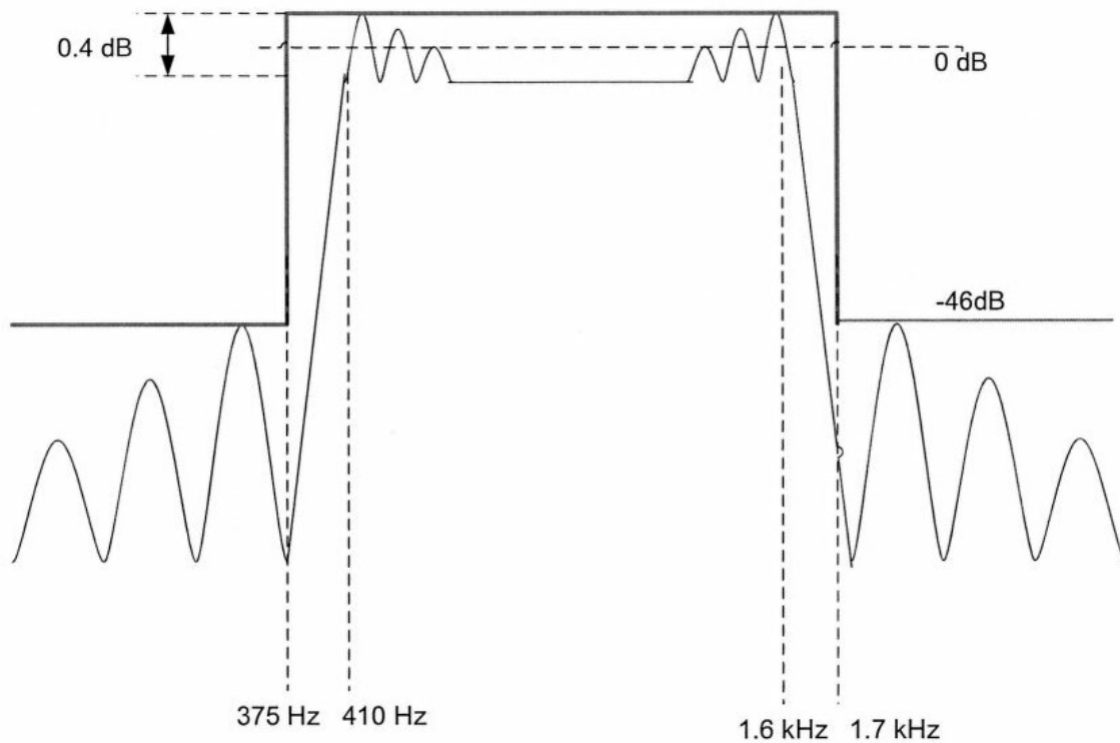


Figure 3: Filter Specifications

Figure 3 details the specifications required for the FIR filter to be achieved in this laboratory session. The low frequency stop band ranges from 0Hz to 375Hz, across which there is a minimum attenuation of -46dB, or 0.005. The low frequency transition band spans from 375Hz to 410Hz, after which the passband starts. The maximum allowable ripple in this range is 0.4dB, or 0.023. The passband stops at 1.6kHz, from which another transition band occurs, and finishes at 1.7kHz. The high frequency stop band has the same gain restrictions as the low frequency one.

To find the filter order and coefficients, the `firpm` and `firpmord` functions in MATLAB were used.

The function `firpmord` takes the following inputs: a vector  $f$ , of the frequency band edges; a vector  $a$  of the frequency band amplitudes; and a vector  $dev$  that specifies the maximum allowable deviation from the desired response in each band (not in decibels). The sampling frequency  $f_s$  can also be specified as a fourth input. For a filter with three frequency bands,  $f$  will have 4 elements, and  $a$  and  $dev$  will each have 3 elements.

**Note:** Passband ripple is not equal to deviation. Let the ripple (dB) be  $rp$

$$\text{Pass band ripple (not in dB)} = \frac{10^{rp/20} - 1}{10^{rp/20} + 1}$$

$$\text{Deviation in stop - bands} = 10^{-\text{attenuation}/20}$$

This gives a deviation vector,  $\text{dev} = [0.005 \ 0.023 \ 0.005]$ . The function will then output: the approximate order  $N$ ; normalised frequency band edges,  $F_o$ ; frequency band amplitudes  $A_o$ ; and weights  $w$  that meet the specifications  $f_o$ ,  $a_o$ , and  $\text{dev}$ .

The MATLAB code to meet the specific requirements is therefore:

`[N,Fo,Ao,W]=firpmord([375,410,1600,1700],[0,1,0],[0.005,0.023,0.005],8000)` (Note that target filter has features specified in decibels, whereas MATLAB input is specified in base 10 ratios)

The `firpm` function then takes in input these outputs and calculates a row vector  $b$  containing  $N + 1$  coefficients of the order  $N$  FIR filter as specified by the inputs to `firpmord`. Thus the code is written as:

`b=firpm(N,Fo,Ao,W)`

The above coefficients are calculated by the Parks-McClellan algorithm, which is an iterative algorithm that finds the optimal coefficients for a linear-phase Chebyshev filter.

This resulted in the filter below:

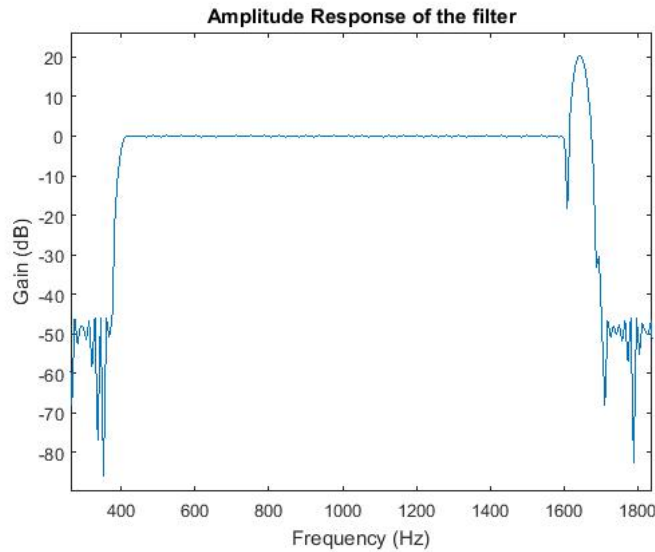


Figure 4: Amplitude response of filter with non-symmetric transition bands



As observed, there is a large, undesirable spike in gain at the high frequency side of the passband. This is caused by the fact that the filter being designed is a linear phase FIR filter which needs to have a symmetric magnitude response (refer to equation 4 above) and hence, the uneven widths of the transition bands cause this anomaly. By changing the frequency range of the high frequency transition band to 35Hz as well, the spike in gain will be eliminated. Thus, the frequency range for the second transition band was altered to 1620Hz to 1655Hz. The second transition band was not started at 1600Hz in case the filter implemented in hardware started attenuating the signals slightly before the cut-off frequency. The resulting filter has an amplitude response with a flat passband as can be seen in Figure 5.

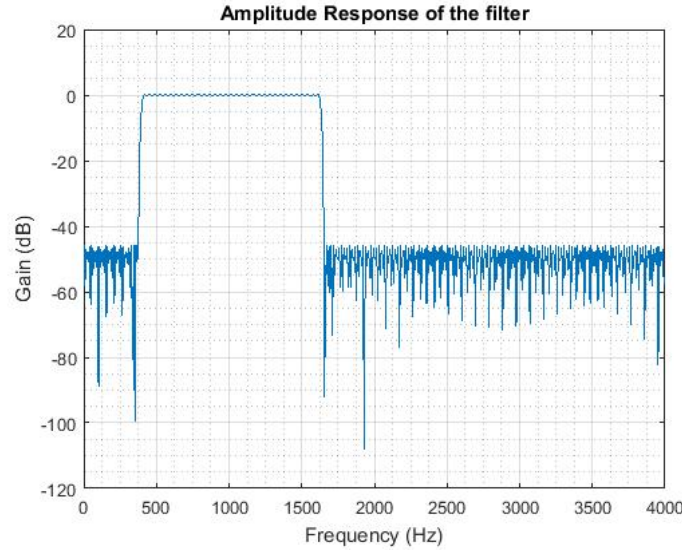


Figure 5: Amplitude response of filter with symmetric transition bands

To confirm that the filter met the specifications in stop band attenuation and ripple, the amplitude response was observed in greater detail:

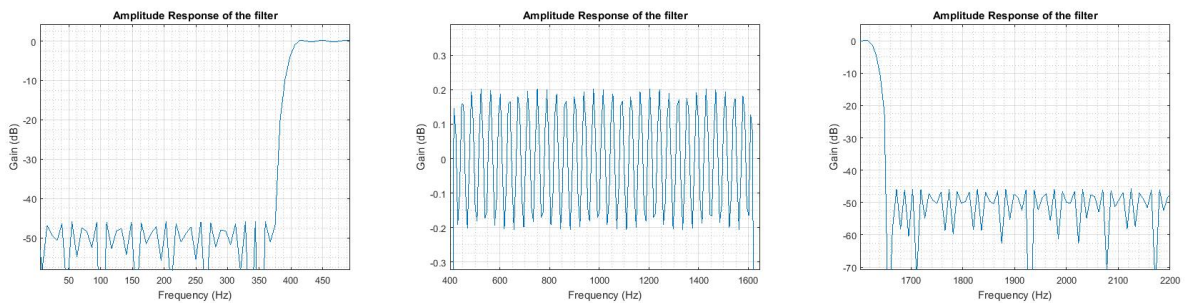


Figure 6: (Left to right) Plots of attenuation in low frequency stop band, ripple in passband, and attenuation in high frequency stop band

To affirm the linear phase in the passband, and thus symmetric coefficients properties of the filter, the phase response was also observed:

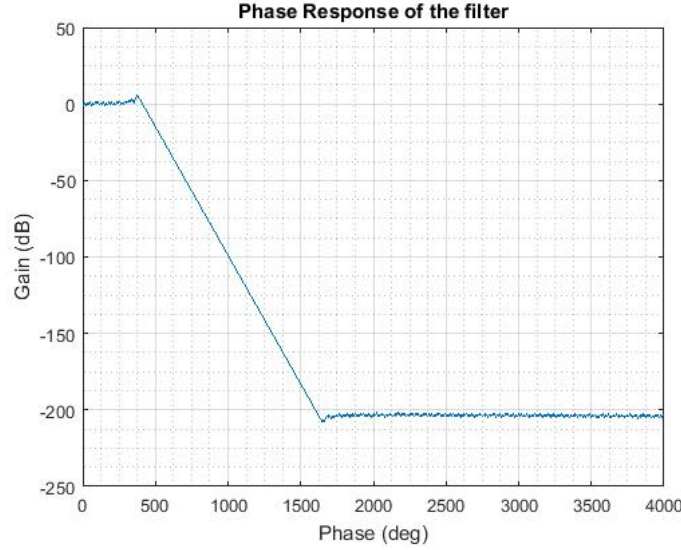


Figure 7: Phase response of filter with symmetric transition bands

Taking the phase in radians, the group delay (slope) was found to 214 sample (26.75ms). Ultimately, a filter of order  $N=428$  with 429 coefficients was determined. These coefficients were saved to a .txt file named `fir_coefs.txt` through the use of a series of `fprintf` functions. This was necessary due to the way the coefficients are read into Code Composer. The full code, including code for plotting the figures seen in this section, can be found in the appendix.

