RTDSP Lab 4 Report

{ayy17,zy4417}@ic.ac.uk

I. Introduction

In this lab, the aim is to design a FIR filter using MATLAB and optimise the implementation of FIR filter (both circular and non-circular) in C on C6713 DSK system. Below are answers to four assessed questions.

II. Design of FIR filter with proof

The starting point is to construct a FIR filter with coefficients that satisfy the given constraints as shown in Fig 1.

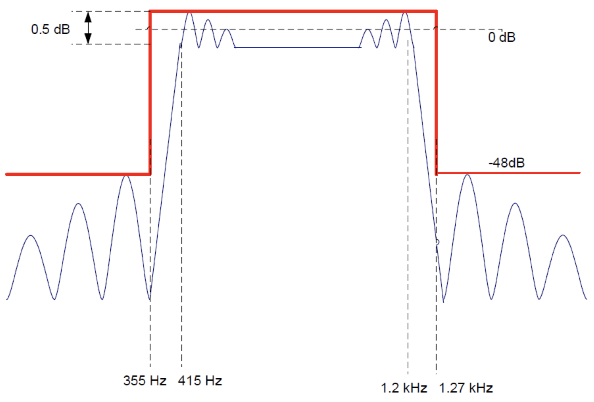

Fig. 1: Filter Specification

The designed filter has been simulated in MATLAB as shown in Fig 2, and the frequency response clearly satisfies the specification listed as follows: Cutoff frequency {355,415,1200,1270} Hz, pass-band ripple 0.5dB, stop-band ripple -48dB.

The Matlab functions used are *firpm*() and *firpmord()*, which approximate the filter order and return filter coefficients based on the Parks-McClelland algorithm.

Fig 2 illustrates the actual frequency response analysed by the audio analyser APX520, which slightly differs from the expected one but certainly satisfies the specification. This can be explained in terms of gain and phase deviation.

Gain Analysis By tracing back to the DSK circuit diagram given in lab 3, a potential divider of 2:1 has been place before the signal arrives at the input socket. Thus, the maximum gain obtained would be halved before sampled. In addition, two channels (left & right) were used and, hence, the amplitude would be halved when taken as input. Therefore, as shown in Fig 2, the obtained gain between the pass-band should be expressed as,

$$20log(\frac{1}{4}) \approx -12dB \tag{1}$$

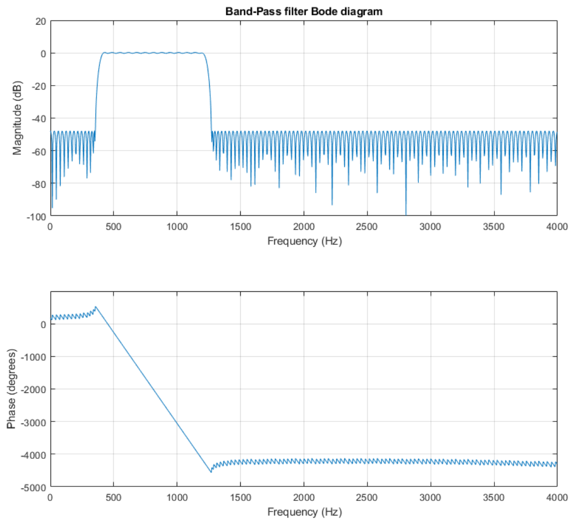

Fig. 2: Bode Diagram for designed band pass filter

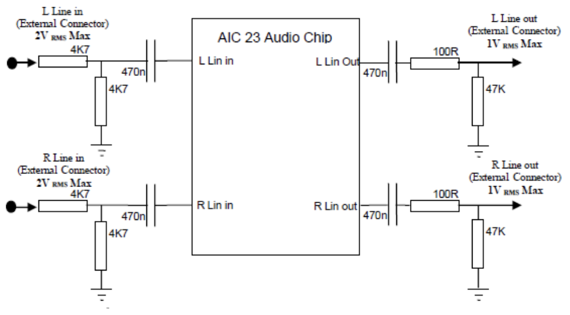
L Line in (External Connector) 2V RMS Max
4K7
470n
4K7
AIC 23 Audio Chip
L Lin in
L Lin Out
470n
100R
47K
L Line out (External Connector) 1V RMS Max
R Line in (External Connector) 2V RMS Max
4K7
470n
4K7
R Lin in
R Lin out
470n
100R
47K
R Line out (External Connector) 1V RMS Max

