

EE3-19 Real Time Digital Signal Processing

Coursework Report 3: Lab 4

Jeremy Chan*

Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
jc4913@ic.ac.uk | 00818433

Chak Yeung Dominic Kwok*

Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
cyk113@ic.ac.uk | 00827832

*These authors contributed equally to this work

Table of Contents

I.	Introduction.....	3
II.	Correct design of the filter in MATLAB with proof of the expected frequency response.....	3
III.	Code Operation	7
A.	Pre-processor statements	7
B.	Global variables/declarations.....	7
C.	Function prototypes.....	7
D.	Initialisation functions.....	8
E.	Main testbench	8
F.	Interrupt Service Routine.....	8
G.	FIR Filter Calculation Functions	9
1)	Shuffling Non-Circular Buffer.....	9
2)	Modification 1: Moving the non circular buffer shift inside the MAC for loop.....	9
3)	Modification 2: Circular Buffer.....	10
4)	Modification 3-4: Linear Phase Properties with non-circular buffer / with circular buffer	11
5)	Modification 5: Optimizing Speed by Memory Trade-off.....	13
6)	Modification 6: Using pointers instead of array indices.....	14
IV.	Benchmarks and Discussion	15
V.	Frequency response of the best implementation (sym_fir_ptr)	17
VI.	Conclusion.....	22
VII.	References.....	22
VIII.	Appendix.....	23
A.	MATLAB Script.....	23
B.	MATLAB fir_coef.txt	23
C.	intio.c Source Code	24
D.	Dr. Paul Mitcheson's email.....	29

I. INTRODUCTION

Modifying lab 3's skeleton code, a FIR filter was realized on hardware given generated filter coefficients using MATLAB. This report details the process of putting such a filter on hardware, and includes discussions on results, benchmarking, and filter responses, measured using an audio analyzer.

Multiple functions were used to generate the FIR filter, and justifications for modifications are given in the discussion on the different implementations.

II. CORRECT DESIGN OF THE FILTER IN MATLAB WITH PROOF OF THE EXPECTED FREQUENCY RESPONSE

A FIR filter was required for this lab – the algorithm to be used in the design of the filter coefficient was the Parks-McClelland (P.M.) algorithm [1]. This algorithm converts input specifications of a filter to coefficients which can be output to a file. The specifications were as follows:

TABLE I.

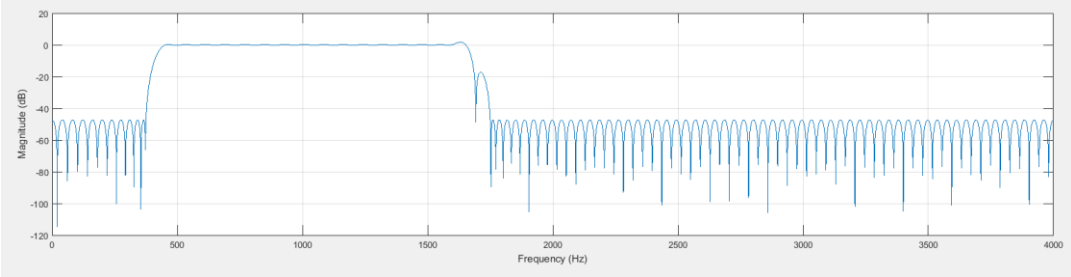
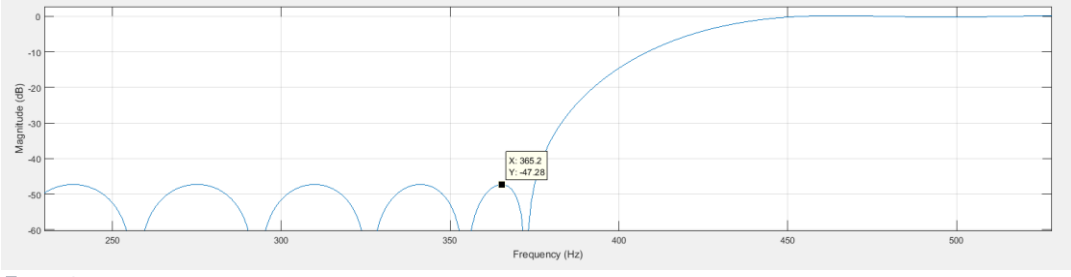
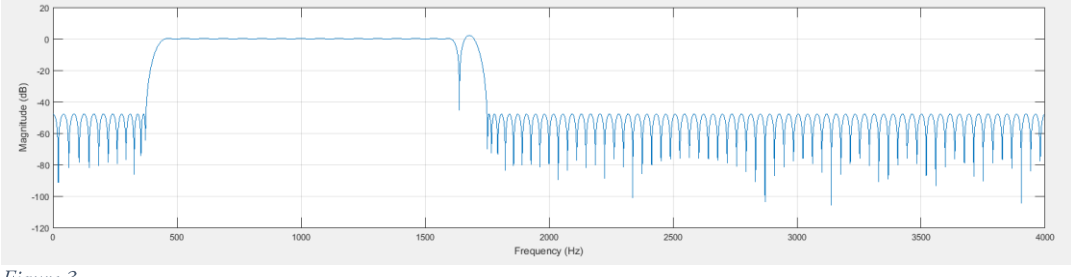
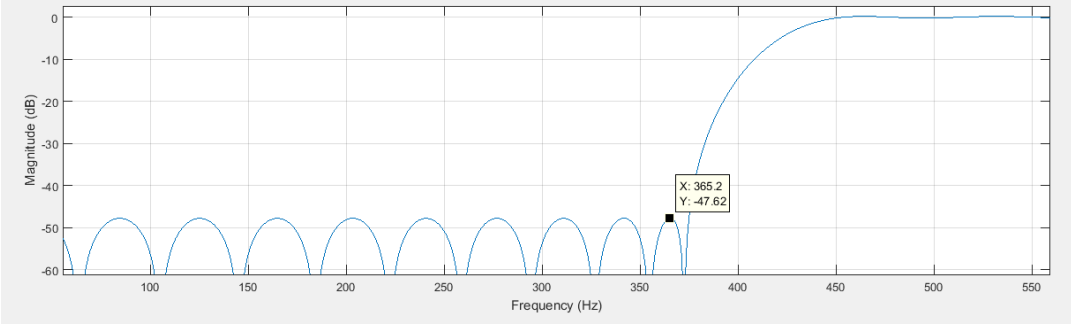
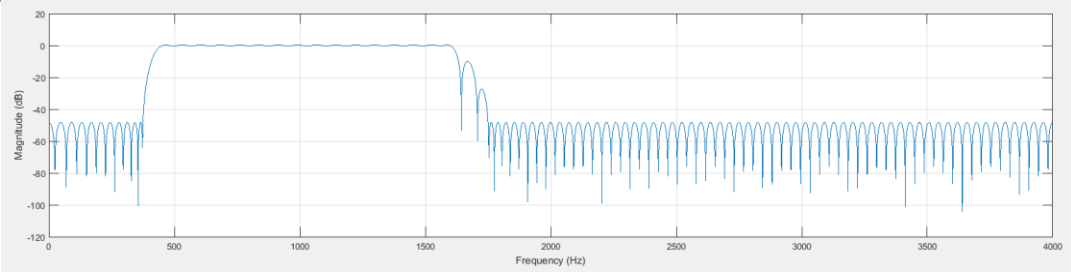
	Value
Sampling Frequency	8000Hz
Pass Band Ripple	0.4dB
Pass Band Cut Off Frequency	450Hz/1600Hz
Min Stop Band Attenuation	-48dB
Stop Band Cut Off Frequency	375Hz/1750Hz

```
F=[375,450,1600,1750];
Fs=8000;
rp=0.4;
A=[0 1 0];
DEV=[10^(-48/20) (10^(rp/20)-1)/(10^(rp/20)+1) 10^(-48/20)];
[N,Fo,Ao,W]=firlpmord(F,A,DEV,Fs);
N=N+6;
B=firlpm(N,Fo,Ao,W);
freqz(B,1,4096,Fs);
fid=fopen('fir_coef.txt','w');
fprintf(fid,'double b[]={');
fprintf(fid,'%10e,\t',B);
fprintf(fid,'};\n');
fprintf(fid,'#define N %d',length(B));
fclose(fid);
```

The above code snippet was used to specify and generate the FIR filter. The output coefficients were written into a text file using `fprintf(fid, '%10e, \t', B);`. This also ensured the correct precision was used for the coefficients. For the code snippet above, MATLAB initially generated a double of size 207 – this is the number of taps of the FIR filter.

However, the initial generated filter does not strictly meet the minimum stop band attenuation specification. The number of taps was then incremented by 2 in Table II (to keep an odd order) until all the specifications were met. 211+ taps successfully met all parts of the specification; our final decision was to choose the numbers of taps to be 213 to allow for some leeway in case of noise or measurement errors in our hardware. The original specification was not changed at all.

TABLE II.

Taps	MATLAB Plot
207 (gain)	<div><p><i>Figure 1</i></p></div>
207 (stop band attenuation too large)	<div><p><i>Figure 2</i></p></div>
209 (gain)	<div><p><i>Figure 3</i></p></div>
209 (stop band attenuation too large)	<div><p><i>Figure 4</i></p></div>
211 (gain)	<div><p><i>Figure 5</i></p></div>

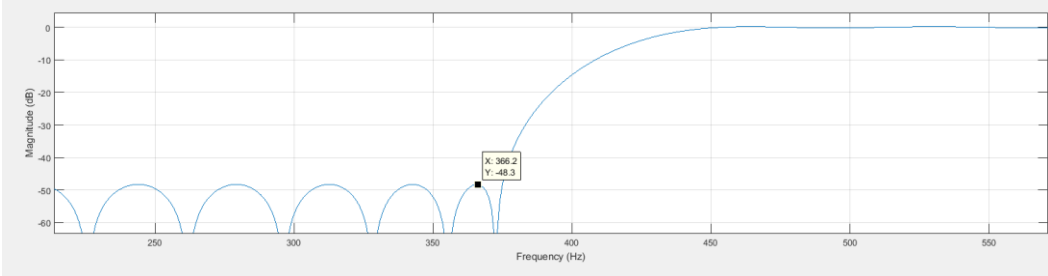
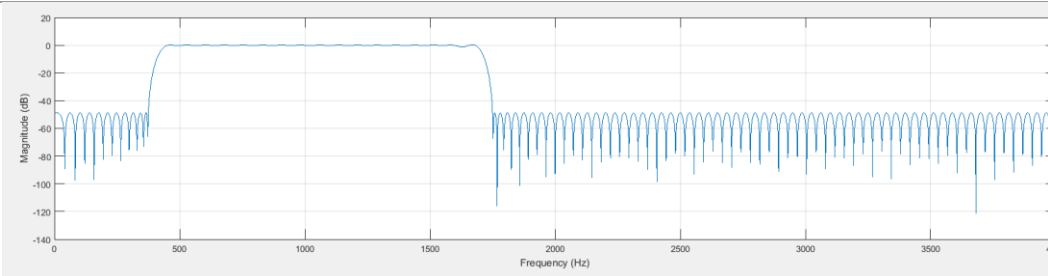
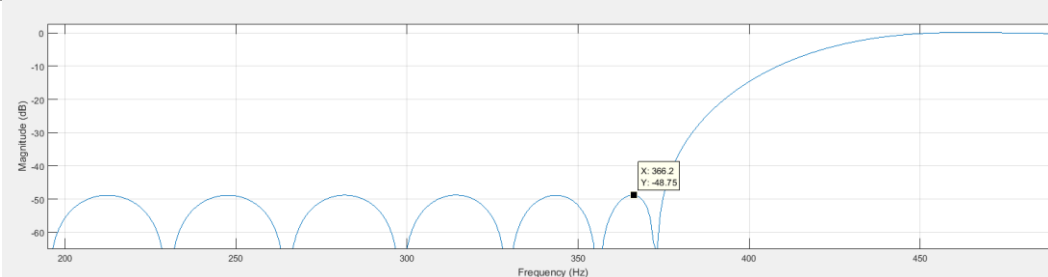
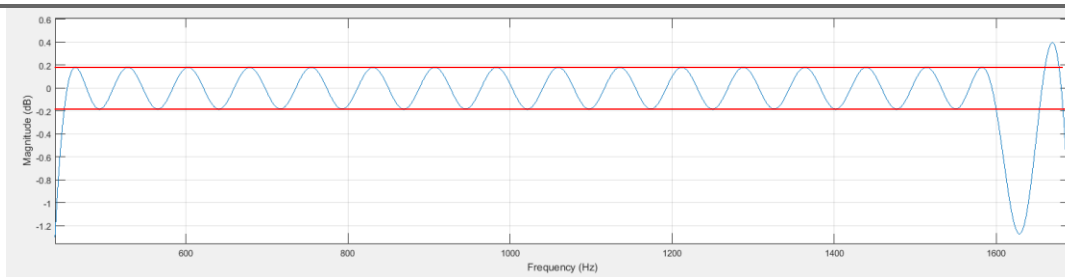
211 (stop band attenuation fits)	 <p>Figure 6</p>
213 (gain)	 <p>Figure 7</p>
213 (stop band attenuation fits)	 <p>Figure 8</p>