The impulse response of the filter can be seen in Figure 8 below:

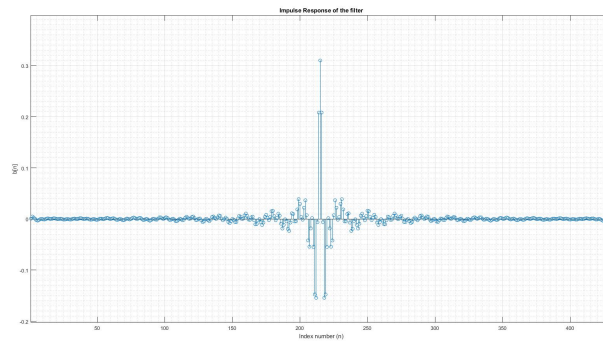


Figure 8: Impulse Response of the filter

As the filter is bandpass, the zero-pole plot will have a lot of zeros and poles on the unit circle except for those range of frequencies for which the gain is 1. For the latter frequencies, zeros are expected on either side of the curvature of the unit circle in order to keep the ripple small, i.e., the smaller the ripple specified, the larger the number and density of zeros. Figure 9 below confirms the expectations of the zero pole plot:

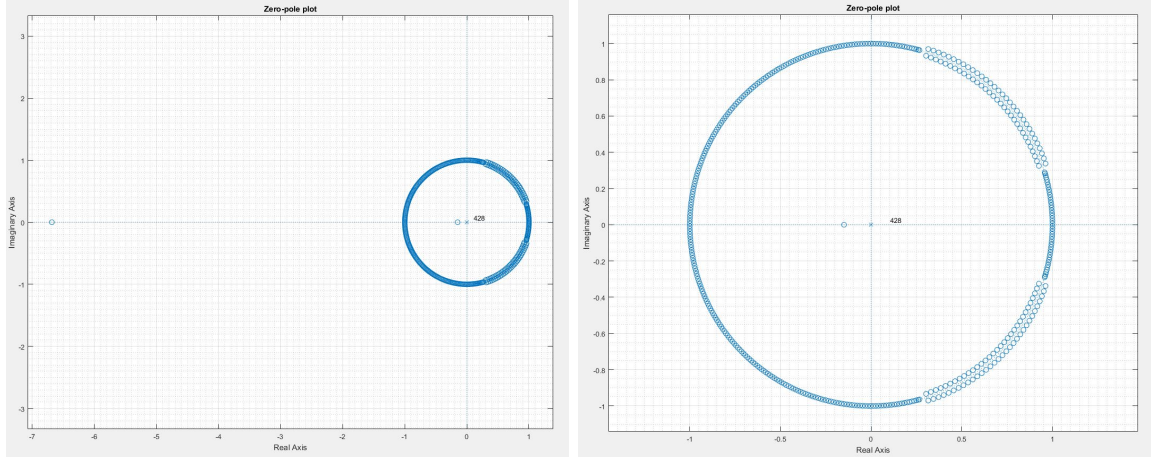


Figure 9: Zero-pole plot (right: zoomed in to unit circle)

However, contrary to expectations, there was a zero outside the unit circle, very far away from it on the real axis. This was found to be associated with the fact that the second transition band started at 1620Hz and not 1600Hz, as for the latter that zero occurred a lot closer to the unit circle.

### 3 Optimization Levels

Prior to writing the code, the optimization levels offered by Texas Instruments Code Composer Studio were looked at in depth, from the TI datasheet [2]. The CCS software offers different levels of optimisation, from no optimisation at all to level 3, to achieve efficient code. High-level optimisations are performed in the optimiser and low-level, target-specific optimizations occur in the code generator.

Each level builds upon previous ones by employing additional optimisation techniques:

- **No Optimisation**
- **Level 0 (-o0)**
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates unused code
  - Simplifies expressions and statements
  - Expands calls to functions declared inline
- **Level 1 (-o1)**

- Performs local copy/constant propagation
- Removes unused assignments
- Eliminates local common expressions

- **Level 2 (-o2)**

- Performs software pipelining
- Performs loop optimizations
- Eliminates global common subexpressions
- Eliminates global unused assignments
- Converts array references in loops to incremented pointer form
- Performs loop unrolling

- **Level 3 (-o3)**

- Removes all functions that are never called
- Simplifies functions with return values that are never used
- Inlines calls to small functions
- Reorders function declarations; the called functions attributes are known when the caller is optimized
- Propagates arguments into function bodies when all calls pass the same value in the same argument position
- Identifies file-level variable characteristics

In the following sections, the codes written were optimised to levels 0 (-o0) and 2 (-o2). However, levels 1 (-o1) and 3 (-o3) are not used. As can be seen above, level 1 (-o1) performs local changes and hence, the effects are not as pronounced as those for levels 2 (-o2) and 0 (-o0) when compared to lower levels, i.e. level 0 (-o0) and no optimisation respectively.

Level 3 (-o3), on the other hand, was not used for an entirely different reason. As can be seen above, it strips the code apart function by function and reorganises them in the most efficient way possible. This includes inlining functions and getting rid of unused functions. Since the method used for determining the number of clock cycles taken depends on breakpoints placed in the C code, the actual execution may perform more than it needs to while executing between breakpoints. Hence, the values obtained may not be relevant at all. However, if the breakpoints are placed in the optimised assembly code, the cycles can be accurately measured.

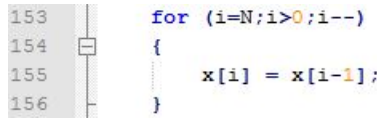
## 4 Non-circular FIR

### 4.1 Theory and code

The first FIR method to be implemented is an array shuffling method. This is based on assigning the 429 filter coefficients as calculated through the MATLAB code to an array `b`. `b` was not explicitly assigned within the code, but rather by reading the text file by a `#include` statement: `#include "fir_coefs.txt"`. This works only if the text file is formatted in the same way as a regular line of `c` code would, which is why the `fprintf` functions were included in the MATLAB code. For instance, `double b[] = {.....};`

The `main` function contains only an infinite while loop, and serves only the purpose of waiting for interrupts to vector to the ISR.

Within the ISR, the signal that is to be filtered is read from a signal generator into the DSK, and stored in a vector `x`. From here, each element in `x` is multiplied with its corresponding element in `b`. This is essentially a dot product, or convolution sum between the two vectors. After this, the `x` vector is shuffled to the right, removing the oldest value and inserting the newest value into the lowest index of the array (`x[0]`). This is done by a `for` loop as described in Figure 10 below:



```

153 for (i=N;i>0;i--)
154 {
155     x[i] = x[i-1];
156 }

```

Figure 10: Shuffling of input values, code

The shuffling is depicted graphically in Figure 11:

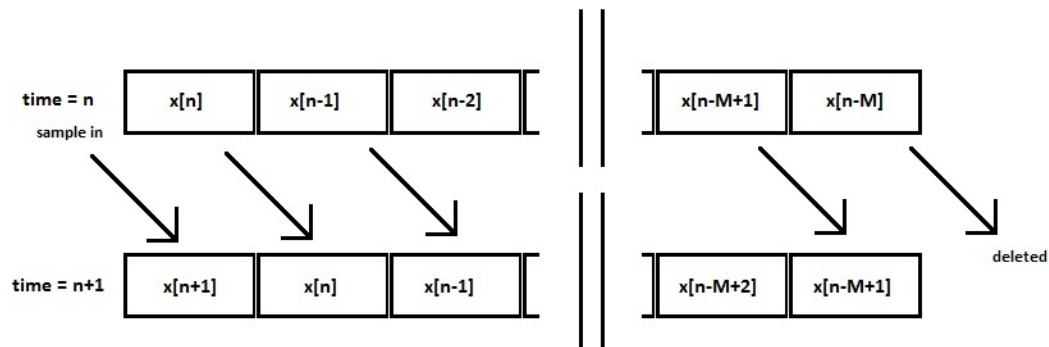


Figure 11: Shuffling of input values, graphical representation

As this shuffling operation requires a loop of  $N$  iterations, it is computationally expensive. The number of clock cycles required to perform the filtering operation at different levels of optimisation is displayed in Table 1 below.

Table 1: Non-circular FIR clock cycles

Optimisation level	Number of cycles
None	27187
0	22353
2	5596

As in the previous lab, the input is taken from the signal generator and read by `mono_read_16Bit()` and the output is done by `mono_write_16Bit()`.

## 4.2 Output signal

In order to verify that this code filters the input signal as desired, the output signals were tested with an oscilloscope.

To ensure that the low frequency stop and transition bands were filtered correctly, the signal was tested at 300Hz, 375Hz, 400Hz and 410Hz, as shown in Figure 12:

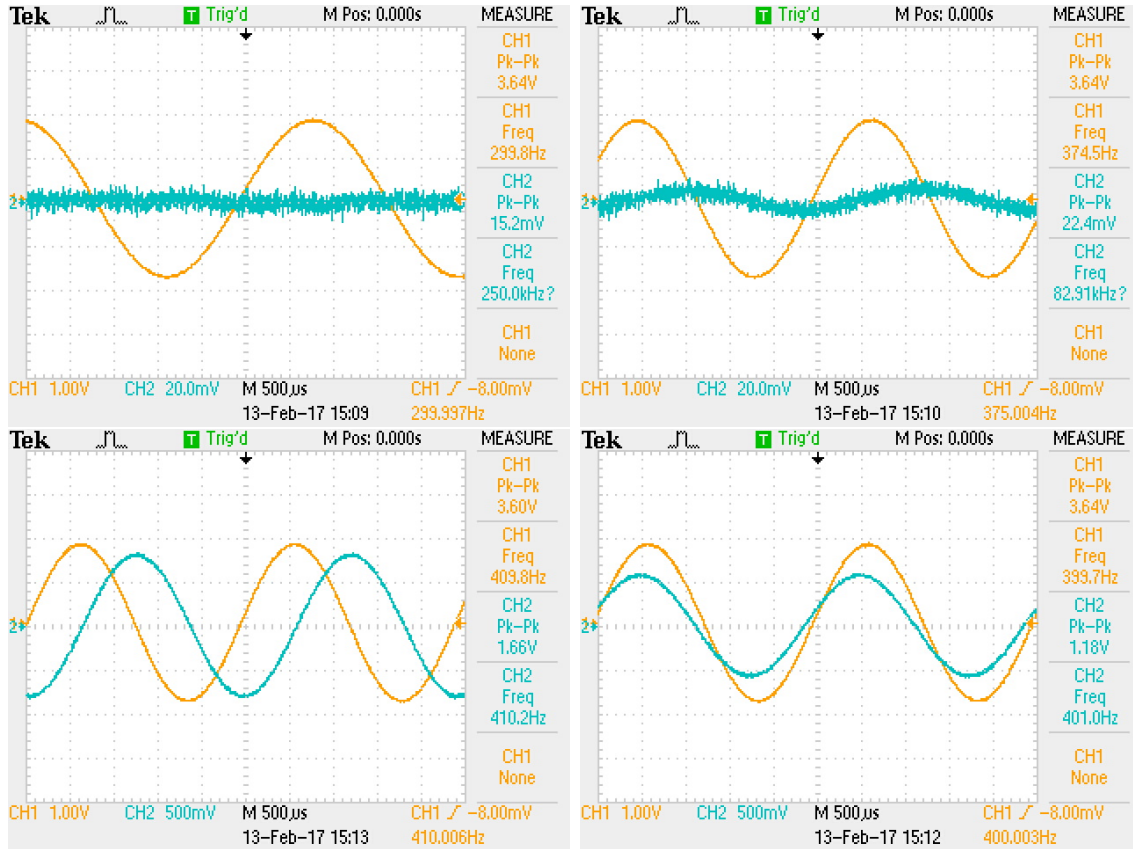


Figure 12: Output waveforms for inputs at frequencies of 300Hz, 375Hz, 400Hz, 410Hz (Clockwise)

As desired, there is very high attenuation at 300Hz. At 375Hz, the transition band begins and a sine wave of small amplitude is observed. 400Hz is in the middle of the transition band and thus there is only slight attenuation of the signal. For a 410Hz input, a full amplitude signal is produced.