Fig. 3: DSK circuit diagram

Phase Analysis Consider the Z-transform expression as follows,

$$H(z) = \sum_{i=0}^{n} b[i]z^{-i} \tag{2}$$

The above equation can be factorized and therefore expressed in zeros and poles as,

$$H(z) = \frac{(z_1 - a_1) + ... + (z_n - a_n)}{z} \tag{3}$$

As depicted in Fig 3, the zeros and poles were placed along the unit circle. In order to analyse how phase changes with frequency, the above sequence can be simplified as,

$$H(z) = \sum_{i=0}^{(n-1)/2} b[i](z^{-i} + z^{-(n-i)}) \tag{4}$$

where,

$$b[i] = b[n-i] \tag{5}$$

satisfies the symmetric property of a FIR filter. Hence, by replacing z with $e^{-j\omega}$,

$$\begin{aligned} H(z) = H(e^{j\omega}) &= \sum_{i=0}^{(n-1)/2} b[i](z^{-i} + z^{-(n-i)}) \\ &= e^{\frac{-j\omega n}{2}} \sum_{i=0}^{(n-1)/2} b[i](z^{i-\frac{n}{2}} + z^{\frac{n}{2}-i}) \\ &= e^{\frac{-j\omega n}{2}} \sum_{i=0}^{(n-1)/2} 2b[i]cos(\omega(i - \frac{n}{2})) \end{aligned}$$

The phase delay can be calculated as,

$$\frac{\mathrm{d}\angle H(z)}{\mathrm{d}\omega} = -\frac{n}{2} \tag{6}$$

where the phase of real numbers is zero. Assuming that N is equal to 248, the group delay can be calculated as $\frac{N}{2f_s} = 124/8000 = 15.5$ms. As a result, Fig 4 confirms the result as a linear line was observed in the pass-region and has a constant group delay time.

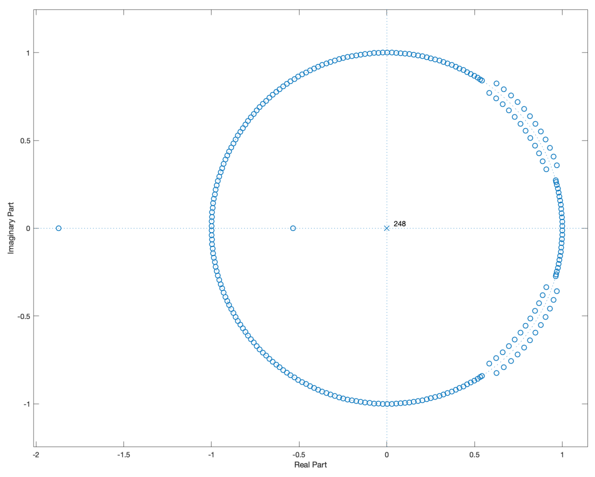

Fig. 4: Z-plane zeros & poles

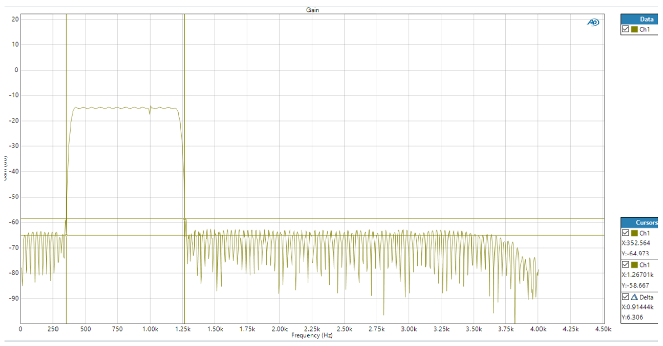

Fig. 5: Actual Frequency Graph by APX520

III. Non-circular Filter Implementation and Different Compiler Optimisations

Filter Implementation This filter design was altered based on Lab 3 with the only changed function being interrupt function InterruptSR(). The function would be called whenever the DSK receives an incoming signal. In our design, we use a Non-circular filter with order 249.

Firstly, the input read from codec would be placed into an input array as the first element. Secondly, all the elements in input array would be convolved with the corresponding coefficients $b[i]$ generated from MATLAB. The result would be passed into *mono_write_16Bit* for output. Lastly, the input array would be shifted to make up new space for the next incoming element. As expected, the output on oscilloscope has been depicted in Fig X .