Figure 7 and 8 show a FIR filter with 213 taps, conforming to specifications. The shape of the frequency response is according to the required form. The passband and stopband frequencies can be seen as the magnitude rises sharply in the transition period between the two zones.

The following figures zoom into the filter to prove that it matches the other specifications.

	Plot
This figure zooms in into the passband to show that the passband ripple is within the specified 0.4dB.	 <p>Figure 9</p>

This figure shows the magnitude response around the lower cut-off frequency of 450Hz. The magnitude is below -48dB at 375Hz and rises rapidly to 0dB at 450Hz, therefore satisfying the filter specification.

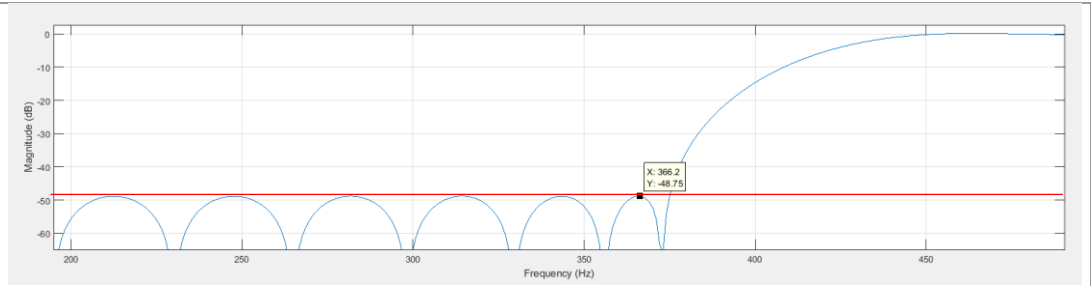


Figure 10

This figure shows the magnitude response around the upper cut-off frequency of 1600Hz. The magnitude is approximately 0dB at 1600Hz and drops below -48dB at 1750Hz, therefore satisfying the filter specification.

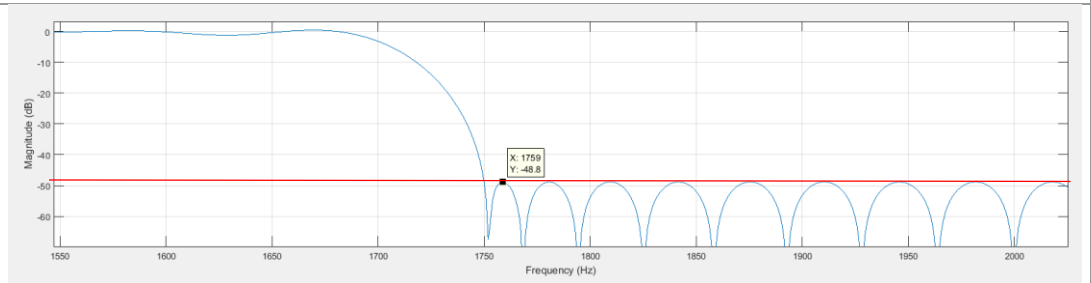


Figure 11

The linear phase response in the passband is as expected from the P.M. algorithm. However, outside the passband, the phase response is flattened and is a periodic saw-tooth waveform. This effect is due to periodic ripples in the stop band.

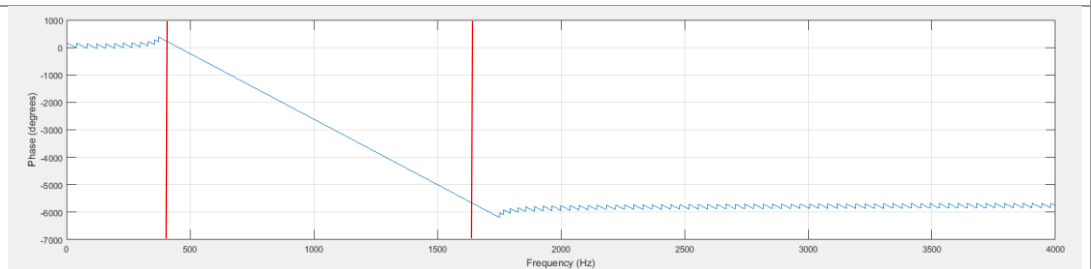


Figure 12

This figure shows the zoomed version on the stop band. It is clear that the phase response is periodic but discontinuous since the ripple is always positive, and hence causes only π phase change in each ripple cycle.

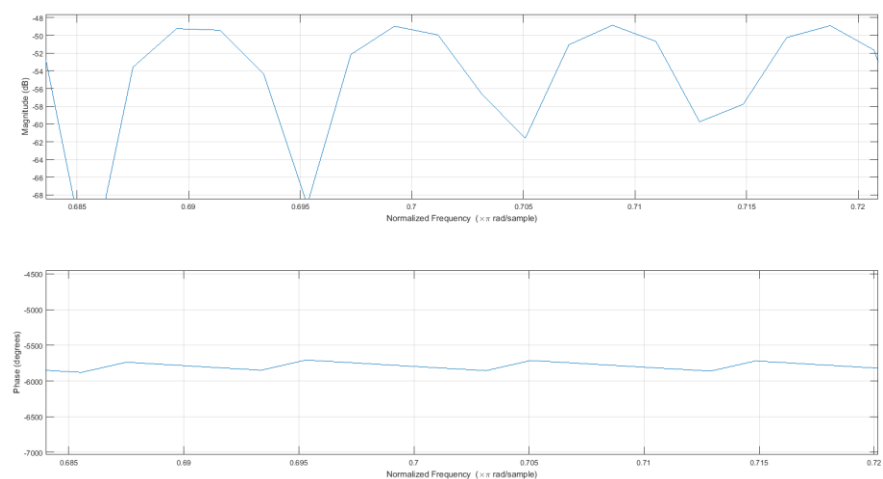


Figure 13

From figure 12, the phase response exhibits a linear relationship with frequency. This is expected as the filter coefficients, computed by the P.M. algorithm has symmetric properties: $b[n] = b[N-1-n]$. Consider the case of an odd number of taps; the proof of linear phase response is proved as following:

$$\begin{aligned}
H(z) &= \sum_{n=0}^{N-1} h(n)z^{-n} \\
&= \sum_{n=0}^{\frac{N-1}{2}-1} h(n)[z^{-n} + z^{-(N-1-n)}] + h\left(\frac{N-1}{2}\right)z^{-\left[\frac{N-1}{2}\right]} \\
\therefore H(e^{j\omega}) &= e^{-j\omega\left[\frac{N-1}{2}\right]} \left\{ h\left(\frac{N-1}{2}\right) + \sum_{n=0}^{\frac{N-3}{2}} 2h(n) \cos\left[\omega\left(n - \frac{N-1}{2}\right)\right] \right\} \\
\therefore \angle(H(e^{j\omega})) &= -\omega\left(\frac{N-1}{2}\right) \\
\frac{d\theta}{d\omega} &= -\left(\frac{N-1}{2}\right) = \text{constant}, \text{ where } \theta \text{ is angle}
\end{aligned}$$

Hence, the gradient of the phase response is constant; it is linear phase.

III. CODE OPERATION

A. Pre-processor statements

Libraries containing helper functions such as `helper_functions_ISR.h` for read/write when using interrupts are defined at the top of the code with `#include <file>`. Other global constants are also defined in this section, e.g. `N`, with `#define <text> <value>`. The pre-processor will look at these statements and replace the statements with their contents into the source file to produce an expanded source code file, before handing it over to the compiler for compilation.

```
#include <../coef/fir_coef_213.txt>
#define N 213

#if (N%2)
#define mid_pt ((N-1)/2)
#else
#define mid_pt (N/2)
#endif
```

Other pre-processor variables defined include the size of the delay line, i.e. number of taps of the filter. An if-else pre-processor statement is used to calculate the midpoint for use in modifications defined below, to take into account both cases of odd/even taps.

B. Global variables/declarations

Instantiations of structs defined in library files can be found below the pre-processor statements – their aim is to define a set of configurations to be called in later functions, whether this is to initialize hardware or specify new data structures. For example, the codec config is defined here, and is called in `init_hardware()`, as the configuration used by the codec (`H_Codec = DSK6713_AIC23_openCodec(0, &Config);`).

C. Function prototypes

```
void init_hardware(void);
void init_HWI(void);

void interrupt_service_routine(void);
double non_cir_FIR(double sample_in);
double cir_FIR(double sample_in);
```

```
double linear_cir_FIR(double sample_in);
double sym_fir(double sample);
double non_cir_FIR_sym(double sample_in);
double non_cir_FIR_2(double sample_in);
double sym_fir_ptr(double sample_in);
```

Function prototypes are declared before `int main()` for the compiler to be able to find the FIR functions. Our functions are defined after the interrupt function which calls them.

D. Initialisation functions

`void init_hardware()` configures the hardware using defined global variables. The struct of the config for configuring the hardware can be found in the included header files; the instantiation in the global declarations section. This function is duly executed as the first line of `main` before any testbench code is executed.

`void init_HWI()` configures hardware interrupts; it disables all interrupts, assigns events to interrupts, then enables all interrupts again.

E. Main testbench

As this code functions on interrupts, there is no testbench code to execute apart from the initialization of hardware. After initialization, the program enters a `while(1)` infinite loop, doing nothing, waiting for interrupts.

When a hardware interrupt is encountered, the program runs a pre-defined function based on the function defined in `dsp_bios_.tcf`. In our case, it is `void interrupt_service_routine(void)`;

F. Interrupt Service Routine

The function that runs when a specific hardware interrupt is encountered is defined in the `dsp_bios_.tcf` file (Figure 11).