To demonstrate the passband ripple, outputs of 1kHz and 1.3kHz were compared in Figure 13 below:

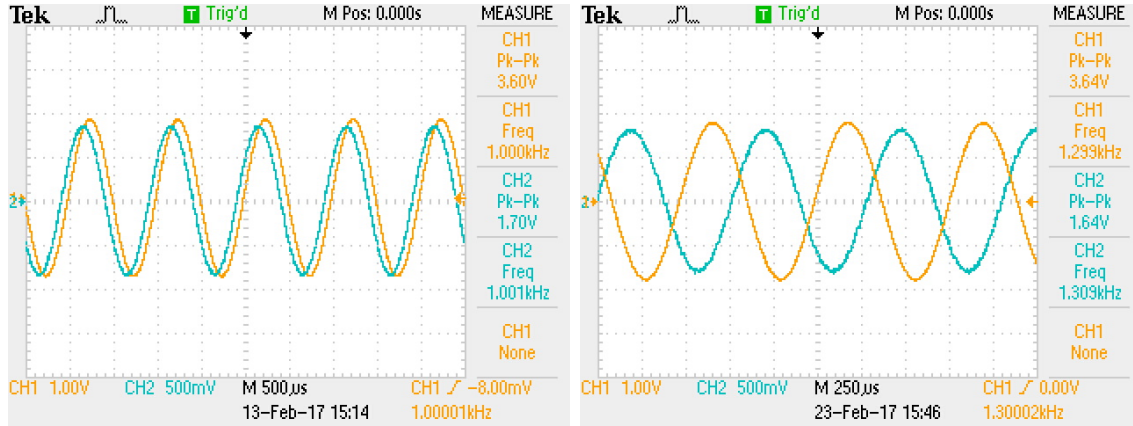


Figure 13: Output waveform at inputs of 1kHz and 1.3kHz

The difference in peak-to-peak amplitudes of these two outputs is 0.06V which is close to the maximum allowed deviation of  $0.03 \times 3.6V = 0.108$ . Further frequency sweeps showed that the requirements for the ripple were met across the entire passband. The reason that the output in the transition band is a half of the input is due to the fact that the two input channels have gains of a quarter, each. As the input function (`mono_read_16Bit()`) adds the two stereo channels to obtain a mono input, the net gain is a half. However, due to other components in the circuit, the gain is slightly below 50%. To verify the correct operation of high frequency stop and transition bands, inputs signals at frequencies of 1.6kHz, 1.625kHz, 1.645kHz and 1.655kHz were compared as can be seen in Figure 14:



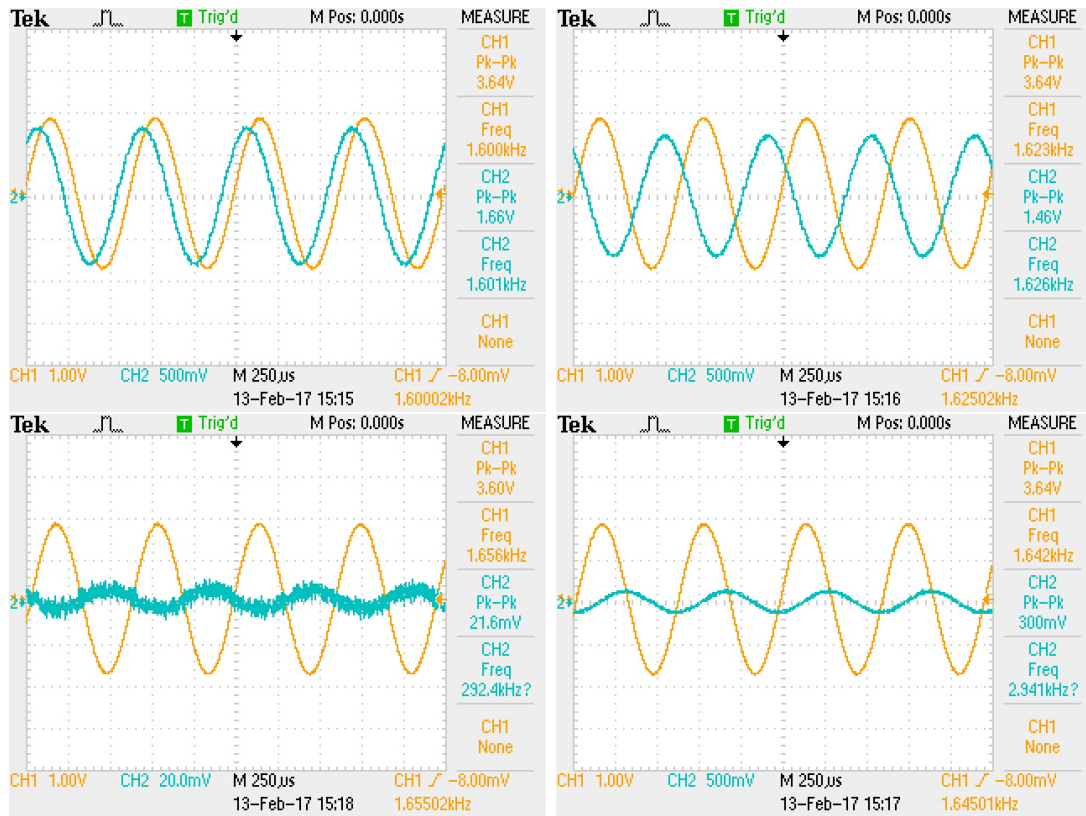


Figure 14: Output waveforms for inputs at frequencies of 1.6kHz, 1.625Hz, 1.645Hz, 1.655Hz (Clockwise)

A 1.6kHz input produces a full amplitude output, showing that that frequency is still within the passband as desired. At 1.625kHz the filter should be at the transition band and as such, slight attenuation of the signal is observed. Further attenuation occurs at 1.645kHz and a much larger attenuation at 1.655kHz. To show that frequencies beyond this are truly a stop band, one more observation was made at 2kHz (Figure 15), and as expected, the signal had an extremely small amplitude.

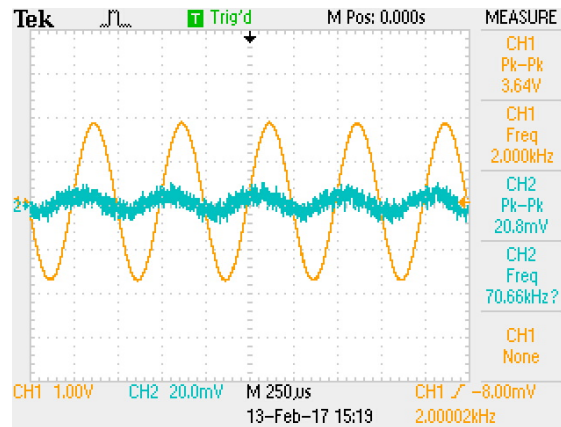


Figure 15: Output waveform at input of 2kHz



## 5 Non-circular FIR, symmetric coefficients

The next method of optimising the filtering was to exploit the fact that the filter is linear phase, meaning the coefficients are symmetric i.e.  $b[i] = b[N - i]$ . Physically this means that the  $i$ th newest sample will be multiplied with the an identical coefficient as the  $i$ th oldest sample (newest and oldest, second newest and second oldest and so on). These samples can therefore, be summed together and then multiplied with the corresponding coefficient, essentially halving the number of required multiplications.

Figure 16 graphically illustrates the samples that can be paired (samples of the same colour can be paired):

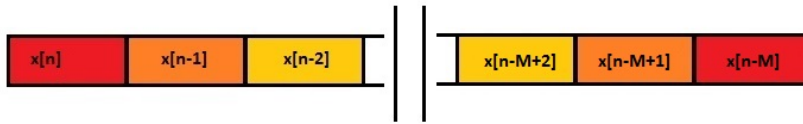


Figure 16: Graphical representation of symmetric coefficients for FIR filter of order M

However, as shown in equation (3) in section 1, filters with odd or even coefficients must be handled differently. This is due to the fact that filters with an odd number of coefficients will have a middle coefficient that will be multiplied with only one middle sample as it does not have a pair. The parts of the code used for calculating the mid point and performing the summing are shown in Figure 17 below:

```

88  if((N+1)%2==0) //calculate the mid point
89  {
90      mid = (N+1)/2 - 1;
91  }
92  else
93  {
94      mid = (N-2)/2;
95  }

183  for (i=0;i<=mid;i++)
184  {
185      output += b[i]*(x[i]+x[N-i]);
186  }
187  if((N+1)%2!=0)
188  {
189      output += b[N/2]*x[N/2];
190  }

```

Figure 17: Handling for odd or even number of coefficients

This code resulted in a large improvement in number of clock cycles required to perform the filtering as shown below:

Table 2: Non-circular FIR, linear phase

Optimisation level	Number of cycles
None	22976
0	17624
2	2966

## 6 Circular FIR

In the previous two FIR implementations, significant overhead is required to re-organise and maintain the order in the buffer, newest to oldest. This can be eliminated by implementing a circular buffer instead, which works by defining a global variable `ptr` that will increment successively through the elements of the buffer, essentially redefining the start point of the buffer. The resulting effect is that `ptr` will index the oldest sample, which will be replaced with the newest input whenever an IRQ is received.

When the convolution is performed, the first coefficient will be multiplied with the oldest sample, which is indexed as  $i = \text{ptr} + 1$ , and both will be incremented throughout every loop until the coefficient vector has been fully traversed. This is illustrated in Figure 18:

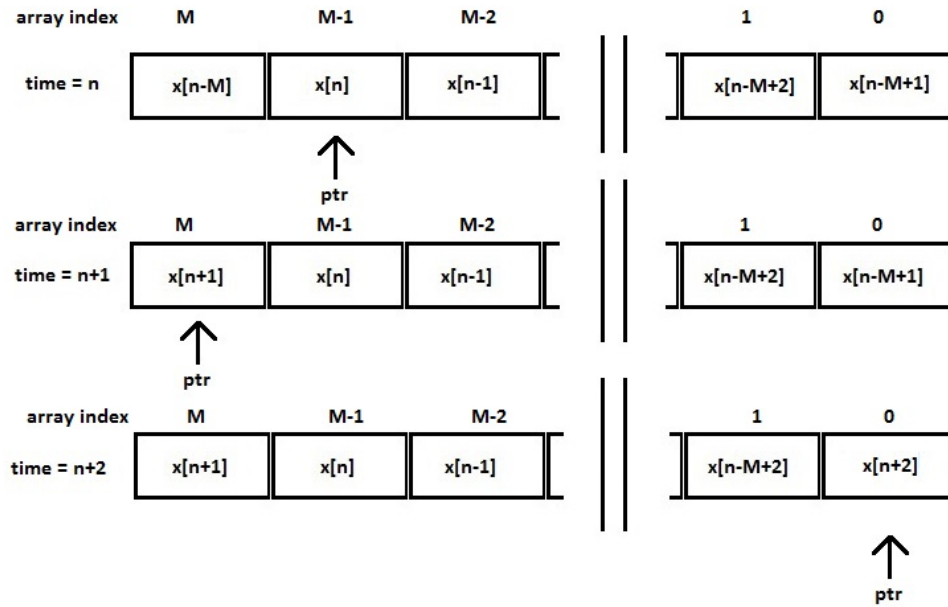


Figure 18: Circular Buffer Illustration for filter of order  $M$

Although the overhead in re-organisation is eliminated, there is a trade-off in terms of the increased complexity required to perform the convolution sum between  $b$  and  $x$ . This comes from the fact that all elements of the sample vector  $x$  must be traversed, and as `ptr` does not necessarily start at the first element of the vector, the maximum index of  $x$  may be exceeded. When this happens, the index should be reset, or wrapped, back to  $0$ . This feature requires a condition that must be checked after every incrementation of the index of  $x$  along with the loop variable. The conditional statement that handles this is shown in Figure 19:

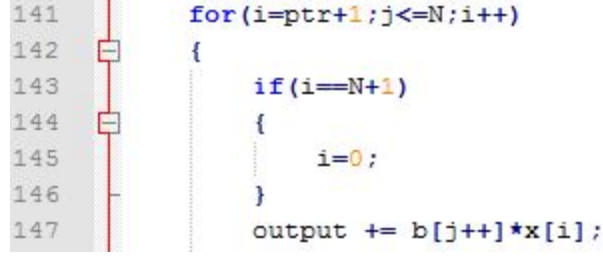


Figure 19: Convolution sum with index range handling

Despite this drawback of circular buffer implementations, an improvement was still observed over the non-circular counterpart, bearing in mind that the symmetry in coefficients is not yet exploited.