Compiler Optimisation Table 1 compares the number of clock cycles for running the compiled code with different levels of optimisation. It is worth mentioning that a decrease is observed among different trails. This might be explained by memory access is required for the first function call. Once data has been loaded into registers, arithmetic operations on registers take less clock cycles to execute.

Trial #	Optimization level / Clock cycles			
	No	Level 0	Level 2	Level 3
1	15916	13106	3983	3994
2	15400	12655	3464	3464
3	15399	12654	3464	3464

TABLE I: Different optimization level performance comparison on Non-circular Buffers

```
    void InteruptSR(){
    int i;
    int j;

    input[0]=mono_read_16Bit();
    ///read a new sample and place it
    // into buffer

    double temp=0;
    // initialise the temporary variable
    // as a accumulator
    for(j=0;j<filter_size;j++){
    // loop through all the elements
    // needed to compute convolution
      temp+=input[j]*b[j];
    // compute element-wise multiplication
    // for convolution
    }
    mono_write_16Bit(temp);
    //write the convolution result to codec

    for (i=filter_size-1;i>0;i--){
      input[i]=input[i-1];
      // shift the buffer by one element
      // in order to make a new space
      // for next incoming input
    }
}
```

IV. Circular Filter Implementation and different compiler optimisations

Basic Circular Buffer Design As shown in the previous section, shifting elements in buffer requires making duplicates of arrays at run-time. Consequently, the number of cycles per function call would be relatively high. Hence, a smarter design would be shown in this section. The code section below indicates how the basic circular buffer was designed in C. *Offset* is a global index which keeps tracks of the current position of the newest element. By increasing this index, what effectively happens is a ”circular” buffer which is equivalent to the previous design of ”shifting” buffer without the redundancy introduced by shifting operations.

Fig 3 depicts how the convolution is performed on circular buffers. The most challenging part would be to correctly trace array indices. Code line 6 corresponds to the sections 1 indicated in Fig 3, where the partial convolution for elements between *Offset* and last element in the array. In order to complete the convolution operation for all elements, a ”wrap-around” is needed for elements that are before *Offset* in the array. The actual implementation is by Code line 9 which perform this operations on sections 2, completing the convolution for elements between index 0 and *Offset*.

In conclusion, this design is smarter and computationally efficient by exploiting the trick of indexing.

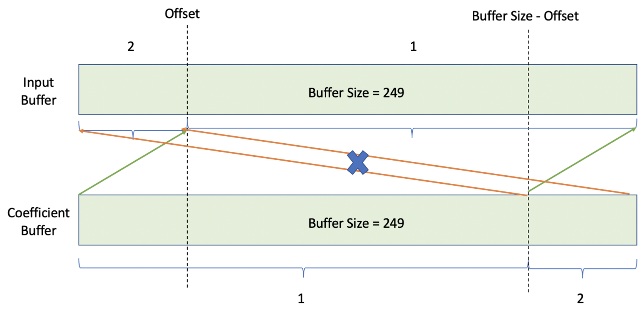

Fig. 6: Basic Circular Buffers

Trial #	Optimization level / Clock cycles			
	No	Level 0	Level 2	Level 3
1	12863	12353	3952	3952
2	12430	11918	3520	3515
3	12431	11913	3520	3515

TABLE II: Different optimization level performance comparison on Base Circular Buffer

```
1  void base_cir(){
2      // read data into appropriate position
3      input[offset] = mono_read_16Bit();
4      int i;
5      float output = 0;
6      for(i = 0;i<filter_size-offset;i++){
7      // convolution by choosing the offset element
8      //as starting point
9          output+=input[offset+i]*b[i];
10     }
11     for(i = filter_size-offset;i<filter_size;i++){
12     //wrap around and continue with convolution
13     //for elements before the offset element
14         output+=input[offset+i-filter_size]*b[i];
15     }
16     offset++; // increment offset
17     //wrap around offset index
18     if (offset==filter_size){offset=0;}
19     mono_write_16Bit(output);
20 }
```