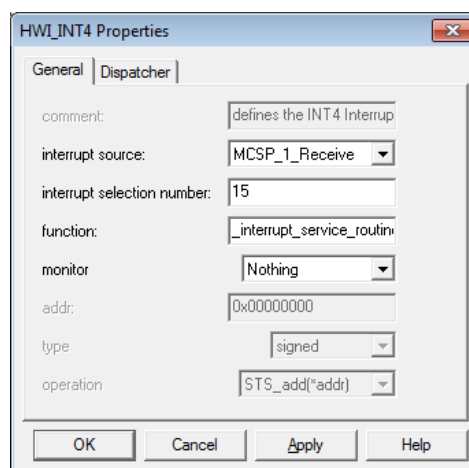


Figure 14: ISR Function Definition

```
/**
 * This function gets executed upon a hardware interrupt. It reads in a sample,
 * then outputs the output of one the FIR functions.
 */
void interrupt_service_routine(void) {
    double sum = 0.0;
    double sample = 0.0;
    sample = (double)mono_read_16Bit();
    //all pass if needed
    //mono_write_16Bit((Int16)(sample*4));
    //FIR FUNCTIONS BELOW
    //sum = non_cir_FIR(sample);
    //sum = cir_FIR(sample);
    //sum = linear_cir_FIR(sample);
```



```

        //sum = sym_fir(sample);
        //sum = non_cir_FIR_sym(sample);
        //sum = non_cir_FIR_2(sample);
        sum = sym_fir_ptr(sample);
        mono_write_16Bit((Int16)(sum * 2));
    }

```

The interrupt function merely reads in a sample and outputs the output of one of the MAC functions to both channels, multiplying by 2 to reduce the passband gain offset.

G. FIR Filter Calculation Functions

1) Shuffling Non-Circular Buffer

```

/**
 * Naive non circular FIR implementation. Uses a for loop to implement delay
 * line.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return          [64 bit MAC value]
 */
double non_cir_FIR(double sample_in){
    int i, j; //define variables for use in loops
    double sum=0;
    for (i=N-1; i>0; i--){
        x[i] = x[i-1]; //array shuffling method as described in handout
    }
    x[0] = sample_in; //read in newest sample to one end of the array

    for (j=0; j<N; j++){
        sum += b[j]*x[j]; //MAC
    }
    return sum;
}

```

The basic non-circular buffer implementation of the FIR filter uses an array shuffling method to implement the delay operator. The function defines variables *i* and *j* to be used in the for loop – they are used as iterators through vector *x* and *b* respectively, where vector *b* is the array storing all the generated FIR coefficients from MATLAB. When a new sample arrives, i.e. when the function is called, the existing samples are moved by one array index (Table III). This shift is implemented using the first for loop in the code snippet above. Since the buffer is of a finite size, the oldest sample will be lost.

Shifting all samples is costly – it takes *N* iterations, where *N* is the number of taps of the filter, and hence causes *N* more computations per ISR. Finally, the newest sample is written to the now empty first index of the array *x* and the MAC result, stored in the variable *sum*, is computed.

TABLE III.

Cycle	X[0]	X[1]	X[2]	...	X[N-1]
0	0	0	0	...	0
1	sample_in_0	0	0	...	0
2	sample_in_1	sample_in_0	0	...	0
3	sample_in_2	sample_in_1	sample_in_0	...	0
...

2) Modification 1: Moving the non circular buffer shift inside the MAC for loop

```

/**
 * This version of the non circular buffer moves the shift for loop inside the
 * MAC for loop. This version is, surprisingly, quicker than the previous
 * circular buffer linear phase functions as the compiler parallizes both the
 * shifting and the MAC. God bless Texas Instruments.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return          [64 bit MAC value]
 */
double non_cir_FIR_2(double sample_in){

```

```

    int i;
    double sum=b[N-1]*x[N-1];           //might as well initialize to a value not
    //calculated by for loop as the for loop iteration size for delay and MAC differ by 1
    x[0] = sample_in;                   //read input sample
    for (i=N-2; i>=0; i--){             //size of MAC loop is larger than delay
loop by 1
        sum+=b[i]*x[i];
        x[i+1] = x[i];                 //change from i>i-1 to i+1>i, as MAC needs
to index b[0] and x[0], and x[-1] is undefined
    }
    return sum;
}

```

Instead of having one for loop to implement the delay line and one to calculate the MAC, they are merged into one for loop. However, the size of the delay line's loop is one smaller than the MAC loop, so the first iteration of the MAC is moved into the initialization of the sum variable. A con of this implementation of merging the for loops is that the symmetry of linear phase filters (loop iteration/2) now cannot be implemented.

3) Modification 2: Circular Buffer

The first improvement to our non-circular buffer FIR filter is to make the buffer 'circular', as it can save N computations per function call (from the delay line shuffling of the array). A circular buffer refers to a buffer whose start and end act as if they are connected together. Instead of the newest sample written to the first element of the delay line - the position of the oldest sample, a circular buffer writes the incoming sample to a changing array index each time. In the case of Figure 15, the position of the oldest element, i.e. the position to write to, increases each time.

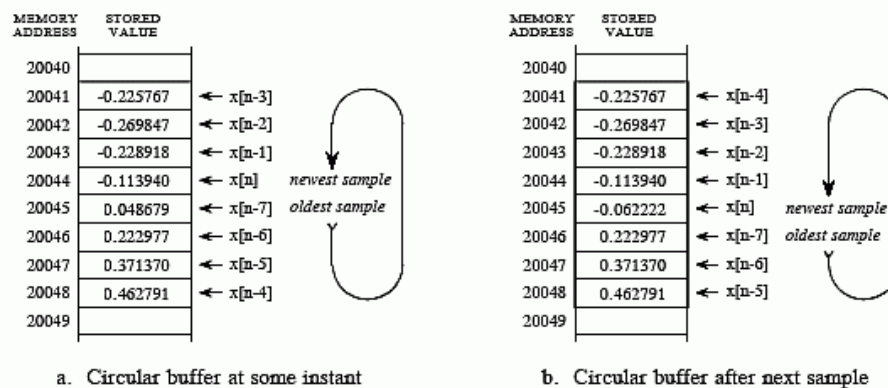


FIGURE 28-3 Circular buffer operation. Circular buffers are used to store the most recent values of a continually updated signal. This illustration shows how an eight sample circular buffer might appear at some instant in time (a), and how it would appear one sample later (b).

Figure 15 [2]

However, since C has no such data structure [3] in-built, the only way to implement this circular buffer is to 'fake' it in software (there is hardware support [4], but we are not writing assembly code). Creating an entire circular buffer data structure can lead to prohibitive overhead when doing real time digital signal processing [5]. Therefore, our improvements contains only the logic needed to handle the wrap-around of the buffer, i.e. when the array index is larger than the array size and needs to be looped back to the start.

```

/**
 * Implements circular buffer FIR filter. Read/write index used to keep track of
 * oldest sample and MAC iteration. If statements used to handle wraparound.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return [64 bit MAC value]
 */

```

```

double cir_FIR(double sample_in){
    int j, read_index;           //define variables for use in loops
    double sum=0.0;
    static int write_index=N-1;  //use a static variable to prevent
    reinitialization between function calls, write index is the pointer to the newest sample
    in the vector x
    x[write_index] = sample_in;  //read the newest sample in at the pointer
    write_index
    for (j=0; j<N; j++){
        read_index=(write_index+j>N-1)?write_index+j-N:write_index+j; //read
        index starts from write index (j=0 at first run) and iterates through the array,
        wrapping around (circular buffer) when the iterator is larger than the size of array x
        sum += b[j]*x[read_index]; //MAC
    }
    if(--write_index== -1)write_index=N-1; //update write_index to oldest sample for
    overwriting on next function call, and check for wraparound of write_index - circular
    buffering
    return sum;
}

```

The main components to our circular buffer are: a pointer that indicates start of the buffer memory (`write_index`), a pointer indicating the end of the buffer memory or a variable indicating size (`N`), and the step size of the memory addressing (in our case one array index holds one sample as it is 64-bit, so one). The circular buffer method did not use the in-built C modulus function (%) as division is costly in terms of computation time.

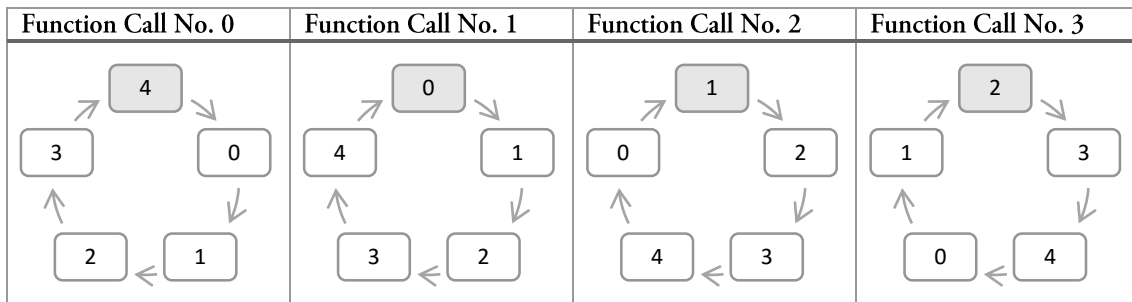
The array index with the oldest sample is also kept track of (`static int write_index`), as this position is where the newest sample should be inserted. Another iterator, `read_index`, is responsible for looping through the array during the computation of the filter using MAC's. If statements are used to handle wraparound of `read_index` and `write_index`. Table IV shows this circular buffer in use at some instance in time.

TABLE IV.

Cycle	X[0]	X[1]	X[2]	...	X[N-1]
i	x[n-(N-1)]	x[n]=sample_in	x[n-1]
i+1	x[n]=sample_in	x[n-1]	x[n-2]
i+2	x[n-1]	x[n-2]	x[n-3]	...	x[n]=sample_in
...

Table V shows the relationship between the `write_index` (decrements by 1 every function call), and the for loop which calculates the convolution. The grey cell represents the starting point of the for loop and hence of the MAC, and is also where the newest sample is written. The MAC runs for the entirety of the buffer using the for loop (5 iterations in the case of the diagrams), and wraps around when the current index is larger than the array size. For the first function call, the newest sample is written to the last cell of the array, then the MAC iterates through array cells 4, then 0, then 1, then 2, then 3.

TABLE V.



4) Modification 3-4: Linear Phase Properties with non-circular buffer / with circular buffer

Since coefficients are mirrored in linear phase FIR filters [6], i.e. $b(n) = b(N-1-n)$, optimizations can be made to the loop to cut the number of iterations in half, i.e., from N iterations to $N/2$ iterations. The algorithm starts by writing the incoming sample to the array at the aforementioned `write_index`, then executes the MAC inwards, towards the centre of the array. Table VI shows this inwards movement for an array size of 5 – the grey cells are the cells which the MAC is currently operating on.

TABLE VI.