Table 3: Circular FIR, non-linear phase

Optimisation level	Number of cycles
None	25938
0	20712
2	4018

Note that due to the possibility of wrapping occurring, the number of cycles required to perform the ISR varied with each run. This differed from the non-circular buffer where the number of cycles required was constant at any point in time. From this point onwards, numbers taken for tables of cycle number will be the average from 20 samples.

## 7 Circular FIR, symmetric coefficients

### 7.1 Handling out of bounds with if statements

By considering the symmetry in filter coefficients, further improvements on the circular FIR implementation (in the previous section) can be made. As explained in section 5, the newest sample will be paired with the oldest and so on. However, the way that the circular buffer is constructed means that `ptr` will index the newest sample and `ptr+1` will be the oldest. Thus the procedure to perform the convolution sum is as follows:

- Input sample is read into `x[0]`
- Two indices are defined, `i` and `j`.
- `i` is initialized at `ptr` (initially 0) and `j` at `ptr+1`
- The required buffer of coefficients is indexed with variable `k`; initialised at 0.
- Within a loop:
  - `x[i]` and `x[j]` are summed and multiplied with the corresponding filter coefficient `b[k]`
  - The result is added to the output
  - `k` and `j` are incremented; `i` decremented
  - `i` and `j` are checked for wrap around and reinitialised in case of overflow or underflow
- Loop occurs until `k` reaches middle of  $N$  (evaluated using the variable `mid` for odd and even cases separately)
- As the number of coefficients ( $N+1$ ) is odd (in this case), there will be a middle coefficient with index  $\frac{N}{2}$ , where  $N$  is the order of the filter, that will only be multiplied with the middle input sample. This is handled outside the loop.
- Result is then added to the output
- Finally, `ptr` is incremented, wrapping back to 0 if it reaches  $N$  (order of the filter and maximum array index)

As with the circular buffer, there is the possibility of `i` or `j` decrementing or incrementing out of range of the buffer and so conditional statements must be written to handle these cases by indexing to  $N$  or 0 respectively. These `if` statements are checked every loop, adding significant overhead.

The convolution sum and index handling are shown in Figure 20 below:

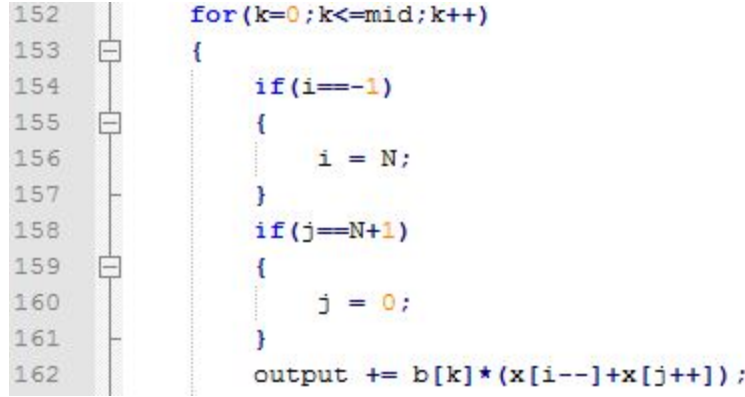


Figure 20: Linear phase convolution sum with index range handling

Despite this added complexity, the total number of cycles was reduced further still:

Table 4: Circular FIR, linear phase

Optimisation level	Number of cycles
None	17457
0	13610
2	2140

## 7.2 Wrap prediction - no `if` statements

One of the largest overheads associated with this FIR implementation is that a conditional statement must be assessed within the loop to check whether or not `i` or `j` have been decremented or incremented out of the range of the buffer. This can be eliminated by considering different cases depending on the positioning of `ptr`.

In order to do so, variables `wrap` and `diff` are defined. The former is either 0 or 1 depending on whether `j` or `i` will wrap around respectively. The latter is the difference between the position of the pointer that will wrap around (`i` or `j`) and the end (0 or `N`, respectively).

- $N - j \leq i - j$  will wrap around to 0
  - `wrap` set to 0
  - `diff` set to  $N - j$
  - for loop calculating output is run `diff` times
  - `j` wrapped around to 0
  - for loop run again `mid-diff` times where `mid` is the mid-point of the coefficient array `b`
  - Since the number of taps ( $N+1$ ) is odd, the product of the middle element and corresponding coefficient is added to the output (explained in section 7.1)

- **N - j > i** - *i* will wrap around to N
  - wrap set to 1
  - diff set to *i*
  - for loop calculating output is run diff times
  - *i* wrapped around to N
  - for loop run again mid-diff times where mid is the mid-point of the coefficient array *b*
  - Since the number of taps (N+1) is odd, the product of the middle element and corresponding coefficient is added to the output (explained in section 7.1)
- **ptr** is in the middle of the buffer - neither of *i* or *j* will wrap around and the for loop is run mid times

Therefore, where the loop performing the convolution sum used to perform  $\frac{N-2}{2}$  iterations (N is the order of the filter), whilst checking for wrap-arounds in each loop, it will now be performed for *k* = 0 to diff, after which the index that is to be wrapped will be set to 0 or N, and the convolution sum will be finished with another for loop of  $\frac{N-2}{2} - diff$  iterations. This is a lot more efficient (refer to Table 5 below) as earlier explained.

Table 5: Circular FIR, linear phase with pointer prediction

Optimisation level	Number of cycles
None	12400
0	10080
2	2200

An interesting fact was observed when comparing the results in Tables 4 and 5. As the code that implemented pointer prediction is meant to be a lot more efficient, it took fewer clock cycles at no optimisation and level 0 (-o0). However, at level 2, loops are unrolled and optimised. As a result of this optimisation, the former is slightly faster as the resulting optimised assembly code was more efficient. This is because simple codes are easily identified by the compiler as multiply-accumulate (MAC) implementations, and hence optimised a lot more than others.

## 8 Circular FIR, Double Buffer

To eliminate the problem of out of bound indices completely, an input buffer of double the size was implemented.

In this implementation,  $x$  is assigned a size of  $2(N + 1)$  instead of  $(N + 1)$ . The list of inputs is repeated twice, i.e. the input values come in order as the first  $N + 1$  elements, and again from the  $N + 2$ th element until the end. In terms of the array indices, the input at  $x[i]$  is identical to the input at  $x[N+1+i]$ .

To perform the convolution sum:

- $ptr$  is again assigned and initialised as 0
- Input sample is taken into buffer at  $x[ptr]$  and replicated at  $x[ptr+N+1]$
- $i$  is initialized at  $ptr+1$  and  $j$  at  $ptr+N+1$
- Within a loop:
  - $x[i]$  and  $x[j]$  are summed and multiplied with the corresponding filter coefficient  $b[k]$
  - The result is added to the output
  - $k$  and  $i$  are incremented;  $j$  decremented
- Loop occurs until  $k$  reaches middle of  $N$  (evaluated using the variable  $mid$  for odd and even cases separately)
- As the number of coefficients  $(N+1)$  is odd (in this case), there will be a middle coefficient with index  $\frac{N}{2}$ , where  $N$  is the order of the filter, that will only be multiplied with the middle input sample. This is handled outside the loop.
- Result is then added to the output
- Finally,  $ptr$  is incremented, wrapping back to 0 if it reaches  $N$  (order of the filter and maximum array index)

Thus wrapping is avoided for the index pointers as  $i$  and  $j$  will never exceed the boundaries of the buffer. The trade offs associated with this implementation are memory costs of having a double sized buffer. In addition to this, further overhead is added by having to add another element to the buffer for each input.

However, pronounced improvements in efficiency were still made as documented below:

Table 6: Double buffer

Optimisation level	Number of cycles
None	11688
0	9513
2	1833

## 9 Further Improvements

Ultimately, the double buffer size code required the lowest number of cycles, but small improvements could still be made to improve efficiency, including inlining the FIR function in the ISR. Another improvement will be to minimise the number of global variables that are used, to local variables within the ISR to side-step the disadvantages of global variables such as data leakage (data can be modified by any function) and the fact that global variables remain in the memory till the execution completes.

The biggest improvement however was to change the output and input and coefficient array types from double precision `double` to single precision `float`. The resulting multiplications would be between two IEEE 754 32-bits numbers, as opposed to two IEEE 754 64-bit number, which would require more clock cycles to perform. Although precision is reduced, this alteration can be made relatively safely due to the aspect of the FIR where errors will not accumulate infinitely and as the difference in precision between the two variable types is very small for 429 coefficients (explained in section 1). This is not the case for IIR filters.

Other smaller optimisations include getting rid of customisability, i.e. making the code specific to a certain number of coefficients, in this case 429. `mid` can be assumed fixed at  $\frac{N-2}{2}$  as for this case the number of coefficients is odd.

After all these modifications were implemented, a minimum of **420** clock cycles was achieved for the MAC loop. When taking into account adding inputs to the input buffer and pointer initialisations and incrementations, the number of clock cycles was **443**. Moreover, the outputs were the same as before confirming that the change in variable precision does not affect the filtering operation.

However, as customisability (in terms on `N` and `mid`) is an important feature, after completely optimising the code, this was put back in and the code performed the operation in **942** cycles. To further the point of customisability, `N` was initialised using the `sizeof` function in `c`, rather than hard-coding the number 428 in.

**Note:** Whenever clock cycles are measured in the previous sections, they do not include the time taken by `mono_read_16Bit()` and `mono_write_16Bit()`. The breakpoints are placed around the call to the filtering function in the ISR. For the case of inline filtering within the ISR, the breakpoints are placed after `mono_read_16Bit()`.

When the above optimisations are added to the code described in section 7.2 (along with customisability in `N` and `mid`), the number of cycles taken reduced to **1090**.



## 10 Spectrum Analysis

To further verify that the filter was working as intended, an Audio Precision APX520 was used to measure the frequency response. The fastest c implementation, circular double buffer, was tested. The following plot was obtained:

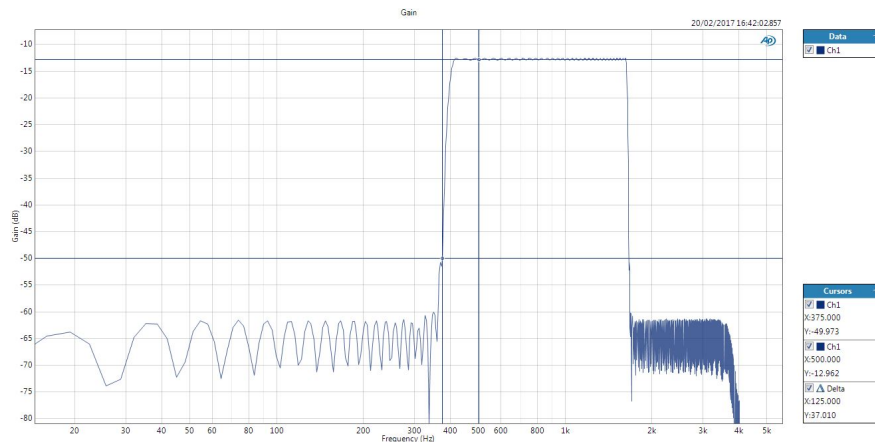


Figure 21: Filter amplitude response from APX520 (logarithmic frequency axis)

One obscurity to note is that the gain at the passband is not at 0dB, but at roughly -13dB. This is due to the gain of a quarter at the input stage of the DSK, which is -12dB. The reason this was different from the gain obtained while using the signal generator is because instead of using the 3.5mm cable input which is stereo, only one of the lines is used. As a result, the signal observed on the oscilloscope has a gain of a half, and the spectrum analyser observes gain of a quarter.