Circular buffer using pointers to float
It is noticed that index calculations require load & write via memory which includes extra computation time. Hence, it would be reasonable to use pointer to arrays and increment accordingly to implement a circular buffer. As shown in Fig X and code below, the logic flow is similar to the basic circular buffer. Moreover, it is noticed that changing data type from double to float can improve performance because float is only 32 bits and less clock cycle is needed for ”float” operation than ”double” operation.

```
void ptr_cir(){
    int i;
    float* iter_val = coef_ptr;
    float* buf_val;
    output = 0;
    ++offset;
    //prevent offset from out-of-range
    offset = offset % filter_size;
    buf_val  = cir_ptr + offset;

    *buf_val = mono_read_16Bit();
   while(iter_val - coef_ptr <filter_size - offset -1){
       //-1 because the pointer need to be within range
       output += (*buf_val++)* (*iter_val++);
   }
   // compensate for the -1 above
   output += (*buf_val)*(*iter_val);
    //wrap around to continue for elements before offset
   buf_val-= filter_size;

   while(iter_val - coef_ptr <filter_size-2){

       output += (*buf_val++)* (*iter_val++);
   }
   output += (*buf_val)*(*iter_val);
   mono_write_16Bit(output);
}
}
```

Trial #	Optimization level / Clock cycles			
	No	Level 0	Level 2	Level 3
1	9462	7484	1823	1812
2	9143	7144	1479	1508
3	9141	7143	1468	1502

TABLE III: Different optimization level performance comparison on Circular Buffer using pointer

Circular Buffers based on coefficient symmetry Multiplication in C takes longer cycles than basic arithmetic operations. Hence, it would useful if the number of calculations can be reduced. Fig 4 indicates that the Finite Impulse Response filter coefficients used are symmetric along the frequency axis. Therefore, a further optimization can be achieved by combining the input related with the symmetric positive and negative frequency coefficients. Different colors denotes different sections of coefficient pairings. It is also worth mentioning that the value of offset affects the pairing equation. Hence, in the code appended below, an if-else statement (line 6,19) was introduced to separate the two sub-cases. Fig 5 demonstrates how inputs are paired up for the first case.

```
1      void symm_cir(){
2      input[offset] = mono_read_16Bit();
3      int i;
4      output = 0;
5      // First case offset less than half buffer size
6      if(offset<((filter_size-1)/2)){
7          for(i=0;i<offset;i++){
8                  output+=(input[offset+i]+
9                  ...input[offset-i-1])*b[i];
10             }
11         for(i = offset;i<(filter_size-1)/2;i++){
12             output+=(input[offset-i-1+filter_size]+
13             ...input[offset+i])*b[i];
14             }
15         output+=b[(filter_size-1)/2]*
16         ...input[(filter_size-1)/2+offset];
17     }
18     // Second case larger than buffer size
19     else if(offset>(filter_size-1)/2){
20         for(i=0;i<filter_size - offset;i++){
21             output+=(input[offset+i]+
22             ...input[offset-i-1])*b[i];
23         }
24         for(i = filter_size-offset;
25         ...i<(filter_size-1)/2;i++){
26             output+=(input[offset-filter_size+i]+
27             ...input[offset-1-i])*b[i];
28         }
29         output+=b[(filter_size-1)/2]*
30         ...input[offset-(filter_size-1)/2 -1];
31     }
32     offset++;
33     if (offset==filter_size){offset=0;}
34     mono_write_16Bit(output);
35 }
```

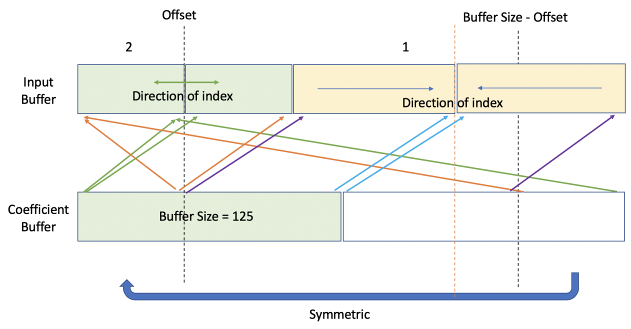

Fig. 7: Optimized symmetric buffer design

Table 4 indicates how cycles for per function call is related with compile time optimization levels. In comparison with the basic circular buffer, a significant improvement can be seen.