Array Index	0	1	2	3	4
Iteration 1	X				X
Iteration 2		X		X	
Iteration 3			X		

```

/**
 * Original non-circular approach, with symmetrical properties.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return [64 bit MAC value]
 */
double non_cir_FIR_sym(double sample_in){
    int i, j;
    double sum=0;
    for (i=N-1; i>0; i--){
        x[i] = x[i-1]; //array shuffling
    }
    x[0] = sample_in;
    sum=b[mid_pt]*x[mid_pt]; //compute midpoint MAC first as it only
requires one coefficient
    for (j=0; j<mid_pt; j++){ //N/2 iterations
        sum += b[j]*(x[j]+x[N-1-j]); //rest of MAC
    }
    return sum;
}

```

The operation of the non-circular buffer uses the same array shuffling method as section III.F.1, but the number of iterations in the MAC loop is cut down in half. This is done by pre-adding the symmetric coefficients pairs ($x[j]$, $x[N-1-j]$) per iteration of the MAC. This function is only valid for odd number of taps of the filter, as the variable `sum` is initialized to the midpoint dot product.

```

/**
 * Exploits linear phase filter properties (symmetrical) to cut down half of MAC
 * iterations. Uses a circular buffer.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return [64 bit MAC value]
 */
double linear_cir_FIR(double sample_in){
    int j, read_index, i, read_index2;
    double sum;
    static int write_index=N-1; //use static to keep value of write_index
between function calls
    i=N-1; //last array index
    x[write_index] = sample_in; //write incoming sample to current value of
write_index
    read_index=(write_index+mid_pt>N-1)?write_index+mid_pt-N:write_index+mid_pt;
//handle circular buffering wraparounds,
and acts as start of MAC
    sum=b[mid_pt]*x[read_index]; //execute MAC on midpoint, as there is only
one point to convolve, instead of two, so do outside for loop
    for (j=0; j<mid_pt; j++){ //calculate FIR filter inwards -
read_index2 is the mirror of read_index

```

```

        read_index=(write_index+j>N-1)?write_index+j-N:write_index+j;
        read_index2=((read_index+i)>N-1)?read_index+i-N:read_index+i;
        sum += b[j]*(x[read_index]+x[read_index2]);
        i=i-2;                                     //as calculating inwards, move both
    iterators
    }
    if(--write_index==N-1)write_index=N-1; //update write_index to oldest sample for
    overwriting on next function call, and check for wraparound of write_index - circular
    buffering
    return sum;
}

```

The operation of the circular buffer implementation is shown in Table VII (using an array size of 5). The MAC converges into the mid-point of the array (grey, different with each function call as it is a circular buffer) with every iteration (1st: blue, 2nd: yellow). Since it is a circular buffer, the newest sample is written to different positions each time (green). Logic is included to handle the wrap-around of all indices.

TABLE VII.

Function Call No. 0	Function Call No. 1	Function Call No. 2	Function Call No. 3
Green: Position of new sample Grey: Position of 'mid-point' of MAC		Blue: First iteration of MAC Yellow: Second iteration of MAC	

5) Modification 5: Optimizing Speed by Memory Trade-off

```

/**
 * Doubles the size of the buffer used to implement the delay line to remove
 * need for if statements checking for wraparound in the MAC loop as it will
 * never overflow.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return [64 bit MAC value]
 */
double sym_fir(double sample){
    static int new_index=N-1;
    int old_index,iterator;
    double sum;
    old_index=new_index+N-1; //index the extended buffer by adding array
    size
    x_extended[new_index]=sample; //copy sample in
    x_extended[new_index+N]=sample; //to both sections of the array
    sum=b[mid_pt]*x_extended[new_index+mid_pt]; //calculate midpoint MAC first as it
    only needs one value
    for (iterator=0;iterator<mid_pt;iterator++){ //loop iterations decreased by half
        sum+=b[iterator]*(x_extended[new_index+iterator]+x_extended[old_index--
    ]); //calculate MAC using both sections of the
    array
    }
    if (--new_index<0)new_index=N-1; //update oldest sample index
    return sum;
}

```

An optimization to the previous linear phase FIR filter is one that trades off memory usage for speed. The memory used is double as the buffer is extended to twice the normal size to create a back to back copy of the buffer [7]. This way, when the original implementation's array index would overflow, it merely

‘overflows’ to the copied array – logic can be put in to index the copied array (just add the size of the array!). This removes the need for overflow checks for both iterators, saving on instructions per ISR.

Of course, this also means that when writing the new sample to the array, the sample needs to be written to the extended array as well (at `current write index + array size`). Table IX shows an example implementation on an array size of 5, hence a delay line size of 10. Grey X’s show the cell is currently being used for the MAC operation. Yellow cells show the original size of the buffer (our MAC was configured to start from the end of the extended array). Note that iterators will always be contained within the doubled buffer.

TABLE VIII.

N=5	Array Index									
Function call/Iteration	0	1	2	3	4	0+N	1+N	2+N	3+N	4+N
1/1						X				X
1/2							X		X	
1/3								X		
2/1					X				X	
2/2						X		X		
2/3							X			
3/1				X				X		
3/2					X		X			
3/3						X				
4/1			X				X			
4/2				X		X				
4/3					X					
...

6) Modification 6: Using pointers instead of array indices

```

/**
 * Explots linear phase properties, and uses a double delay line for speed up.
 * Also replace array indices with const double pointers. Inward movements of
 * the points calculate the MAC, with the midpoint being calculated outside the
 * loop.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return [64 bit MAC value]
 */
double sym_fir_ptr(double sample_in) {
    int i;
    static int index = N-1;           //starting write index for new sample
    double sum = 0.0;
    const double *ptr_b = b, *ptr_x = x_extended + index, *ptr_x2 = x_extended +
index + N - 1;           //use const double pointer to index the array instead of array
indices
    //const double means the pointed to data is constant, but the pointer can change
    //ptr_b points to starting address of array b

    x_extended[index] = x_extended[index + N] = sample_in;           //write incoming
sample to both array and extended array

    for (i = 0; i < mid_pt; i++) {
        sum += *ptr_b++ * (*ptr_x++ + *ptr_x2--);           //compute MAC by moving
pointers inwards
    }
    sum += *ptr_b++ * *ptr_x++;           //compute midpoint, points are conveniently in the
right place after for loop

    if (--index < 0) index += N;           //handle write index overflow
    return sum;
}

```

From the previous implementation using symmetrical properties and an extended buffer, another improvement was made to convert array indices into pointer form. Three points, using the form `const double <pointer> [8]`, are pointed to the start of the coefficient array, delay line array, and extended delay line array. The points move inwards inside the MAC for loop, with the midpoint calculation being done when the for loop finishes executing, i.e., the pointers are at the midpoint. The operation is identical to that of Table VIII.

IV. BENCHMARKS AND DISCUSSION

Benchmarks were done using breakpoints (one breakpoint either side of the function to be benchmarked in the ISR function) in Code Composer Studio v4. Results were collected for three optimization levels, found in the compiler basic options: none, -o0, and -o2. The results are shown in Table IX, and the corresponding line graph in Figure 16.

TABLE IX.

Latency (clock cycles)	Optimization Level		
	None	-o0	-o2
non_cir_FIR	13084	10716	1068
cir_fir	14400	10844	2355
linear_cir	10413	7568	2441
sym_fir	5862	5111	899
non_cir_FIR_sym	10694	7923	1787
non_cir_FIR_2	10327	8391	860
sym_fir_ptr	5779	4060	712

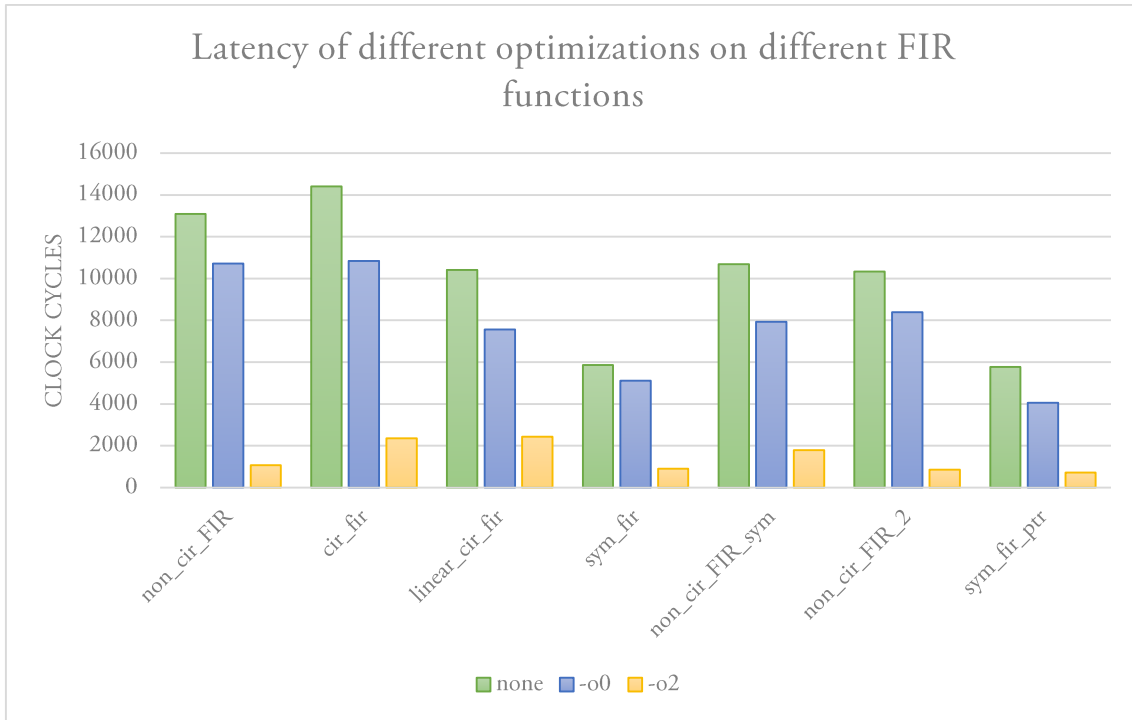


Figure 16

From the results, the best performing function is `sym_fir_ptr`, the symmetrical double delay line implementation using pointers. However, `non_cir_FIR_2`, the non-circular implementation with one for loop to do both shifting and the MAC, was not far behind. This is quite interesting as this means the Texas

Instruments compiler found a way to optimize the shifting and MAC instructions so much so that it is almost as fast as the linear phase circular buffer methods.

3.1 Invoking Optimization

The C/C++ compiler is able to perform various optimizations. High-level optimizations are performed in the optimizer and low-level, target-specific optimizations occur in the code generator. Use high-level optimization levels, such as `--opt_level=2` and `--opt_level=3`, to achieve optimal code.

The easiest way to invoke optimization is to use the compiler program, specifying the `--opt_level=n` option on the compiler command line. You can use `-On` to alias the `--opt_level` option. The n denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization.

- **--opt_level=0 or -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
- **--opt_level=1 or -O1**

Performs all `--opt_level=0` (-O0) optimizations, plus:

 - Performs local copy/constant propagation
 - Removes unused assignments
 - Eliminates local common expressions
- **--opt_level=2 or -O2**

Performs all `--opt_level=1` (-O1) optimizations, plus:

 - Performs software pipelining (see [Section 3.2](#))
 - Performs loop optimizations
 - Eliminates global common subexpressions
 - Eliminates global unused assignments
 - Converts array references in loops to incremented pointer form
 - Performs loop unrolling

The optimizer uses `--opt_level=2` (-O2) as the default if you use `--opt_level` (-O) without an optimization level.

Figure 17 [9]

With no optimizations, the compiler translates every line of C to its equivalent in ASM. Hence, our results with no optimizations are mostly as expected, with the shuffling buffer taking a much longer time than the symmetrical implementations (loop iteration/2). Surprisingly, our circular buffer implementation actually took longer than the shuffling buffer. This is due to the added if statements, used to control the two iterators, in `cir_fir()`; both `non_cir_fir()` and `cir_fir()` have the same length of MAC loop. From the improvements using linear phase properties, `linear_cir_fir()` had a visible improvement over `cir_fir()`. This is due to the number of MAC loop iterations being cut in half. Additionally, `sym_fir_ptr()` had the shortest latency, as it both uses linear phase properties, as well as removing the iterator overflow if statements for extra memory usage. Finally, the non-circular buffer implementation of the symmetrical FIR filter (`non_cir_FIR_sym`) had a longer latency than `sym_fir_ptr()` because of the shuffling buffer requiring N iterations per ISR, giving a total of $N+N/2$ iterations per ISR.