To make rigorous checks on the performance of the filter, closer inspections were made:

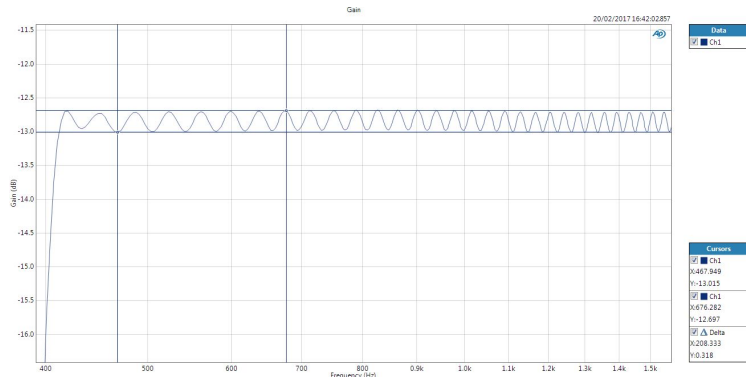


Figure 22: Close inspection of frequency response (passband ripple)

It is evident from the plot above that the size of the passband has a peak-to-peak difference of 0.318dB which is within the required limits. Figure 21 shows that the stop band

attenuation requirements are not met right at 375Hz, as the difference in gains is 37dB, not 46dB as required. For the other stop band, the attenuation is about 48dB (beyond expectations). It must be noted however, that at the higher end, the frequency at which attenuation is measured is 80Hz beyond the start of the stop-band (1700Hz and 1620Hz) while on the other side, the corresponding measurement is 35Hz (375Hz and 410Hz). When both are taken equidistant from the passband, i.e., at 375Hz and 1655Hz, the symmetry is seen to be maintained while both have attenuations below 46dB. This is because the filter implemented does not meet the specifications completely due to non-idealities in the implementation.

The phase response of the filter was also checked on the APX520:

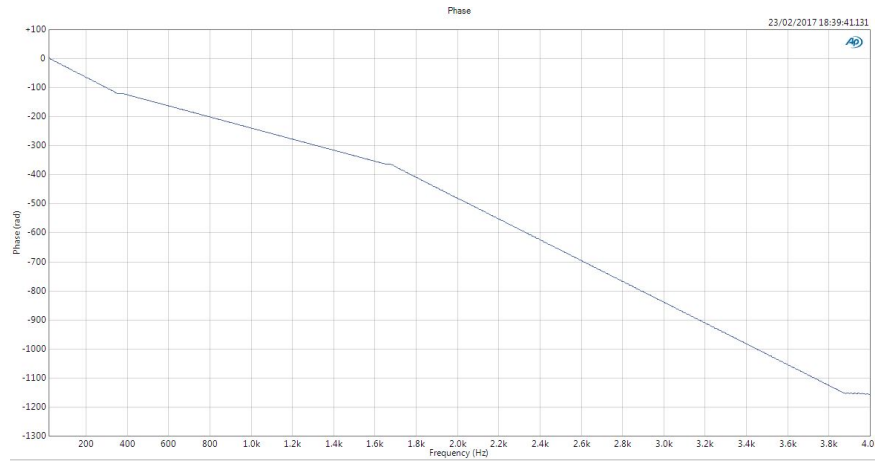


Figure 23: Filter phase response from APX520

Thus, the phase response is linear within the passband range, as expected. However, the phase response at other frequency ranges was not flat, differing again from expected (MATLAB implementation). This is probably because outside the passband, the signal amplitude is so small that the APX520 might not be able to recognise the signal. The phase response of the passband is seen below in Figure 24:

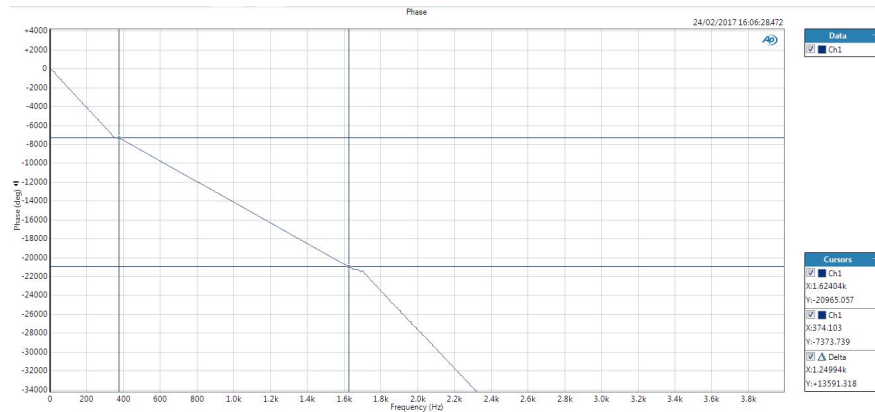


Figure 24: Filter phase response from APX520 in passband ranges

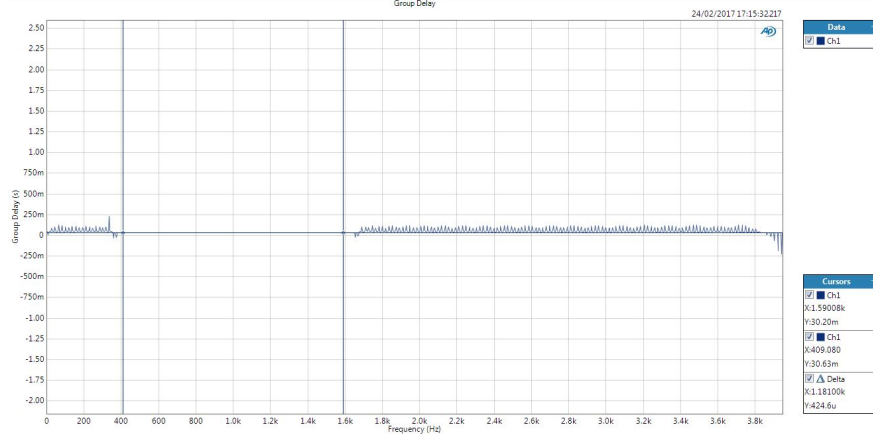


Figure 25: Group delay associated with the phase response

At passband frequencies, the group delay is constant, as explained in section 1. The measured group delay is roughly 30ms as shown in Figure 25. On the other hand, the theoretical group delay is:

$$\text{Group delay (ms)} = \frac{N}{2 \times f_s} = \frac{428}{2 \times 8000} = 26.75\text{ms}$$

The group delay is quite close to the theoretical value. Figure 25 also shows that the group delay is not constant outside the passband and hence, the variations outside the passband are probably not due to a constant group delay introduced by the APX 520.

To summarise the above results, the filter implemented is almost an ideal linear phase filter with stopband attenuation slightly smaller than required by specification and non-constant phase outside the passband (possibly due to spectrum analyser).

## 10.1 Improved filter designs

Attempts were made to design a filter that would meet the attenuation specification at 375Hz. This was done by going back into MATLAB and decreasing the ripple and stop band attenuation limits, as well as decreasing transition bandwidth. Despite these attempts, the maximum attenuation that could be achieved was roughly -39dB.

## References

- [1] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms and Applications, 3rd Edition*. 1996.
- [2] Texas Instruments. *TMS320C6000 Optimizing Compiler v7.4*. 2012. [Online; accessed 2017-02-20].

# Appendices

## A MATLAB code for obtaining FIR coefficients

```
1 %calculate filter coefficients
2 [N,Fo,Ao,W]=firlmord([375,410,1620,1655],[0,1,0],[0.005,0.023,0.005],8000);
3 b = firlm(N,Fo,Ao,W);
4 a = 1;
5 a(end) = 1;
6
7 %calculate frequency and phase responses
8 [h,w] = freqz(b);
9 [phi,w] = phasez(b);
10
11 %plot amplitude response
12 figure
13 plot(w*4000/pi,20*log10(abs(h)));
14 xlabel('Frequency (Hz)');
15 ylabel('Gain (dB)');
16 title('Amplitude Response of the filter');
17 ax.XTick = 0:.5:2;
18 grid on
19 grid minor
20
21 %plot phase response
22 figure
23 plot(w*4000/pi,phi);
24 xlabel('Frequency (Hz)');
25 ylabel('Phase (degrees)');
26 title('Phase Response of the filter');
27 ax.XTick = 0:.5:2;
28 grid on
29 grid minor
30
31 %plot filter coefficients
32 figure
33 stem(b);
34 xlabel('Index number (n)');
35 ylabel('b[n]');
36 title('Impulse Response of the filter');
37 ax.XTick = 0:1:size(b);
38 grid on
39 grid minor
40
41 %zero-pole plot
42 figure
43 zplane(b,a);
44 xlabel('Real Axis');
45 ylabel('Imaginary Axis');
46 title('Zero-pole plot');
47 grid on
48 grid minor
```

```
49
50
51 %save filter coefficients to
52 fileID = fopen('fir_coef2.txt','w');
53 fprintf(fileID, 'double b[] = {');
54 fprintf(fileID, '%.15e, ', b);
55 fprintf(fileID, '};');
```

## B Non-circular FIR

```
1  /*****
2      DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3      IMPERIAL COLLEGE LONDON
4
5      EE 3.19: Real Time Digital Signal Processing
6      Dr Paul Mitcheson and Daniel Harvey
7
8      LAB 3: Interrupt I/O
9
10     ***** I N T I O. C *****
11
12     Demonstrates inputting and outputting data from the DSK's audio port
13     using interrupts.
14
15     *****
16     Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
17     Updated for CCS V4 Sept 10
18     *****/
19     /*
20     * You should modify the code so that interrupts are used to service
21     * the
22     * audio port.
23     */
24     /***** Pre-processor statements
25     *****/
26
27     #include <stdlib.h>
28     // Included so program can make use of DSP/BIOS configuration tool.
29     #include "dsp_bios_cfg.h"
30
31     /* The file dsk6713.h must be included in every program that uses the
32     BSL. This
33     example also includes dsk6713_aic23.h because it uses the
34     AIC23 codec module (audio interface). */
35     #include "dsk6713.h"
36     #include "dsk6713_aic23.h"
37
38     // math library (trig functions)
39     #include <math.h>
40
41     // Some functions to help with writing/reading the audio ports when
42     using interrupts.
43     #include <helper_functions_ISR.h>
44     #include "fir_coef.txt"
45
46     #define N sizeof(b)/sizeof(float)-1 //customisable order of filter
47     double x[N+1]; //input buffer
48     double output,sample; //initialisation of global variables
49     /***** Global declarations
50     *****/
```

```

46  /* Audio port configuration settings: these values set registers in the
47     AIC23 audio
48     interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more
49     info. */
48  DSK6713_AIC23_Config Config = { \
49      /*
50          ****
51          */
52          /* REGISTER          FUNCTION          SETTINGS
53          */
54          /*
55          ****
56          */
57          0x0017, /* 0 LEFTINVOL Left line input channel volume 0dB
58                  */
59          0x0017, /* 1 RIGHTINVOL Right line input channel volume 0dB
60                  */
61          0x01f9, /* 2 LEFTHPVOL Left channel headphone volume 0dB
62                  */
63          0x01f9, /* 3 RIGHTHPVOL Right channel headphone volume 0dB
64                  */
65          0x0011, /* 4 ANAPATH Analog audio path control DAC on,
66                  Mic boost 20dB*/
67          0x0000, /* 5 DIGPATH Digital audio path control All
68                  Filters off */
69          0x0000, /* 6 DPOWERDOWN Power down control All
70                  Hardware on */
71          0x0043, /* 7 DIGIF Digital audio interface format 16 bit
72                  */
73          0x008d, /* 8 SAMPLERATE Sample rate control 8 KHZ
74                  */
75          0x0001 /* 9 DIGACT Digital interface activation On
76                  */
77      /*
78          ****
79          */
80  };
81
82  // Codec handle:- a variable used to identify audio interface
83  DSK6713_AIC23_CodecHandle H_Codec;
84
85  /***** Function prototypes
86  *****/
87  void init_hardware(void);
88  void init_HWI(void);
89  void ISR_AIC(void);
90  void non_circ_FIR(); //function that implements the FIR
91  /***** Main routine
92  *****/
93  void main(){
94
95      // initialize board and the audio port