Circular buffer using double memory and coefficient symmetry It is seen that coefficient symmetry can

Trial #	Optimization level / Clock cycles			
	No	Level 0	Level 2	Level 3
1	5542	4903	946	944
2	5542	4903	946	944
3	5542	4904	947	945

TABLE IV: Different optimization level performance comparison on Optimized symmetric buffers

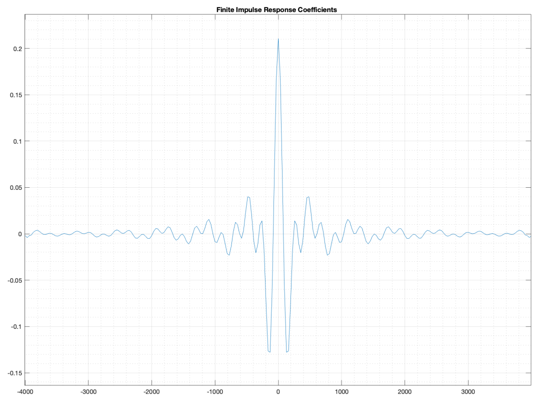

Fig. 8: FIR frequency response

significantly improve the real-time performance because of algebraic trick. However, there are many if-else statements in this approach, which results in lots of time wasted for checking overflow. A further improvement to this is having larger memory to avoid condition checking because double the memory means that if-else statements can be removed and each pair of for loops can be collapsed to avoid index wrap-around.

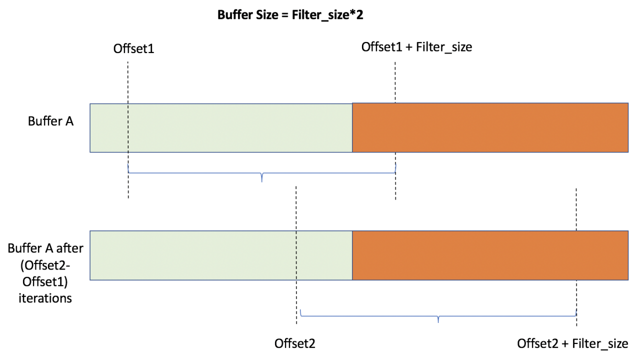
Buffer Size = Filter_size*2
Offset1
Offset1 + Filter_size
Buffer A
Buffer A after (Offset2-Offset1) iterations
Offset2
Offset2 + Filter_size

Fig. 9: Optimized buffer design

Figure 6 shows how the principles behind this mechanism. Each time a value is read from codec, it will be placed in two distinct positions, whose distance to each other is a exactly the number of FIR coefficients. In other words, the values used for convolution is always the ones between two ”walls” as indicated by figure 6. Once the offset reaches the value which is equal to the number of coefficients, both ”walls” will be reset back to the beginning points. Therefore, what effectively happens is no more overflow checking or index wrap-around is needed, as two ”walls” scan through the buffer. Below is the code which implements this best approach.

```
void double_cir (){
    float temp=mono_read_16Bit();
    input2[index]=temp;
    // index2 is always index+filter_size
    input2[index2]=temp;

    output = 0;
    int i;
    for (i=0;i<(filter_size-1)/2;i++){
        //only loop for half of the size
        //of filter coefficients
        output+=(input2[index+i]+
        input2[index2-i-1])*b[i];
        //using symmetry property
    }
    //compensate for the middle coefficient
    output+= input2[index+(filter_size-1)/2]*
    b[(filter_size-1)/2];

    index++;
    if (index==filter_size) index=0;
    index2=index+filter_size;//reset index
    mono_write_16Bit(output);
}
```

As discussed above, this design allows condition checking to be skipped, making the code tidy and efficient. Firstly, a single value read from codec is passed into an array of double memory at two different index location which are always 249 (filter coefficient size) apart. Secondly, 124 iterations have been performed to calculate the convolution in the way that two multiplications with same coefficients can be combined. Thirdly, the middle coefficient which was left out need to be compensated. Lastly, index need to be incremented and checked against the range of this double-size buffer.

Trial #	Optimization level / Clock cycles			
	No	Level 0	Level 2	Level 3
1	5140	4758	630	634
2	5140	4758	630	634
3	5140	4757	631	634

TABLE V: Different optimization level performance comparison on Optimized Circular Buffers with coefficient symmetry and double memory

Circular buffer using double memory, coefficient symmetry, and pointer access Further to the approach above, an attempt has been made to further accelerate the real-time performance. In theory, pointer implementation should give a faster performance because pointer access is faster than index access. However, the result in this design is not satisfactory. This might be potentially due to the over-complicated operations for pointer in C code below.

```
void tripple_cir(){

    float* iter_val = coef_ptr;
    //initialise two pointers corresponding
    //to two "walls" in the previous design
    float* buf_val1;
    float* buf_val2;
    output = 0;
    //wrap around offset once out of range
     offset = ++offset % filter_size;
     //set two pinters to the offset position
     // and (offset+filter_size) position
     buf_val1  = cir_ptr2 + offset;
     buf_val2 = cir_ptr2 + filter_size+offset-1;
     float temp = mono_read_16Bit();
     *buf_val1 = temp;
     *buf_val2 = temp;

    while(iter_val - coef_ptr <((filter_size -1)/2)-1)
    {// while loop finishes once the coefficient
    // pointer reaches the end of this array
        output += ((*buf_val1++)+
        (*buf_val2--))*(*iter_val++);
    }
    // compensation for middle coefficient
    output += ((*buf_val1))* (*iter_val);
    mono_write_16Bit(output);
}
```