TABLE X.

Latency Improvement	Optimization Level		
	None	-o0	-o2
Function			
non_cir_FIR	0.00%	18.10%	91.84%
cir_fir	0.00%	24.69%	83.65%
linear_cir_fir	0.00%	27.32%	76.56%
sym_fir	0.00%	12.81%	84.66%
non_cir_FIR_sym	0.00%	25.91%	83.29%
non_cir_FIR_2	0.00%	18.75%	91.67%
sym_fir_ptr	0.00%	29.75%	87.68%

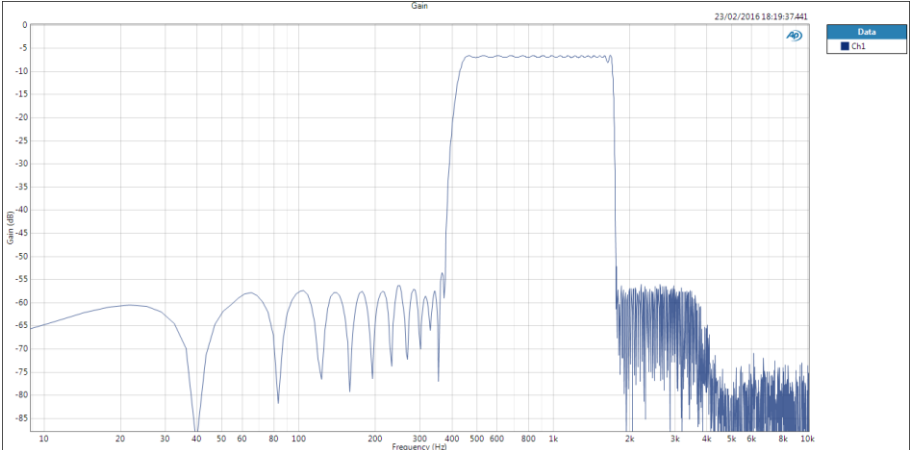
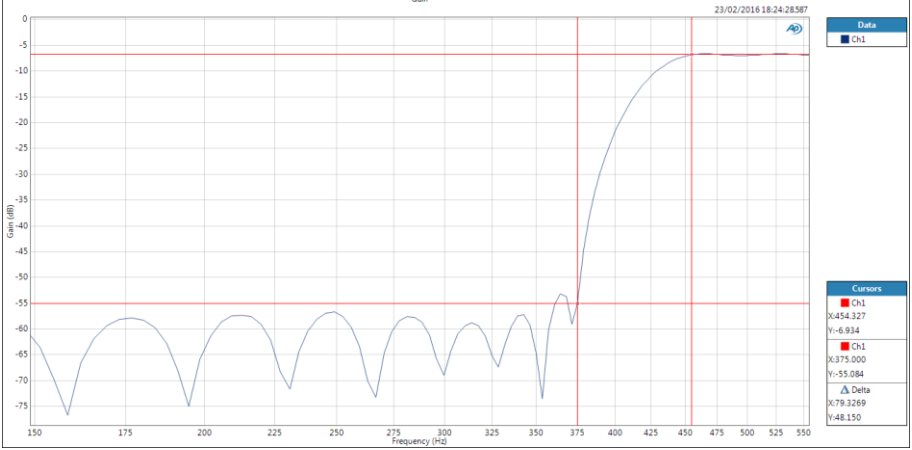
With optimization `-o0`, the compiler performs optimizations on the code according to figure 17. Table X shows the percentage change in instruction clock cycles, before and after optimization. Large

improvements came from both `sym_fir_ptr()` and `linear_cir()`, as the compiler performed control-flow-graph simplification, i.e., optimized the `if` statements controlling the two iterators.

With optimization `-O2`, the compiler performs additional optimizations on the code, according to Figure 17. The most significant improvement came from the loop optimizations (max 91.84% improvement in instruction clock cycles). This can be seen from the improvement on the non-circular implementations, with `non_cir_FIR_2` outperforming almost all other, even the non-pointer version of the symmetrical implementations. Another loop optimization, loop unrolling, was used to replace the `for` loop with copies of the instructions inside the loop. This enables the compiler to use the last significant optimization, software pipelining. As the hardware is capable of doing two MAC's per clock cycle, the linear phase implementations can be pipelined (especially incrementing/decrementing pointers for the MAC), leading to improved performance (the fastest implementation, `sym_fir_ptr()`, had a 87.68% decrease in instruction cycles).

V. FREQUENCY RESPONSE OF THE BEST IMPLEMENTATION (SYM_FIR_PTR)

TABLE XI.

Specification	Plot
General Continuous Sweep Waveform	 <p>Figure 18</p>
Showing difference of >48dB between stopband and passband at low frequencies, hence stopband attenuation is <-48dB	 <p>Figure 19</p>

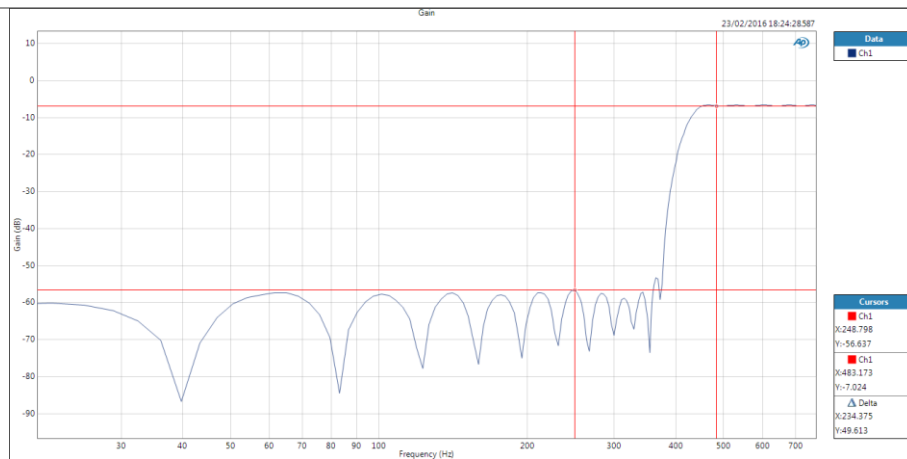


Figure 20

Showing difference of $>48\text{dB}$ between stopband and passband at high frequencies, hence stopband attenuation is $<48\text{dB}$

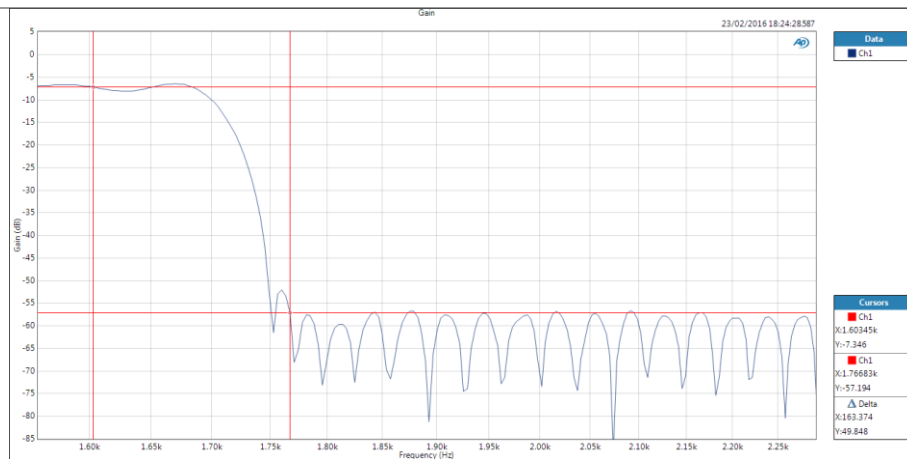


Figure 21

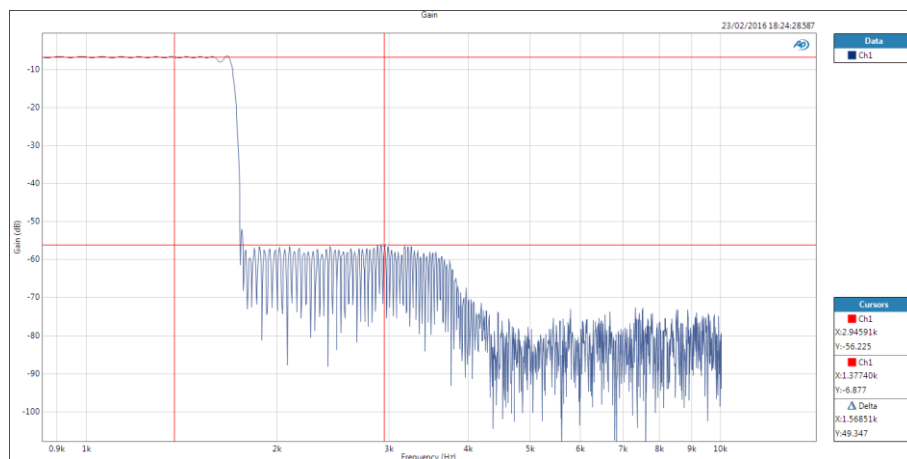


Figure 22

Showing passband ripple is within 0.4dB

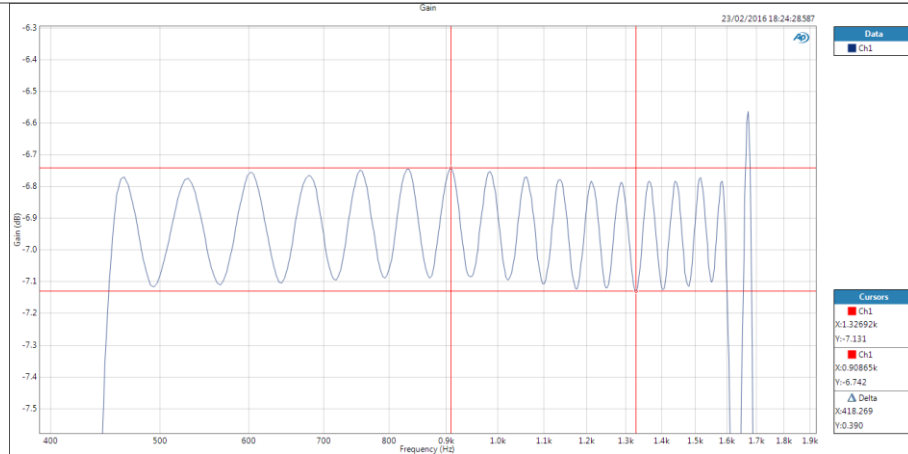


Figure 23

Phase plot in passband, showing linear phase

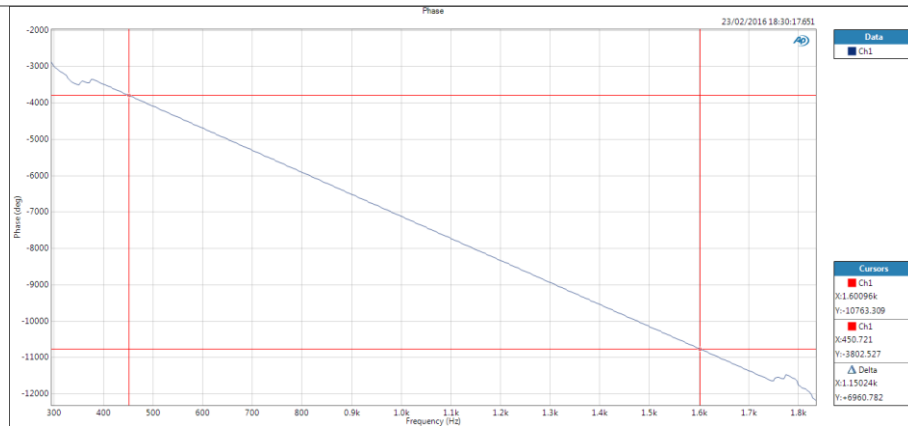


Figure 24

Figure 19 and 20 show a small lobe peaking above the required -48dB in the stopband, at approximately 360Hz . Referring back to Figure 10, this lobe does not appear in the MATLAB plots, and should theoretically not appear on the frequency response plots of the audio analyzer as MATLAB and our FIR implementation in C run the same MAC calculation, on the same set of coefficients. After discussion with Dr. Paul Mitcheson (paul.mitcheson@imperial.ac.uk) (Appendix.D), this small error can be attributed to “noise”, given a “great[er] attenuation in the stop band”. Hence, our conclusion is that this small discrepancy is due to “measurement errors on the APX units”.