```

```

79  init_hardware();
80
81  /* initialize hardware interrupts */
82  init_HWI();
83
84  /* loop indefinitely, waiting for interrupts */
85  while(1)
86  {};
87
88  }
89
90  ***** init_hardware()
*****
91  void init_hardware()
92  {
93      // Initialize the board support library, must be called first
94      DSK6713_init();
95
96      // Start the AIC23 codec using the settings defined above in config
97      H_Codec = DSK6713_AIC23_openCodec(0, &Config);
98
99      /* Function below sets the number of bits in word used by MSBSP (
serial port) for
100     receives from AIC23 (audio port). We are using a 32 bit packet
containing two
101     16 bit numbers hence 32BIT is set for receive */
102     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
103
104     /* Configures interrupt to activate on each consecutive available 32
bits
105     from Audio port hence an interrupt is generated for each L & R sample
pair */
106     MCBSP_FSETS(SPCR1, RINTM, FRM);
107
108     /* These commands do the same thing as above but applied to data
transfers to
109     the audio port */
110     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
111     MCBSP_FSETS(SPCR1, XINTM, FRM);
112
113
114  }
115
116  ***** init_HWI()
*****
117  void init_HWI(void)
118  {
119      IRQ_globalDisable();      // Globally disables interrupts
120      IRQ_nmiEnable();          // Enables the NMI interrupt (used by the
debugger)
121      IRQ_map(IRQ_EVT_RINT1,4);  // Maps an event to a physical interrupt
122      IRQ_enable(IRQ_EVT_RINT1); // Enables the event
123      IRQ_globalEnable();        // Globally enables interrupts
124  }

```



```

125
126 /****** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE
    ******/
127
128 void ISR_AIC()
129 {
130     sample = mono_read_16Bit();    //takes input sample
131     output=0;                      //resetting output to zero for accumulation
132     non_circ_FIR();
133     mono_write_16Bit((short)floor(output));    //writing to output
134 }
135
136 void non_circ_FIR()                //non-circular implementation of FIR
137 {
138     int i;
139     for(i=N;i>0;i--)                //linear shifting of data in input buffer
140     {
141         x[i]=x[i-1];
142     }
143     x[0] = sample;                  //newest input sample put at beginning of
    buffer
144     for(i=0;i<=N;i++)                //MAC implementation
145     {
146         output += b[i]*x[i];
147     }
148 }

```

## C Non-circular FIR, symmetric coefficients

```
1  /*****
2      DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3      IMPERIAL COLLEGE LONDON
4
5      EE 3.19: Real Time Digital Signal Processing
6      Dr Paul Mitcheson and Daniel Harvey
7
8      LAB 3: Interrupt I/O
9
10     ***** I N T I O. C *****
11
12     Demonstrates inputting and outputting data from the DSK's audio port
13     using interrupts.
14
15     *****
16     Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
17     Updated for CCS V4 Sept 10
18     *****/
19 /*
20  * You should modify the code so that interrupts are used to service
21  * the
22  * audio port.
23  */
24 /***** Pre-processor statements
25     *****/
26
27 #include <stdlib.h>
28 // Included so program can make use of DSP/BIOS configuration tool.
29 #include "dsp_bios_cfg.h"
30
31 /* The file dsk6713.h must be included in every program that uses the
32    BSL. This
33    example also includes dsk6713_aic23.h because it uses the
34    AIC23 codec module (audio interface). */
35 #include "dsk6713.h"
36 #include "dsk6713_aic23.h"
37
38 // math library (trig functions)
39 #include <math.h>
40
41 // Some functions to help with writing/reading the audio ports when
42 // using interrupts.
43 #include <helper_functions_ISR.h>
44 #include "fir_coef.txt"
45
46 #define N sizeof(b)/sizeof(float)-1 //customisable order of filter
47 double x[N+1]; //input buffer
48 int mid; //middle element in buffer
49 double output, sample; //initialisation of global variables
50 /***** Global declarations
51     *****/
```

```

46
47 /* Audio port configuration settings: these values set registers in the
48    AIC23 audio
49    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more
50    info. */
49 DSK6713_AIC23_Config Config = { \
50     /*
51     ****
52     */
51     /* REGISTER FUNCTION SETTINGS
52     */
52     /*
53     ****
54     */
53     0x0017, /* 0 LEFTINVOL Left line input channel volume 0dB
54              */
54     0x0017, /* 1 RIGHTINVOL Right line input channel volume 0dB
55              */
55     0x01f9, /* 2 LEFTHPVOL Left channel headphone volume 0dB
56              */
56     0x01f9, /* 3 RIGHTHPVOL Right channel headphone volume 0dB
57              */
57     0x0011, /* 4 ANAPATH Analog audio path control DAC on,
58              Mic boost 20dB */
58     0x0000, /* 5 DIGPATH Digital audio path control All
59              Filters off */
59     0x0000, /* 6 DPOWERDOWN Power down control All
60              Hardware on */
60     0x0043, /* 7 DIGIF Digital audio interface format 16 bit
61              */
61     0x008d, /* 8 SAMPLERATE Sample rate control 8 KHZ
62              */
62     0x0001 /* 9 DIGACT Digital interface activation On
63              */
63     /*
64     ****
65     */
64 };
65
66
67 // Codec handle:- a variable used to identify audio interface
68 DSK6713_AIC23_CodecHandle H_Codec;
69
70 /****** Function prototypes
71 *****/
71 void init_hardware(void);
72 void init_HWI(void);
73 void ISR_AIC(void);
74 void non_circ_FIR(); //function that implements the FIR
75 /****** Main routine
76 *****/
76 void main(){
77
78     if(N+1%2==0) //middle element calculation for odd and even

```

```

    cases
79 {           //as loop counter limit
80     mid = (N-1)/2;
81 }
82 else
83 {
84     mid = (N-2)/2;
85 }
86
87 // initialize board and the audio port
88 init_hardware();
89
90 /* initialize hardware interrupts */
91 init_HWI();
92
93 /* loop indefinitely, waiting for interrupts */
94 while(1)
95 {}
96
97 }
98
99 /***** init_hardware()
    *****/
100 void init_hardware()
101 {
102     // Initialize the board support library, must be called first
103     DSK6713_init();
104
105     // Start the AIC23 codec using the settings defined above in config
106     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
107
108     /* Function below sets the number of bits in word used by MSBSP (
        serial port) for
109     receives from AIC23 (audio port). We are using a 32 bit packet
        containing two
110     16 bit numbers hence 32BIT is set for receive */
111     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
112
113     /* Configures interrupt to activate on each consecutive available 32
        bits
114     from Audio port hence an interrupt is generated for each L & R sample
        pair */
115     MCBSP_FSETS(SPCR1, RINTM, FRM);
116
117     /* These commands do the same thing as above but applied to data
        transfers to
118     the audio port */
119     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
120     MCBSP_FSETS(SPCR1, XINTM, FRM);
121
122
123 }
124
125 /***** init_HWI()

```

```

126 void init_HWI(void)
127 {
128     IRQ_globalDisable();    // Globally disables interrupts
129     IRQ_nmiEnable();        // Enables the NMI interrupt (used by the
                             // debugger)
130     IRQ_map(IRQ_EVT_RINT1,4); // Maps an event to a physical interrupt
131     IRQ_enable(IRQ_EVT_RINT1); // Enables the event
132     IRQ_globalEnable();      // Globally enables interrupts
133 }
134
135 /***** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE *****/
136
137 void ISR_AIC()
138 {
139     sample = mono_read_16Bit(); //takes input sample
140     output=0;                  //resetting output to zero for accumulation
141     non_circ_FIR();
142     mono_write_16Bit((short)floor(output)); //writing to output
143 }
144
145 void non_circ_FIR()           //non-circular buffer implementation of FIR
                             //with symmetry
146 {
147     int i;
148     for(i=N;i>0;i--)          //linear shifting of data in input buffer
149     {
150         x[i]=x[i-1];
151     }
152     x[0] = sample;            //newest input sample put at beginning of
                             //buffer
153     for(i=0;i<=mid;i++)       //MAC implementation
154     {
155         output += b[i]*(x[i]+x[N-i]);
156     }
157     if(N+1%2!=0)              //handling of middle element in case of odd
                             //number of taps
158     {
159         output += b[mid+1]*x[mid+1];
160     }
161 }

```

## D Circular FIR

```
1  /*****
2      DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3      IMPERIAL COLLEGE LONDON
4
5      EE 3.19: Real Time Digital Signal Processing
6      Dr Paul Mitcheson and Daniel Harvey
7
8      LAB 3: Interrupt I/O
9
10     ***** I N T I O . C *****
11
12     Demonstrates inputting and outputting data from the DSK's audio port
13     using interrupts.
14
15     *****
16     Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
17     Updated for CCS V4 Sept 10
18     *****/
19 /*
20  * You should modify the code so that interrupts are used to service
21  * the
22  * audio port.
23  */
24 /***** Pre-processor statements
25     *****/
26
27 #include <stdlib.h>
28 // Included so program can make use of DSP/BIOS configuration tool.
29 #include "dsp_bios_cfg.h"
30
31 /* The file dsk6713.h must be included in every program that uses the
32    BSL. This
33    example also includes dsk6713_aic23.h because it uses the
34    AIC23 codec module (audio interface). */
35 #include "dsk6713.h"
36 #include "dsk6713_aic23.h"
37
38 // math library (trig functions)
39 #include <math.h>
40
41 // Some functions to help with writing/reading the audio ports when
42 // using interrupts.
43 #include <helper_functions_ISR.h>
44 #include "fir_coef.txt"
45
46 #define N sizeof(b)/sizeof(float)-1 //customisable order of filter
47 double x[N+1]; //input buffer
48 double output,sample; //initialisation of global variables
49 int ptr=0; //circular buffer input pointer
50 /***** Global declarations
51     *****/
```

```

46
47 /* Audio port configuration settings: these values set registers in the
48    AIC23 audio
49    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more
50    info. */
49 DSK6713_AIC23_Config Config = { \
51     /*
52     *****
53     */
51     /* REGISTER FUNCTION SETTINGS
52     */
52     /*
53     *****
54     */
53     0x0017, /* 0 LEFTINVOL Left line input channel volume 0dB
54              */
54     0x0017, /* 1 RIGHTINVOL Right line input channel volume 0dB
55              */
55     0x01f9, /* 2 LEFTHPVOL Left channel headphone volume 0dB
56              */
56     0x01f9, /* 3 RIGHTHPVOL Right channel headphone volume 0dB
57              */
57     0x0011, /* 4 ANAPATH Analog audio path control DAC on,
58              Mic boost 20dB */
58     0x0000, /* 5 DIGPATH Digital audio path control All
59              Filters off */
59     0x0000, /* 6 DPOWERDOWN Power down control All
60              Hardware on */
60     0x0043, /* 7 DIGIF Digital audio interface format 16 bit
61              */
61     0x008d, /* 8 SAMPLERATE Sample rate control 8 KHZ
62              */
62     0x0001, /* 9 DIGACT Digital interface activation On
63              */
63     /*
64     *****
65     */
64 };
65
66
67 // Codec handle:- a variable used to identify audio interface
68 DSK6713_AIC23_CodecHandle H_Codec;
69
70 /****** Function prototypes
71 *****/
71 void init_hardware(void);
72 void init_HWI(void);
73 void ISR_AIC(void);
74 void circ_FIR(); //function that implements the FIR
75 /****** Main routine
76 *****/
76 void main(){
77
78 // initialize board and the audio port