Trial #	Optimization level / Clock cycles			
	No	Level 0	Level 2	Level 3
1	5542	4903	946	944
2	5542	4903	946	944
3	5542	4904	947	945

TABLE VI: Different optimization level performance comparison on Optimized symmetric buffers

Analysis of different optimisers As seen throughout the report, optimisers can significantly reduce the number of clock cycle and improve real-time performance. Therefore, it is important to reflect on how different optimsiers work and improve code above, thereby exploring if C code can be tailored further for improvements. To keep the report tidy, only relevant features of compilers will be discussed with related code and design.

In the process of convolution, the most time consuming part is the looping, which requires condition checking and multiplication. With Level 0 optimisation, time can be saved because this optimiser simplifies the control statement. With Level 2 optimisation, more time can be saved as software pipelining and loop unrolling will be conducted.

Further thoughts and reflection Various other methods were proposed but not implemented due to performance prediction and time constraints. In this report, it was shown that the best performance obtained while using level 2 optimisation can be 600 clock cycles. Therefore, a bottleneck was predicted to have occurred and convergence of performance was predicted. The first method involves doubly linked list using dynamic memory allocation. This would be significantly beneficial if dynamic coefficients are acquired during run time. Another point which is worth illustrating is double linked list are efficient on node modification. In the basic non-circular buffer case, shifting data would be replaced by inserting a new node and removing the last node. The total computation time and clock cycles used would be extensively reduced. The other method proposed involved research on the TI C6000 compiler data-sheet. According to the original documentation, manual unrolling for loops with *pragma UNROLL, MUSTITERATE(bounds)* could be useful. This section demonstrates work that were not done but relates

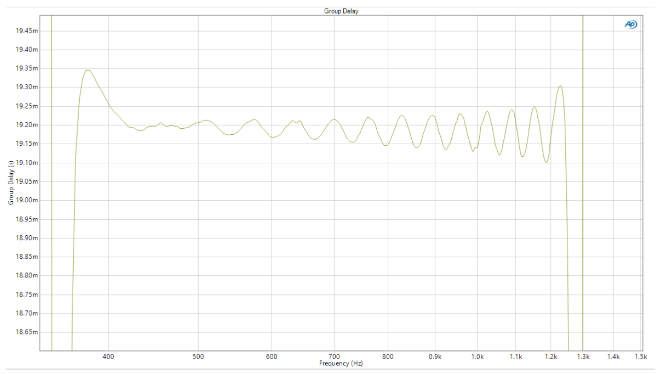
Fig. 10: Group delay Measurement

to future work if necessary.

V. Analysis of filter frequency response

In this section, the best optimized technique found was tested on the spectral analyzer. Fig 6 & 7 indicates the gain and phase diagram. The cutoff frequency observed are 350 and 12070 Hz, which is consistent with the desired frequency. The phase diagram observed is approximately linear, which is consistent with the desired phase delay. A linear phase delay would simply be a time shift which was proven in the first section. The tested group delay within the pass-band frequency lays between 19.15-19.25 ms, as shown in Fig 10. By recalling the theoretic group delay of 15.5ms, further investigation were conducted to explain. As a result of various test, a 3.5ms time delay was discovered while the raw input was fed into the spectral analyzer. Therefore, the total time delay for the signal to arrive would be $15.5 + 3.5 \approx 19.2$ms.

Besides, an interesting observation is that there will be an overshoot around the frequency at which the sine wave generator operates, if sine wave generator is not turned off. This phenomenon is due to the way audio analyser works (setting as shown by fig 10). In theory, the gain and phase diagrams below are generated by audio analyser which alters the frequency continuously such that gain and phase can be measured at each frequency. During this process, if the sine wave input from PC is not turned off, there will be a superposition between two different inputs, which significantly amplifies the gain at a specific frequency.