Additionally, an interesting point to note is that the passband gain is less than 1. From Figure 25, signals are attenuated to 0.25 of their original value (-12dB). However, in CCS, our output sample is multiplied by a gain of 2, so the gain on the APX graphs are approximately centred on -6dB .

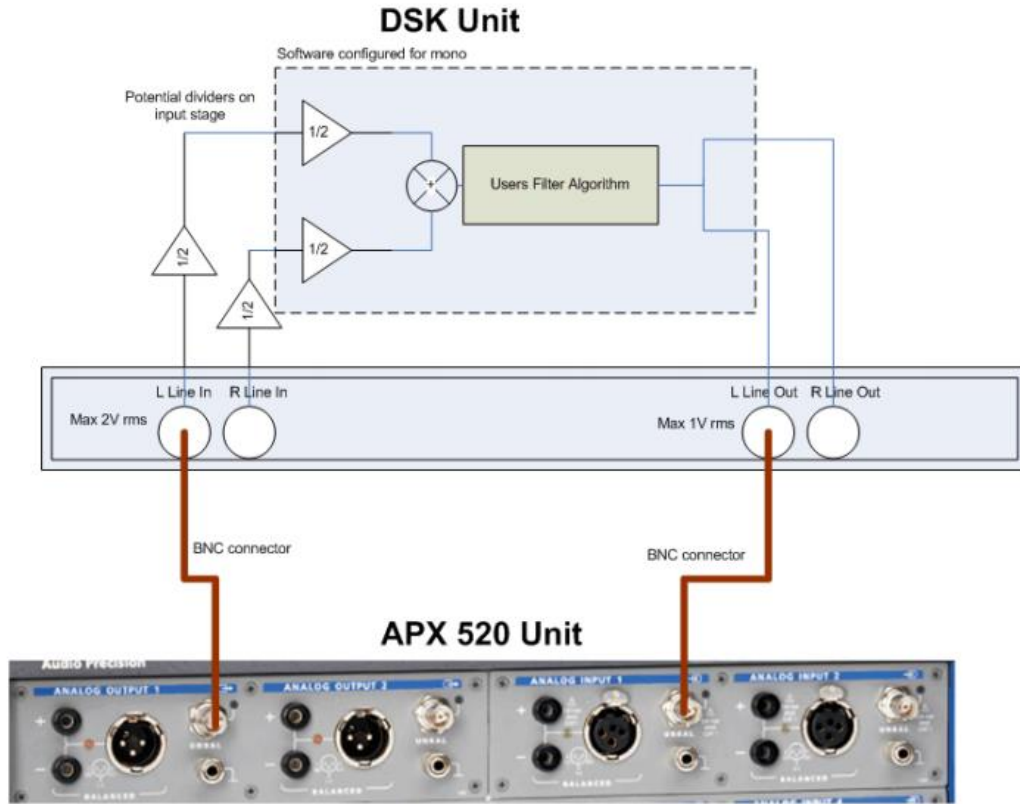


Figure 25

In addition to passband gain difference, the phase response of the actual filter was different from the theoretical specification from MATLAB. The slope of the actual phase response is greater than that in MATLAB. The gradient of the phase response on the filter is also known as negative group delay. From theory, the group delay of a linear phase filter is $t_g = -\frac{d\theta}{f_s} = \frac{N-1}{2}T_s$, where θ is phase. From MATLAB's Figure 26,

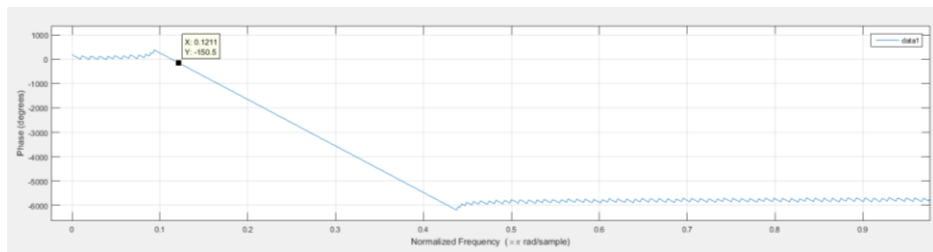
$$t_g = \frac{\delta \text{ deg}}{f_{\text{norm}}} * \frac{1}{180} * \frac{1}{T_s} = 13.2746 \text{ ms.}$$


Figure 26

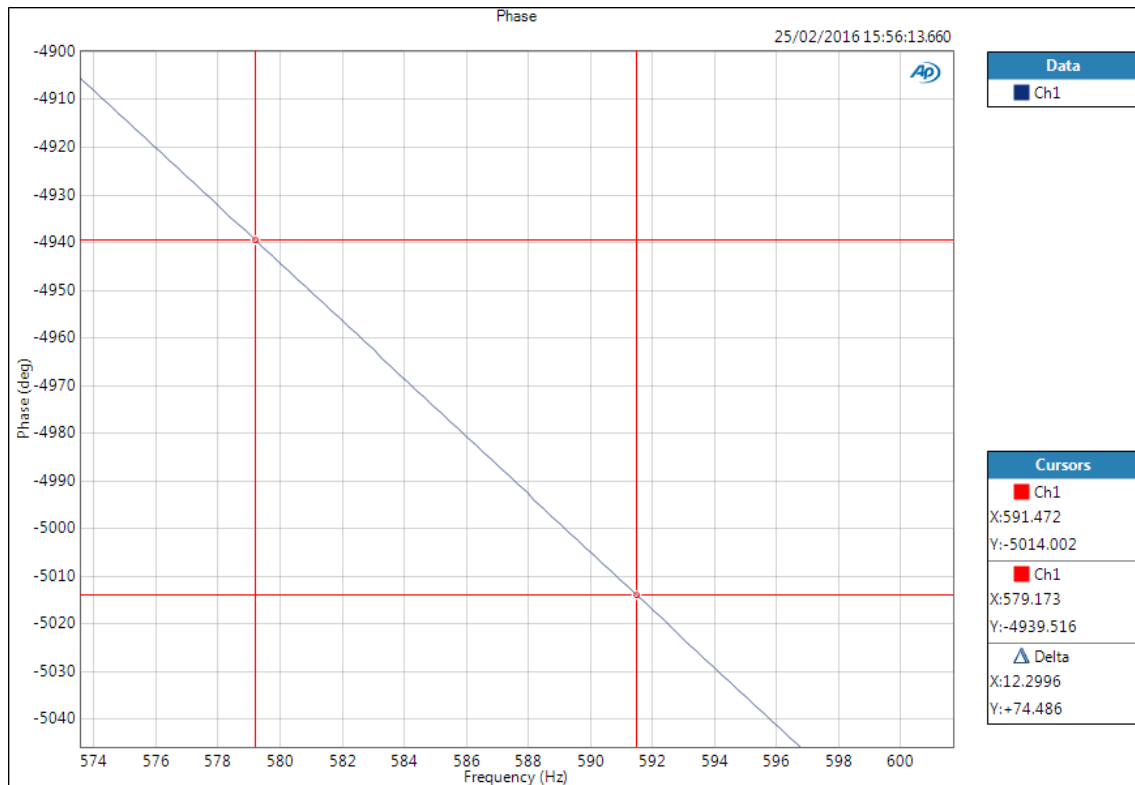


Figure 27

Figure 27 shows the phase response for our FIR filter. The group delay in this case is $t_g = \frac{\delta y}{\delta x} * \frac{1}{360} = 16.82ms$. To account for the difference in the two group delays, further measurements were performed on the phase response of the hardware without applying any filtering.