```

```

79  init_hardware();
80
81  /* initialize hardware interrupts */
82  init_HWI();
83
84  /* loop indefinitely, waiting for interrupts */
85  while(1)
86  {};
87
88  }
89
90  ***** init_hardware()
*****
91  void init_hardware()
92  {
93      // Initialize the board support library, must be called first
94      DSK6713_init();
95
96      // Start the AIC23 codec using the settings defined above in config
97      H_Codec = DSK6713_AIC23_openCodec(0, &Config);
98
99      /* Function below sets the number of bits in word used by MSBSP (
serial port) for
100     receives from AIC23 (audio port). We are using a 32 bit packet
containing two
101     16 bit numbers hence 32BIT is set for receive */
102     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
103
104     /* Configures interrupt to activate on each consecutive available 32
bits
105     from Audio port hence an interrupt is generated for each L & R sample
pair */
106     MCBSP_FSETS(SPCR1, RINTM, FRM);
107
108     /* These commands do the same thing as above but applied to data
transfers to
109     the audio port */
110     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
111     MCBSP_FSETS(SPCR1, XINTM, FRM);
112
113
114  }
115
116  ***** init_HWI()
*****
117  void init_HWI(void)
118  {
119      IRQ_globalDisable();      // Globally disables interrupts
120      IRQ_nmiEnable();          // Enables the NMI interrupt (used by the
debugger)
121      IRQ_map(IRQ_EVT_RINT1,4);  // Maps an event to a physical interrupt
122      IRQ_enable(IRQ_EVT_RINT1); // Enables the event
123      IRQ_globalEnable();        // Globally enables interrupts
124  }

```



```

125
126 /****** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE
    ******/
127
128 void ISR_AIC()
129 {
130     sample = mono_read_16Bit();    //takes input sample
131     output=0;                      //resetting output to zero for accumulation
132     circ_FIR();
133     mono_write_16Bit((short)floor(output));    //writing to output
134 }
135
136 void circ_FIR()                    //circular buffer implementation of FIR
137 {
138     int i,j;
139     j = 0;
140     x[ptr] = sample;                //sample input into index ptr
141     for(i=ptr+1;j<=N;i++)           //MAC implementation
142     {
143         if(i==N+1)                  //conditional statement to check wrap around of
            i
144         {
145             i=0;
146         }
147         output += b[j++]*x[i];
148     }
149
150     if(ptr==N)                      //conditional statement to check wrap around of
        ptr
151     {
152         ptr=0;
153     }
154     else
155     {
156         ptr++;                      //incrementation of ptr
157     }
158 }

```

## E Circular FIR, symmetric coefficients

```
1  /*****
2      DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3      IMPERIAL COLLEGE LONDON
4
5      EE 3.19: Real Time Digital Signal Processing
6      Dr Paul Mitcheson and Daniel Harvey
7
8      LAB 3: Interrupt I/O
9
10     ***** I N T I O. C *****
11
12     Demonstrates inputting and outputting data from the DSK's audio port
13     using interrupts.
14
15     *****
16     Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
17     Updated for CCS V4 Sept 10
18     *****/
19 /*
20  * You should modify the code so that interrupts are used to service
21  * the
22  * audio port.
23  */
24 /***** Pre-processor statements
25     *****/
26
27 #include <stdlib.h>
28 // Included so program can make use of DSP/BIOS configuration tool.
29 #include "dsp_bios_cfg.h"
30
31 /* The file dsk6713.h must be included in every program that uses the
32    BSL. This
33    example also includes dsk6713_aic23.h because it uses the
34    AIC23 codec module (audio interface). */
35 #include "dsk6713.h"
36 #include "dsk6713_aic23.h"
37
38 // math library (trig functions)
39 #include <math.h>
40
41 // Some functions to help with writing/reading the audio ports when
42 // using interrupts.
43 #include <helper_functions_ISR.h>
44 #include "fir_coef.txt"
45
46 #define N sizeof(b)/sizeof(float)-1 //customisable order of filter
47 double x[N+1]; //input buffer
48 double output,sample; //initialisation of global variables
49 int ptr=0; //circular buffer input pointer
50 int mid; //middle element of input buffer
51 /***** Global declarations
```

```

47 *****/
48 /* Audio port configuration settings: these values set registers in the
49    AIC23 audio
50    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more
51    info. */
52 DSK6713_AIC23_Config Config = { \
53     /*
54     *****
55     */
56     /* REGISTER FUNCTION SETTINGS
57     */
58     /*
59     *****
60     */
61     0x0017, /* 0 LEFTINVOL Left line input channel volume 0dB
62              */
63     0x0017, /* 1 RIGHTINVOL Right line input channel volume 0dB
64              */
65     0x01f9, /* 2 LEFTHPVOL Left channel headphone volume 0dB
66              */
67     0x01f9, /* 3 RIGHTHPVOL Right channel headphone volume 0dB
68              */
69     0x0011, /* 4 ANAPATH Analog audio path control DAC on,
70              Mic boost 20dB */
71     0x0000, /* 5 DIGPATH Digital audio path control All
72              Filters off */
73     0x0000, /* 6 DPOWERDOWN Power down control All
74              Hardware on */
75     0x0043, /* 7 DIGIF Digital audio interface format 16 bit
76              */
77     0x008d, /* 8 SAMPLERATE Sample rate control 8 KHZ
78              */
79     0x0001, /* 9 DIGACT Digital interface activation On
80              */
81     /*
82     *****
83     */
84 };
85
86 // Codec handle:- a variable used to identify audio interface
87 DSK6713_AIC23_CodecHandle H_Codec;
88
89 /***** Function prototypes
90 *****/
91 void init_hardware(void);
92 void init_HWI(void);
93 void ISR_AIC(void);
94 void circ_FIR(); //function that implements the FIR
95 /***** Main routine
96 *****/
97 void main(){
98

```

```

79  if(N+1%2==0)           //middle element calculation for odd and even
    cases
80  {                       //as loop counter limit
81      mid = (N-1)/2;
82  }
83  else
84  {
85      mid = (N-2)/2;
86  }
87
88  // initialize board and the audio port
89  init_hardware();
90
91  /* initialize hardware interrupts */
92  init_HWI();
93
94  /* loop indefinitely, waiting for interrupts */
95  while(1)
96  {};
97
98  }
99
100 /****** init_hardware()
    *****/
101 void init_hardware()
102 {
103     // Initialize the board support library, must be called first
104     DSK6713_init();
105
106     // Start the AIC23 codec using the settings defined above in config
107     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
108
109     /* Function below sets the number of bits in word used by MSBSP (
        serial port) for
110     receives from AIC23 (audio port). We are using a 32 bit packet
        containing two
111     16 bit numbers hence 32BIT is set for receive */
112     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
113
114     /* Configures interrupt to activate on each consecutive available 32
        bits
115     from Audio port hence an interrupt is generated for each L & R sample
        pair */
116     MCBSP_FSETS(SPCR1, RINTM, FRM);
117
118     /* These commands do the same thing as above but applied to data
        transfers to
119     the audio port */
120     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
121     MCBSP_FSETS(SPCR1, XINTM, FRM);
122
123
124 }
125

```

```

126 /***** init_HWI()
      *****/
127 void init_HWI(void)
128 {
129     IRQ_globalDisable();    // Globally disables interrupts
130     IRQ_nmiEnable();        // Enables the NMI interrupt (used by the
        debugger)
131     IRQ_map(IRQ_EVT_RINT1,4); // Maps an event to a physical interrupt
132     IRQ_enable(IRQ_EVT_RINT1); // Enables the event
133     IRQ_globalEnable();      // Globally enables interrupts
134 }
135
136 /***** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE
      *****/
137
138 void ISR_AIC()
139 {
140     sample = mono_read_16Bit(); //takes input sample
141     output=0;                  //resetting output to zero for accumulation
142     circ_FIR();
143     mono_write_16Bit((short)floor(output)); //writing to output
144 }
145
146 void circ_FIR()                //circular buffer implementation of FIR
    with symmetry
147 {
148     int i,j,k;
149     x[ptr] = sample;           //sample input into index ptr
150     i = ptr;                   //index pointer that moves left (to lower
        indices)
151     j = ptr+1;                 //index pointer that moves right (to higher
        indices)
152     for(k=0;k<=mid;k++)        //MAC implementation
153     {
154         if(i==-1)              //conditional statement to check wrap around of
            i
155         {
156             i = N;
157         }
158         if(j==N+1)             //conditional statement to check wrap around
            of j
159         {
160             j = 0;
161         }
162         output += b[k]*(x[i--]+x[j++]);
163     }
164
165     if(j==N+1)                 //conditional statement to check wrap around
            of j
166     {                           //before middle element is processed
167         j=0;
168     }
169     if(N+1%2!=0)               //handling of middle element in case of odd
        number of taps

```

```
170 {
171     output += b[mid+1]*x[j++];
172 }
173 if(ptr==N)                //conditional statement to check wrap around
    of ptr
174 {
175     ptr=0;
176 }
177 else
178 {
179     ptr++;                //incrementation of ptr
180 }
181 }
```

## F Circular FIR, symmetric coefficients with wrap prediction and float precision optimisation

```
1  /*****
2      DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3      IMPERIAL COLLEGE LONDON
4
5      EE 3.19: Real Time Digital Signal Processing
6      Dr Paul Mitcheson and Daniel Harvey
7
8      LAB 3: Interrupt I/O
9
10     **** I N T I O . C ****
11
12     Demonstrates inputing and outputing data from the DSK's audio port
13     using interrupts.
14
15     ****
16     Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
17     Updated for CCS V4 Sept 10
18     *****/
19 /*
20  * You should modify the code so that interrupts are used to service
21  * the
22  * audio port.
23  */
24 /***** Pre-processor statements
25     *****/
26
27 #include <stdlib.h>
28 // Included so program can make use of DSP/BIOS configuration tool.
29 #include "dsp_bios_cfg.h"
30
31 /* The file dsk6713.h must be included in every program that uses the
32    BSL. This
33    example also includes dsk6713_aic23.h because it uses the
34    AIC23 codec module (audio interface). */
35 #include "dsk6713.h"
36 #include "dsk6713_aic23.h"
37
38 // math library (trig functions)
39 #include <math.h>
40
41 // Some functions to help with writing/reading the audio ports when
42 // using interrupts.
43 #include <helper_functions_ISR.h>
44 #include "fir_coef.txt"
45
46 #define N sizeof(b)/sizeof(float)-1 //customisable order of filter
47 float x[N+1]; //input buffer
48 float output,sample; //initialisation of global variables
49 int ptr=0; //circular buffer input pointer
50 int mid; //middle element of input buffer
```

```

46 /***** Global declarations
    *****/
47
48 /* Audio port configuration settings: these values set registers in the
    AIC23 audio
49 interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more
    info. */
50 DSK6713_AIC23_Config Config = { \
51     /*
    *****/
    */
52     /* REGISTER FUNCTION SETTINGS
    */
53     /*
    *****/
    */\
54     0x0017, /* 0 LEFTINVOL Left line input channel volume 0dB
    */\
55     0x0017, /* 1 RIGHTINVOL Right line input channel volume 0dB
    */\
56     0x01f9, /* 2 LEFTHPVOL Left channel headphone volume 0dB
    */\
57     0x01f9, /* 3 RIGHTHPVOL Right channel headphone volume 0dB
    */\
58     0x0011, /* 4 ANAPATH Analog audio path control DAC on,
    Mic boost 20dB*/\
59     0x0000, /* 5 DIGPATH Digital audio path control All
    Filters off */\
60     0x0000, /* 6 DPOWERDOWN Power down control All
    Hardware on */\
61     0x0043, /* 7 DIGIF Digital audio interface format 16 bit
    */\
62     0x008d, /* 8 SAMPLERATE Sample rate control 8 KHZ
    */\
63     0x0001 /* 9 DIGACT Digital interface activation On
    */\
64     /*
    *****/
    */
65 };
66
67
68 // Codec handle:- a variable used to identify audio interface
69 DSK6713_AIC23_CodecHandle H_Codec;
70
71 /***** Function prototypes
    *****/
72 void init_hardware(void);
73 void init_HWI(void);
74 void ISR_AIC(void);
75 void circ_FIR(); //function that implements FIR
76 /***** Main routine
    *****/
77 void main(){