In conclusion, the fastest implementation has gain and phased that are expected based on the previous gain and phase analysis.

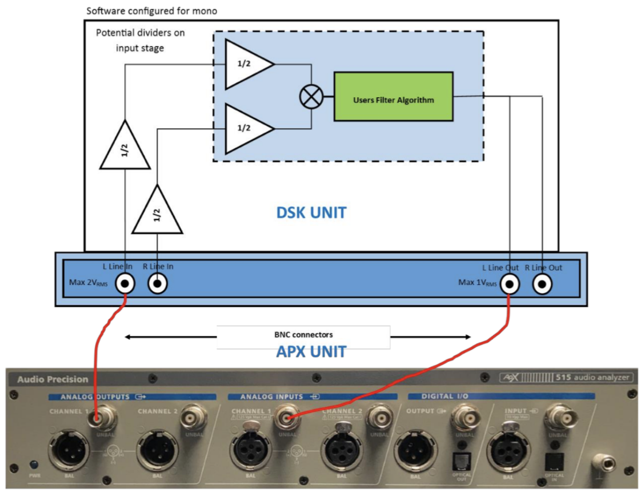

Fig. 11: Phase diagram

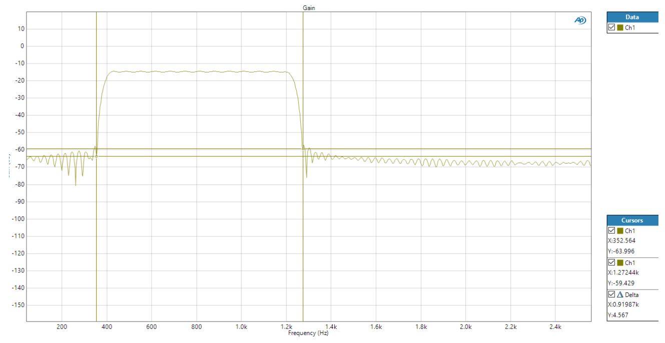
Fig. 12: Gain diagram

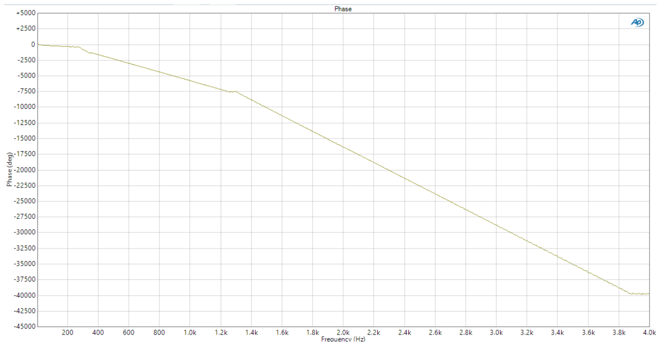
Fig. 13: Phase diagram

VI. Appendix

```c

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

              EE 3.19: Real Time Digital Signal Processing
                 Dr Paul Mitcheson and Daniel Harvey

                      LAB 3: Interrupt I/O

                    ********* I N T I O. C **********

  Demonstrates inputing and outputing data from the DSK's audio port using interrupts.

 ************************************************************************************
        Updated for use on 6713 DSK by Danny Harvey: May-Aug 2006
        Updated for CCS V4 Sept 10
 ************************************************************************************/
/*
 *  You should modify the code so that interrupts are used to service the
 *  audio port.
 */
/**************************** Pre-processor statements ******************************/

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

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

// math library (trig functions)
#include <math.h>
#include <fir_cof.txt>
// Some functions to help with writing/reading the audio ports when using interrupts.
#include <helper_functions_ISR.h>


#define filter_size 249
#define N 249
/******************************* 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                    */\
       /**********************************************************************/
};

int sample;
float input[filter_size]={0};
float input2[filter_size*2]={0};
float* cir_ptr = input;
float* cir_ptr2 = input2;
float* coef_ptr = &b[0]; // Head pointer to coef array

int index=0;
int index2= filter_size;
int offset=0;
float output=0;
// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

 /******************************* Function prototypes ********************************/
void init_hardware(void);
void init_HWI(void);
void opt_cir();
void InterupptSR();
void test_cir();
void ptr_cir();
void processing(double* input);
/********************************** Main routine ************************************/
void main(){


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

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


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

}

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

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

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

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

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


}

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

}

/******************** WRITE YOUR INTERRUPT SERVICE ROUTINE HERE***********************/
void InteruptSR(){
    int i;
    int j;


    input[0]=mono_read_16Bit();//read a new sample and place it into buffer
    double temp=0;// initialise the temporary variable as a accumulator
    for(j=0;j<filter_size;j++){ // loop through all the elements needed to compute convolution
      temp+=input[j]*b[j]; // compute element-wise multiplication for convolution
    }
    mono_write_16Bit(temp);//write the convolution result to codec

    for (i=filter_size-1;i>0;i--){
      input[i]=input[i-1];