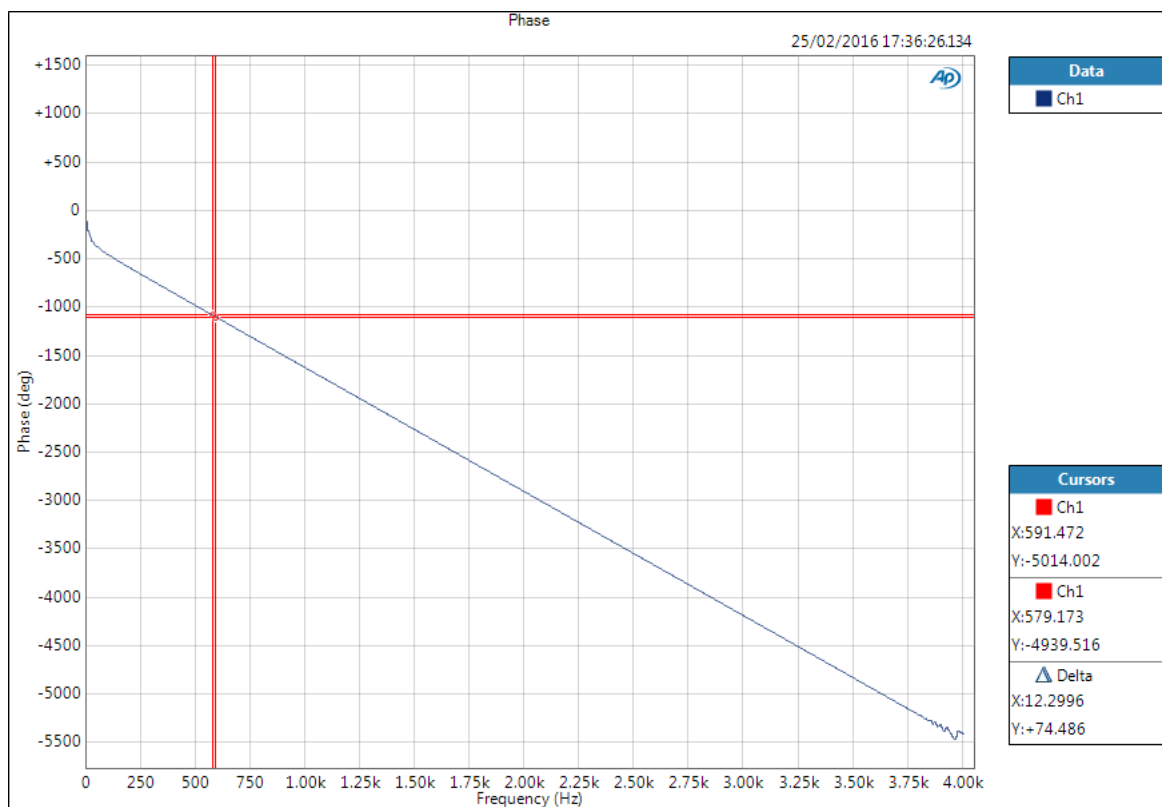


Figure 28

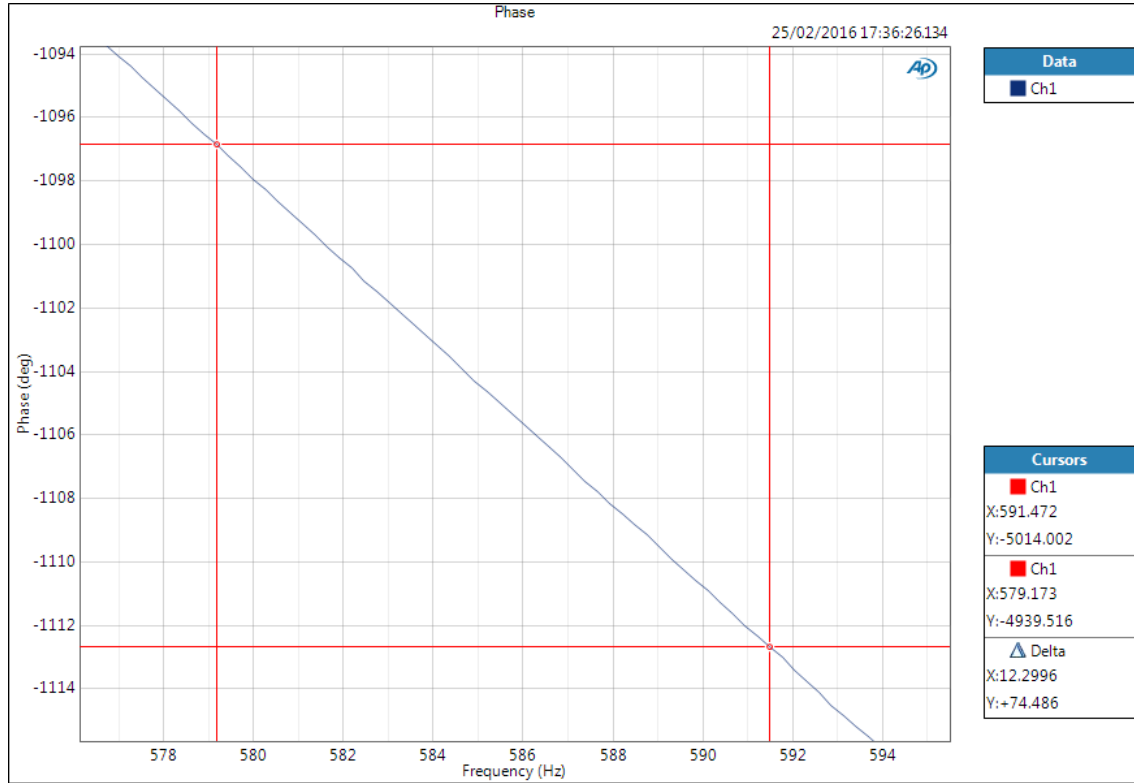


Figure 29

Figure 28 and 29 show the phase response of the board with no filtering. The group delay in this case is $t_g = \frac{\delta y}{\delta x} * \frac{1}{360} = 3.6254ms$. The difference between MATLAB's t_g and the t_g with our FIR filter is $3.54ms \sim 3.6254ms$. Hence, the main reason of a difference in slope between MATLAB's phase response and our FIR filter's phase response is due to the hardware delay.

VI. CONCLUSION

This task was completed successfully and the plots show that the implemented FIR filter conforms to the required specifications. Improvements were made to the implementation of the FIR filter in C and justified using benchmarks done using breakpoints. The frequency and phase response was measured using an audio analyzer and commented on.

VII. REFERENCES

- [1] MATLAB, "Parks-McClellan optimal FIR filter order estimation," [Online]. Available: <http://uk.mathworks.com/help/signal/ref/firpmord.html>. [Accessed 18 2 2016].
- [2] S. W. S. (Ph.D.), "Chapter 28: Digital Signal Processors - Circular Buffering," [Online]. Available: <http://www.dspguide.com/ch28/2.htm>. [Accessed 22 2 2016].
- [3] K. Wada, "Ring Buffer Basics," [Online]. Available: <http://www.embedded.com/electronics-blogs/embedded-round-table/4419407/The-ring-buffer>. [Accessed 22 2 2016].
- [4] D. Rao, "Circular Buffering on TMS320C6000," Texas Instruments, 4 2001. [Online]. Available: <http://www.ti.com/lit/an/spra645a/spra645a.pdf>. [Accessed 22 2 2016].
- [5] M. Tyson, "A simple, fast circular buffer implementation for audio processing," [Online]. Available: <http://atastypixel.com/blog/a-simple-fast-circular-buffer-implementation-for-audio-processing/>. [Accessed 22 2 2016].

- [6] Iowegian International, "FIR Filter Properties," [Online]. Available: <http://dspguru.com/dsp/faqs/fir/properties>. [Accessed 22 2 2016].
- [7] Iowegian International, "FIR Filter Implementation," [Online]. Available: <http://dspguru.com/dsp/faqs/fir/implementation>. [Accessed 22 2 2016].
- [8] A. Save, "constant pointer vs pointer on a constant value [duplicate]," 10 4 2012. [Online]. Available: <http://stackoverflow.com/questions/10091825/constant-pointer-vs-pointer-on-a-constant-value>. [Accessed 25 2 2016].
- [9] Texas Instruments, "TMS320C6000 Optimizing Compiler v7.4," 7 2012. [Online]. Available: <http://www.ti.com/lit/ug/spru187u/spru187u.pdf>. [Accessed 23 2 2016].
- [10] P. D. Mitcheson, "Lab 4 – Real-time Implementation of FIR Filters," 1 2014. [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-591059-dt-content-rid-2714544_1/courses/DSS-EE3_19-15_16/lab4.pdf. [Accessed 25 2 2016].

VIII. APPENDIX

A. MATLAB Script

```
F = [375 450 1600 1750];
Fs = 8000;
rp = 0.4;
A = [0 1 0];
%dB conversion
DEV=[10^(-48/20) (10^(rp/20)-1)/(10^(rp/20)+1) 10^(-48/20)];
[N,Fo,Ao,W]=firlpmord(F,A,DEV,Fs);
N=N+6;
B=firlpm(N,Fo,Ao,W);
freqz(B,1,4096,Fs);
%save coefficients
fid=fopen('fir_coef.txt','w');
fprintf(fid,'double b[]={');
fprintf(fid,'%10e,\t',B);
fprintf(fid,'};\n');
fclose(fid);
```

B. MATLAB *fir_coef.txt*

```
double b[]={3.8000157588e-03, 3.4293399003e-03, 7.9698009644e-04, -2.8463456703e-03, -
3.8817081071e-03, -1.7078377035e-03, 3.9943516181e-04, -3.0432683027e-04, -
2.2844088452e-03, -2.0448832644e-03, 6.5245418549e-04, 2.3148252117e-03,
9.0333001585e-04, -9.8979079372e-04, 5.0642777031e-05, 2.9544138091e-03, 3.4847267401e-
03, 6.1459980536e-04, -1.6840704798e-03, -2.6985502227e-04, 2.2499063667e-03,
1.2685853213e-03, -2.7064078637e-03, -4.4358403087e-03, -1.7165285837e-03,
1.0893566368e-03, -4.9665344898e-04, -4.0865705142e-03, -3.7514130897e-03,
9.3622066748e-04, 3.8823359672e-03, 1.4960976062e-03, -1.7239410574e-03, 1.0513796284e-
04, 5.0859308600e-03, 6.0289886225e-03, 1.3023908745e-03, -2.422389349e-03, -
1.4151650636e-04, 3.7187476449e-03, 1.8964760243e-03, -4.4158755938e-03, -
6.9458381159e-03, -2.7640892596e-03, 1.1141618245e-03, -1.6191921212e-03, -
6.6766214921e-03, -5.4772304689e-03, 1.5886545023e-03, 5.3604697192e-03,
1.7324853432e-03, -1.9739995007e-03, 1.7195044682e-03, 8.5706294398e-03, 8.7043658154e-
03, 1.7736396234e-03, -2.2347322235e-03, 1.6982643353e-03, 5.6761639686e-03,
1.0470459249e-03, -7.7525004372e-03, -9.4468904352e-03, -3.2033102537e-03, -
9.9920489031e-05, -5.8400936617e-03, -1.1343477951e-02, -6.5080563979e-03,
3.6640983397e-03, 6.1392644933e-03, 3.8550236849e-06, -1.7789978456e-03, 7.2254273965e-
03, 1.5821807863e-02, 1.1775880518e-02, 1.2015407981e-03, -2.1380429713e-04,
8.1162827691e-03, 9.7678462260e-03, -3.0955491256e-03, -1.5895103563e-02, -
1.2913320827e-02, -2.1844471312e-03, -4.0307586625e-03, -1.8625889011e-02, -
2.3343661954e-02, -7.1578570404e-03, 1.0054991484e-02, 6.3230178129e-03, -
7.1124788052e-03, -8.0488996807e-04, 2.6019541220e-02, 3.9342484554e-02,
2.0525654758e-02, -1.3984061929e-03, 9.1431818441e-03, 3.5715947720e-02, 2.7284702946e-
02, -2.3318694232e-02, -5.7204877031e-02, -3.1950547386e-02, 2.7950867454e-03, -
```

```

3.7792552355e-02, -1.3998382379e-01, -1.6969388188e-01, -2.6181185689e-02,
2.0725528721e-01, 3.2210216820e-01, 2.0725528721e-01, -2.6181185689e-02, -
1.6969388188e-01, -1.3998382379e-01, -3.7792552355e-02, 2.7950867454e-03, -
3.1950547386e-02, -5.7204877031e-02, -2.3318694232e-02, 2.7284702946e-02,
3.5715947720e-02, 9.1431818441e-03, -1.3984061929e-03, 2.0525654758e-02, 3.9342484554e-
02, 2.6019541220e-02, -8.0488996807e-04, -7.1124788052e-03, 6.3230178129e-03,
1.0054991484e-02, -7.1578570404e-03, -2.3343661954e-02, -1.8625889011e-02, -
4.0307586625e-03, -2.1844471312e-03, -1.2913320827e-02, -1.5895103563e-02, -
3.0955491256e-03, 9.7678462260e-03, 8.1162827691e-03, -2.1380429713e-04,
1.2015407981e-03, 1.1775880518e-02, 1.5821807863e-02, 7.2254273965e-03, -1.7789978456e-
03, 3.8550236849e-06, 6.1392644933e-03, 3.6640983397e-03, -6.5080563979e-03, -
1.1343477951e-02, -5.8400936617e-03, -9.9920489031e-05, -3.2033102537e-03, -
9.4468904352e-03, -7.7525004372e-03, 1.0470459249e-03, 5.6761639686e-03,
1.6982643353e-03, -2.2347322235e-03, 1.7736396234e-03, 8.7043658154e-03, 8.5706294398e-
03, 1.7195044682e-03, -1.9739995007e-03, 1.7324853432e-03, 5.3604697192e-03,
1.5886545023e-03, -5.4772304689e-03, -6.6766214921e-03, -1.6191921212e-03,
1.1141618245e-03, -2.7640892596e-03, -6.9458381159e-03, -4.4158755938e-03,
1.8964760243e-03, 3.7187476449e-03, -1.4151650636e-04, -2.4222389349e-03,
1.3023908745e-03, 6.0289886225e-03, 5.0859308600e-03, 1.0513796284e-04, -1.7239410574e-
03, 1.4960976062e-03, 3.8823359672e-03, 9.3622066748e-04, -3.7514130897e-03, -
4.0865705142e-03, -4.9665344898e-04, 1.0893566368e-03, -1.7165285837e-03, -
4.4358403087e-03, -2.7064078637e-03, 1.2685853213e-03, 2.2499063667e-03, -
2.6985502227e-04, -1.6840704798e-03, 6.1459980536e-04, 3.4847267401e-03,
2.9544138091e-03, 5.0642777031e-05, -9.8979079372e-04, 9.0333001585e-04, 2.3148252117e-
03, 6.5245418549e-04, -2.0448832644e-03, -2.2844088452e-03, -3.0432683027e-04,
3.9943516181e-04, -1.7078377035e-03, -3.8817081071e-03, -2.8463456703e-03,
7.9698009644e-04, 3.4293399003e-03, 3.8000157588e-03, };