```



```

78
79  if(N+1%2==0)           //middle element calculation for odd and even
    cases
80  {                       //as loop counter limit
81      mid = (N-1)/2;
82  }
83  else
84  {
85      mid = (N-2)/2;
86  }
87  // initialize board and the audio port
88  init_hardware();
89
90  /* initialize hardware interrupts */
91  init_HWI();
92
93  /* loop indefinitely, waiting for interrupts */
94  while(1)
95  {};
96
97  }
98
99  /***** init_hardware()
    *****/
100 void init_hardware()
101 {
102     // Initialize the board support library, must be called first
103     DSK6713_init();
104
105     // Start the AIC23 codec using the settings defined above in config
106     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
107
108     /* Function below sets the number of bits in word used by MSBSP (
        serial port) for
109     receives from AIC23 (audio port). We are using a 32 bit packet
        containing two
110     16 bit numbers hence 32BIT is set for receive */
111     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
112
113     /* Configures interrupt to activate on each consecutive available 32
        bits
114     from Audio port hence an interrupt is generated for each L & R sample
        pair */
115     MCBSP_FSETS(SPCR1, RINTM, FRM);
116
117     /* These commands do the same thing as above but applied to data
        transfers to
118     the audio port */
119     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
120     MCBSP_FSETS(SPCR1, XINTM, FRM);
121
122
123 }
124

```

```

125 /***** init_HWI()
    *****/
126 void init_HWI(void)
127 {
128     IRQ_globalDisable();    // Globally disables interrupts
129     IRQ_nmiEnable();        // Enables the NMI interrupt (used by the
        debugger)
130     IRQ_map(IRQ_EVT_RINT1,4); // Maps an event to a physical interrupt
131     IRQ_enable(IRQ_EVT_RINT1); // Enables the event
132     IRQ_globalEnable();      // Globally enables interrupts
133 }
134
135 /***** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE
    *****/
136
137 void ISR_AIC()
138 {
139     sample = mono_read_16Bit();    //takes input sample
140     output=0;                      //resetting output to zero for accumulation
141     circ_FIR();
142     mono_write_16Bit((short)floor(output)); //writing to output
143 }
144
145 void circ_FIR()    //circular buffer implementation of FIR with
        symmetry and wrap prediction
146 {
147     int i,j,k,wrap,diff;
148     x[ptr] = sample;    //sample input into index ptr
149     i = ptr;            //index pointer that moves left (to lower indices)
150     j = ptr+1;          //index pointer that moves right (to higher
        indices)
151     if((N-j)<=i)        //check to if i wraps around
152     {
153         wrap = 0;        //wrap set according to check
154         diff = N-j;      //difference between ptr and N
155     }
156     else                //check to if j wraps around
157     {
158         wrap = 1;        //wrap set according to check
159         diff = i;        //difference between ptr and 0
160     }
161     if(wrap==0)
162     {
163         if(diff>=mid)    //case of ptr in the middle
164         {
165             for(k=0;k<=mid;k++) //MAC implementation
166             {
167                 output += b[k]*(x[i--]+x[j++]);
168             }
169         }
170         else
171         {
172             for(k=0;k<=diff;k++) //MAC implementation upto wrap around
173             {

```

```

174         output += b[k]*(x[i--]+x[j++]);
175     }
176     j=0;           //wrap around
177     for(k=diff+1;k<=mid;k++) //MAC implementation till middle
178         element
179         {
180             output += b[k]*(x[i--]+x[j++]);
181         }
182     if(N+1%2!=0)    //handling of middle element in case of odd
183         number of taps
184         {
185             output += b[mid+1]*x[i--];
186         }
187     else
188     {
189         if(diff>=mid) //case of ptr in the middle
190         {
191             for(k=0;k<=mid;k++) //MAC implementation
192             {
193                 output += b[k]*(x[i--]+x[j++]);
194             }
195         }
196         else
197         {
198             for(k=0;k<=diff;k++) //MAC implementation upto wrap around
199             {
200                 output += b[k]*(x[i--]+x[j++]);
201             }
202             i=N;           //wrap around
203             for(k=diff+1;k<=mid;k++) //MAC implementation till middle
204             element
205             {
206                 output += b[k]*(x[i--]+x[j++]);
207             }
208             if(N+1%2!=0)
209             {
210                 output += b[mid+1]*x[j++]; //handling of middle element in case
211                 of odd number of taps
212             }
213         }
214     if(ptr==N)        //conditional statement to check wrap around of ptr
215     {
216         ptr=0;
217     }
218     else
219     {
220         ptr++;        //incrementation of ptr
221     }

```

## G Circular FIR, double buffer with float precision optimisation and inlining

```
1  /*****
2      DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
3      IMPERIAL COLLEGE LONDON
4
5      EE 3.19: Real Time Digital Signal Processing
6      Dr Paul Mitcheson and Daniel Harvey
7
8      LAB 3: Interrupt I/O
9
10     **** I N T I O . C ****
11
12     Demonstrates inputing and outputing data from the DSK's audio port
13     using interrupts.
14
15     ****
16     Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
17     Updated for CCS V4 Sept 10
18     *****/
19 /*
20  * You should modify the code so that interrupts are used to service
21  * the
22  * audio port.
23  */
24 /***** Pre-processor statements
25     *****/
26
27 #include <stdlib.h>
28 // Included so program can make use of DSP/BIOS configuration tool.
29 #include "dsp_bios_cfg.h"
30
31 /* The file dsk6713.h must be included in every program that uses the
32    BSL. This
33    example also includes dsk6713_aic23.h because it uses the
34    AIC23 codec module (audio interface). */
35 #include "dsk6713.h"
36 #include "dsk6713_aic23.h"
37
38 // math library (trig functions)
39 #include <math.h>
40
41 // Some functions to help with writing/reading the audio ports when
42 // using interrupts.
43 #include <helper_functions_ISR.h>
44 #include "fir_coef.txt"
45
46 #define N sizeof(b)/sizeof(float)-1 //customisable order of filter
47 float x[2*N+2]; //input buffer
48 int ptr=0; //circular buffer input pointer
49 int mid; //middle element of buffer
50 /***** Global declarations
```

```

46 *****/
47 /* Audio port configuration settings: these values set registers in the
48    AIC23 audio
49    interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more
50    info. */
49 DSK6713_AIC23_Config Config = { \
51     /*
52     *****
53     */
51     /* REGISTER FUNCTION SETTINGS
52     */
52     /*
53     *****
54     */
53     0x0017, /* 0 LEFTINVOL Left line input channel volume 0dB
54              */
54     0x0017, /* 1 RIGHTINVOL Right line input channel volume 0dB
55              */
55     0x01f9, /* 2 LEFTHPVOL Left channel headphone volume 0dB
56              */
56     0x01f9, /* 3 RIGHTHPVOL Right channel headphone volume 0dB
57              */
57     0x0011, /* 4 ANAPATH Analog audio path control DAC on,
58              Mic boost 20dB */
58     0x0000, /* 5 DIGPATH Digital audio path control All
59              Filters off */
59     0x0000, /* 6 DPOWERDOWN Power down control All
60              Hardware on */
60     0x0043, /* 7 DIGIF Digital audio interface format 16 bit
61              */
61     0x008d, /* 8 SAMPLERATE Sample rate control 8 KHZ
62              */
62     0x0001 /* 9 DIGACT Digital interface activation On
63              */
63     /*
64     *****
65     */
64 };
65
66
67 // Codec handle:- a variable used to identify audio interface
68 DSK6713_AIC23_CodecHandle H_Codec;
69
70 /***** Function prototypes
71 *****/
71 void init_hardware(void);
72 void init_HWI(void);
73 void ISR_AIC(void);
74 /***** Main routine
75 *****/
75 void main(){
76
77     if(N+1%2==0) //middle element calculation for odd and even

```

```

    cases
78 {           //as loop counter limit
79     mid = (N-1)/2;
80 }
81 else
82 {
83     mid = (N-2)/2;
84 }
85
86 // initialize board and the audio port
87 init_hardware();
88
89 /* initialize hardware interrupts */
90 init_HWI();
91
92 /* loop indefinitely, waiting for interrupts */
93 while(1)
94 {}
95
96 }
97
98 /***** init_hardware()
99      *****/
100 void init_hardware()
101 {
102     // Initialize the board support library, must be called first
103     DSK6713_init();
104
105     // Start the AIC23 codec using the settings defined above in config
106     H_Codec = DSK6713_AIC23_openCodec(0, &Config);
107
108     /* Function below sets the number of bits in word used by MSBSP (
109        serial port) for
110        receives from AIC23 (audio port). We are using a 32 bit packet
111        containing two
112        16 bit numbers hence 32BIT is set for receive */
113     MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
114
115     /* Configures interrupt to activate on each consecutive available 32
116        bits
117        from Audio port hence an interrupt is generated for each L & R sample
118        pair */
119     MCBSP_FSETS(SPCR1, RINTM, FRM);
120
121     /* These commands do the same thing as above but applied to data
122        transfers to
123        the audio port */
124     MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
125     MCBSP_FSETS(SPCR1, XINTM, FRM);
126
127 }
128
129 /***** init_HWI()

```

```

125 void init_HWI(void)
126 {
127     IRQ_globalDisable();           // Globally disables interrupts
128     IRQ_nmiEnable();               // Enables the NMI interrupt (used by the
    debugger)
129     IRQ_map(IRQ_EVT_RINT1,4);      // Maps an event to a physical interrupt
130     IRQ_enable(IRQ_EVT_RINT1);     // Enables the event
131     IRQ_globalEnable();            // Globally enables interrupts
132 }
133
134 /***** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE
    *****/
135
136 void ISR_AIC()
137 {
138     float output=0;                //resetting output to zero for accumulation
139     int i,j,k;
140     x[ptr] = mono_read_16Bit();     //takes input sample
141     x[ptr+N+1] = x[ptr];            //duplicates input sample
142     i = ptr+1;                      //index pointer that moves right (to higher
    indices)
143     j = ptr+N+1;                    //index pointer that moves left (to lower
    indices)
144     for(k=0;k<=(N-2)/2;k++)         //MAC implementation
145     {
146         output += b[k]*(x[i++]+x[j--]);
147     }
148     if(N+1%2!=0)                    //handling of middle element in case of odd
    number of taps
149     {
150         output += b[k]*x[i];
151     }
152     if(++ptr == (N+1))               //conditional statement to check wrap
    around of ptr
153     {                                //along with incrementation of ptr
154         ptr = 0;
155     }
156     mono_write_16Bit((short)output); //writing to output
157 }

```