      // shift the buffer by one element in order to
      //make a new space for next incoming input
    }
}

void base_cir(){
    input[offset] = mono_read_16Bit();
    int i;
    output = 0;
    for(i = 0;i<filter_size-offset;i++){
        output+=input[offset+i]*b[i];
    }
    for(i = filter_size-offset;i<filter_size;i++){
        output+=input[offset+i-filter_size]*b[i];
    }
    offset++;
    if (offset==filter_size){offset=0;}
    mono_write_16Bit(output);
}








//symmetry
void symm_cir(){
    input[offset] = mono_read_16Bit();
    int i;
    output = 0;
    // First case offset less than half buffer size
    if(offset<((filter_size-1)/2)){
        for(i=0;i<offset;i++){
                output+=(input[offset+i]+input[offset-i-1])*b[i];
            }
        for(i = offset;i<(filter_size-1)/2;i++){
            output+=(input[offset-i-1+filter_size]+input[offset+i])*b[i];
            }
        output+=b[(filter_size-1)/2]*input[(filter_size-1)/2+offset];
    }
    // Second case larger than buffer size
    else if(offset>(filter_size-1)/2){
        for(i=0;i<filter_size - offset;i++){
            output+=(input[offset+i]+input[offset-i-1])*b[i];
        }
        for(i = filter_size-offset;i<(filter_size-1)/2;i++){
            output+=(input[offset-filter_size+i]+input[offset-1-i])*b[i];
        }
        output+=b[(filter_size-1)/2]*input[offset-(filter_size-1)/2 -1];
    }
    offset++;
    if (offset==filter_size){offset=0;}
    mono_write_16Bit(output);
}

//pointer for float type (based on base circular buffer)
void ptr_cir(){
    int i;
    float* iter_val = coef_ptr;
    float* buf_val;
    output = 0;
    ++offset;
    offset = offset % filter_size;//prevent offset from out-of-range
    buf_val  = cir_ptr + offset;
    *buf_val = mono_read_16Bit();
   while(iter_val - coef_ptr <filter_size - offset -1){
       //-1 because the pointer need to be within range
       output += (*buf_val++)* (*iter_val++);
   }
   output += (*buf_val)*(*iter_val); // compensate for the -1 above

   buf_val-= filter_size;//wrap around to continue for elements before offset

   while(iter_val - coef_ptr <filter_size-2){

       output += (*buf_val++)* (*iter_val++);
   }
   output += (*buf_val)*(*iter_val);
   mono_write_16Bit(output);
}


//double memory + symmetry
void double_cir (){
    float temp=mono_read_16Bit();
    input2[index]=temp;
    input2[index2]=temp; // index2 is always index+filter_size
    output = 0;
    int i;
    for (i=0;i<(filter_size-1)/2;i++){//only loop for half of the size of filter coefficients
        output+=(input2[index+i]+
                input2[index2-i-1])*b[i]; //using symmetry property
    }
    output+= input2[index+(filter_size-1)/2]*b[(filter_size-1)/2];//compensate for the middle coefficient
    index++;
    if (index==filter_size) index=0;
    index2=index+filter_size;
    mono_write_16Bit(output);
}


// double + symmetry + pointer
void triple_cir(){
    float* iter_val = coef_ptr;
    float* buf_val1;
    float* buf_val2;
    float* des_ptr = coef_ptr+((filter_size -1)/2)-1;

    output = 0;

    offset = ++offset % filter_size;
    buf_val1  = cir_ptr2 + offset;
    buf_val2 = cir_ptr2 + filter_size+offset-1;
    float temp = mono_read_16Bit();
    *buf_val1 = temp;
    *buf_val2 = temp;
   while(iter_val <des_ptr){

       output += ((*buf_val1++)+(*buf_val2--))* (*iter_val++);
   }
   output += ((*buf_val1))* (*iter_val);
   mono_write_16Bit(output);
}
```