```

C. *intio.c* Source 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 4: FIR Filters

***** 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"

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

#include <../coef/fir_coef_213.txt>
#define N 213

```



```

#ifdef (N%2)
#define mid_pt ((N-1)/2)
#else
#define mid_pt (N/2)
#endif
/***** 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;

//delay lines
double x[N] = { 0 };
double x_extended[2 * N] = { 0 };
/***** Function prototypes *****/
void init_hardware(void);
void init_HWI(void);

void interrupt_service_routine(void);
double non_cir_FIR(double sample_in);
double cir_FIR(double sample_in);
double linear_cir_FIR(double sample_in);
double sym_fir(double sample);
double non_cir_FIR_sym(double sample_in);
double non_cir_FIR_2(double sample_in);
double sym_fir_ptr(double sample_in);

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

    // initialize board and the audio port
    init_hardware();
    /* initialize hardware interrupts */
    init_HWI();

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

/***** 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);

}

/***** HWI() *****/
/* Change IRQ_EVT_RINT1 to IRQ_EVT_XINT1 when transitioning from a receive to transmit
interrupt */
void init_HWI(void)
{
    IRQ_globalDisable();           // Globally disables interrupts
    IRQ_nmiEnable();               // Enables the NMI interrupt (used by the
debugger)
    IRQ_map(IRQ_EVT_RINT1, 4);      // Maps an event to a physical interrupt
    IRQ_enable(IRQ_EVT_RINT1);      // Enables the event
    IRQ_globalEnable();             // Globally enables interrupts
}

/***** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE*****/
/**
 * This function gets executed upon a hardware interrupt. It reads in a sample,
 * then outputs the output of one the FIR functions.
 */
void interrupt_service_routine(void) {
    double sum = 0.0;
    double sample = 0.0;
    sample = (double)mono_read_16Bit();
    //all pass if needed
    //mono_write_16Bit((Int16)(sample*4));
    //FIR FUNCTIONS BELOW
    //sum = non_cir_FIR(sample);
    //sum = cir_FIR(sample);
    //sum = linear_cir_FIR(sample);
    //sum = sym_fir(sample);
    //sum = non_cir_FIR_sym(sample);
    //sum = non_cir_FIR_2(sample);
    sum = sym_fir_ptr(sample);
    mono_write_16Bit((Int16)(sum * 2));
}

/**
 * Naive non circular FIR implementation. Uses a for loop to implement delay
 * line.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return [64 bit MAC value]
 */
double non_cir_FIR(double sample_in) {
    int i, j; //define variables for use in loops
    double sum = 0;
    for (i = N - 1; i > 0; i--) {
        x[i] = x[i - 1]; //array shuffling method as described in
handout
    }
}

```

```

        x[0] = sample_in;                                //read in newest sample to one end of the
array

        for (j = 0; j < N; j++) {
            sum += b[j] * x[j];                          //MAC
        }
        return sum;
    }

/**
 * Implements circular buffer FIR filter. Read/write index used to keep track of
 * oldest sample and MAC iteration. If statements used to handle wraparound.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return          [64 bit MAC value]
 */
double cir_FIR(double sample_in) {
    int j, read_index;                                //define variables for use in loops
    double sum = 0.0;
    static int write_index = N - 1;                    //use a static variable to prevent
reinitialization between function calls, write index is the pointer to the newest sample
in the vector x
    x[write_index] = sample_in;                        //read the newest sample in at the pointer
write index
    for (j = 0; j < N; j++) {
        read_index = (write_index + j > N - 1) ? write_index + j - N :
write_index + j; //read index starts from write index (j=0 at first run) and iterates
through the array, wrapping around (circular buffer) when the iterator is larger than
the size of array x
        sum += b[j] * x[read_index]; //MAC
    }
    if (--write_index == -1) write_index = N - 1; //update write_index to oldest
sample for overwriting on next function call, and check for wraparound of write_index -
circular buffering
    return sum;
}

/**
 * Exploits linear phase filter properties (symmetrical) to cut down half of MAC
 * iterations. Uses a circular buffer.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return          [64 bit MAC value]
 */
double linear_cir_FIR(double sample_in) {
    int j, read_index, i, read_index2;
    double sum;
    static int write_index = N - 1;                    //use static to keep value of write_index
between function calls
    i = N - 1;                                         //last array index
    x[write_index] = sample_in;                        //write incoming sample to current value of
write_index
    read_index = (write_index + mid_pt > N - 1) ? write_index + mid_pt - N :
write_index + mid_pt; //handle circular buffering wraparounds, and acts as start of MAC
    sum = b[mid_pt] * x[read_index];                  //execute MAC on midpoint, as there is only
one point to convolve, instead of two, so do outside for loop
    for (j = 0; j < mid_pt; j++) {                    //calculate FIR filter inwards -
read_index2 is the mirror of read_index
        read_index = (write_index + j > N - 1) ? write_index + j - N :
write_index + j;
        read_index2 = ((read_index + i) > N - 1) ? read_index + i - N :
read_index + i;
        sum += b[j] * (x[read_index] + x[read_index2]);
        i = i - 2;                                    //as calculating
inwards, move both iterators
    }
    if (--write_index == -1) write_index = N - 1; //update write_index to oldest
sample for overwriting on next function call, and check for wraparound of write_index -
circular buffering
    return sum;
}

```

```

/**
 * Doubles the size of the buffer used to implement the delay line to remove
 * need for if statements checking for wraparound in the MAC loop as it will
 * never overflow.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return          [64 bit MAC value]
 */
double sym_fir(double sample) {
    static int new_index = N - 1;
    int old_index, iterator;
    double sum;
    old_index = new_index + N - 1;           //index the extended buffer by
adding array size
    x_extended[new_index] = sample;         //copy sample in
    x_extended[new_index + N] = sample;     //to both sections of the array
    sum = b[mid_pt] * x_extended[new_index + mid_pt]; //calculate midpoint MAC
first as it only needs one value
    for (iterator = 0; iterator < mid_pt; iterator++) { //loop iterations decreased
by half
        sum += b[iterator] * (x_extended[new_index + iterator] +
x_extended[old_index--]); //calculate MAC using both sections of the array
    }
    if (--new_index < 0) new_index = N - 1; //update oldest sample index
    return sum;
}

/**
 * Original non-circular approach, with symmetrical properties.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return          [64 bit MAC value]
 */
double non_cir_FIR_sym(double sample_in) {
    int i, j;
    double sum = 0;
    for (i = N - 1; i > 0; i--) {
        x[i] = x[i - 1]; //array shuffling
    }
    x[0] = sample_in;
    sum = b[mid_pt] * x[mid_pt]; //compute midpoint MAC first as it only
requires one coefficient
    for (j = 0; j < mid_pt; j++) { //N/2 iterations
        sum += b[j] * (x[j] + x[N - 1 - j]); //rest of MAC
    }
    return sum;
}

/**
 * This version of the non circular buffer moves the shift for loop inside the
 * MAC for loop. This version is, surprisingly, quicker than the previous
 * circular buffer linear phase functions as the compiler parallizes both the
 * shifting and the MAC. God bless Texas Instruments.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return          [64 bit MAC value]
 */
double non_cir_FIR_2(double sample_in) {
    int i;
    double sum = b[N - 1] * x[N - 1]; //might as well initialize to a value not
calculated by for loop as the for loop iteration size for delay and MAC differ by 1
    x[0] = sample_in; //read input sample
    for (i = N - 2; i >= 0; i--) { //size of MAC loop is larger than
delay loop by 1
        sum += b[i] * x[i];
        x[i + 1] = x[i]; //change from i>i-1 to i+1>i, as MAC needs
to index b[0] and x[0], and x[-1] is undefined
    }
    return sum;
}

```

```

/**
 * Exploits linear phase properties, and uses a double delay line for speed up.
 * Also replace array indices with const double pointers. Inward movements of
 * the points calculate the MAC, with the midpoint being calculated outside the
 * loop.
 * @param sample_in [16 bit sample from mono_read_16Bit, casted to 64 bit]
 * @return          [64 bit MAC value]
 */
double sym_fir_ptr(double sample_in) {
    int i;
    static int index = N-1;          //starting write index for new sample
    double sum = 0.0;
    const double *ptr_b = b, *ptr_x = x_extended + index, *ptr_x2 = x_extended +
index + N - 1; //use const double pointer to index the array instead of array indices
    //const double means the pointed to data is constant, but the pointer can change
    //ptr_b points to starting address of array b

    x_extended[index] = x_extended[index + N] = sample_in;    //write incoming
sample to both array and extended array

    for (i = 0; i < mid_pt; i++) {
        sum += *ptr_b++ * (*ptr_x++ + *ptr_x2--);    //compute MAC by moving
pointers inwards
    }
    sum += *ptr_b++ * *ptr_x++;    //compute midpoint, points are conveniently
in the right place after for loop

    if (--index < 0) index += N;    //handle write index overflow
    return sum;
}

```

D. Dr. Paul Mitcheson's email

RTDSP: lab 4 and lectures this week
Mitcheson, Paul D
Wed 24/02/2016 10:01
To: ee_rtdsp <ee_rtdsp@imperial.ac.uk>;
Dear All,

[...]

Also, I have been having some interesting discussions with you in labs about the specific filter responses that you measure compared to the filter performance calculated in Matlab. It seems that whilst the exact response in Matlab can relatively easily be adjusted to meet the specifications exactly, when you come to implement on the DSP chip, the measured response from the Audio Precision unit is not exactly the same, although it is very very close. This is the first year we have seen such discrepancies and it probably comes from me specifying a greater attenuation in the stop band than previously. The main error seems to be seen around the stop band edge of the first transition band where one of the lobes due to a zero shows some difference in response between Matlab and the measurement. Firstly, the numbers and voltage levels involved at such attenuation levels are small, so we can expect noise to play a part in this. Secondly, when you make a comparison between the DSP filter and the Matlab calculations, make sure that you allow for the fact that the DSP board has some delay due to the ADC/DAC, phase response due include these effects). This can be done by writing the input direct to the output on the DSP with no filtering and measuring the bare board response, which can then be subtracted.

All that said, if you have already tried to account for these effects, and there are still small errors, do not worry about it. In the comparison I have seen, due to the strong attenuation, the errors are really very small and I at this point I can't see why we have what seems to be a systematic error: but in all honesty it's too small to worry about and I think we have to put it down to measurement error on the APX units.

Make sure you comment on it in the reports, but don't worry about the very small errors.

Thanks,

Paul

[...]