Lab 3

FIR Filters and Circular Buffering

ECE 406: Real-Time Digital Signal Processing

March 12, 2015

Christopher Daffron

cdaffron@utk.edu

There were four objectives for this lab. The first one was to get acquainted with filter design based on given specifications using both windowing and optimal methods. The second was to get to know related built-in MATLAB functions and toolboxes for FIR filter design. The third was to understand the difference between linear buffer and circular buffer in FIR implementation. Finally, the fourth was to learn how to take advantage of the linear phase property for more efficient FIR implementation.

Part 1

Subpart A

All of the tasks that are part of this section of the assignment involve doing manual calculation of selected parameters for varying filter types and then verifying those parameters using built-in MATLAB functions. The first part was to determine the minimum length of the impulse response and the value of β for a Kaiser Window bandpass filter with given parameters. The following equations are used to determine these parameters:

$$A = -20\log(\delta)$$

$$\beta = 0.5842(A-21)^{0.4} + 0.007886(A-21)$$

$$M = \frac{A-8}{0.285\Delta\omega}$$

$$\delta = \min(\delta_s, \delta_p)$$

For this problem, the following parameter values are used:

$$\delta_s = 0.01, \delta_p = 0.05$$

$$\Delta\omega = 0.05\pi$$

The following values for the parameters were determined:

$$A = -20\log(0.01) = 40$$

$$\beta = 0.5842(40-21)^{0.4} + 0.07886(40-21) = 3.3953$$

$$M = \frac{40-8}{2.285(0.05\pi)} = 89.1549$$

Therefore, the minimum length of the impulse response is 90. The group delay of the filter is defined as M/2, so it is 45. The ideal impulse response is defined as follows:

$$h_d[n] = \frac{\sin(0.625\pi(n - {}^{M}/_{2}))}{\pi(n - {}^{M}/_{2})} - \frac{\sin(0.3\pi(n - {}^{M}/_{2}))}{\pi(n - {}^{M}/_{2})}$$

The window function is defined as follows:

$$w[n] = \frac{I_0[\beta\left(1 - \left[\frac{n-\alpha}{\alpha}\right]^2\right)^{\frac{1}{2}}]}{I_0(\beta)}$$

Finally, the estimated impulse response is defined as:

$$h[n] = h_d[n]w[n]$$

The kaiserord() function of MATLAB was used to verify the calculations performed manually earlier. The code used to do this can be found in the appendix. After running the code, the calculations were found to be correct and the filter coefficients were found. The frequency response of the filter is shown below.

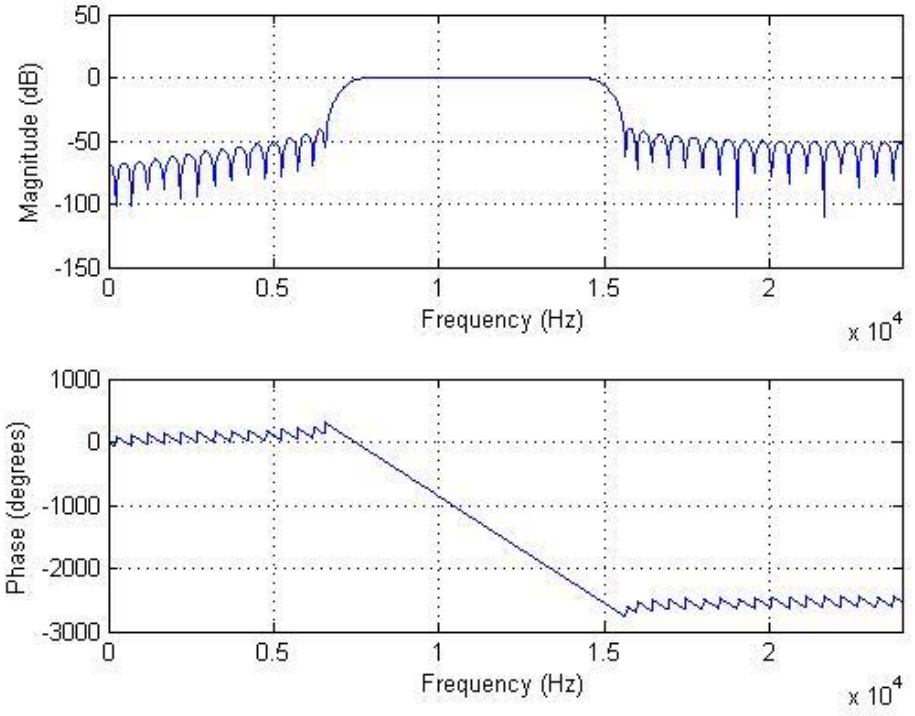

Figure 1: Frequency Response of the Kaiser Window Bandpass Filter

After performing these steps using individual MATLAB functions, the fdatool was used to design the filter and output the coefficients. A screenshot of the interface with the designed filter is shown below and the coefficients exported from this can be found in the appendix.

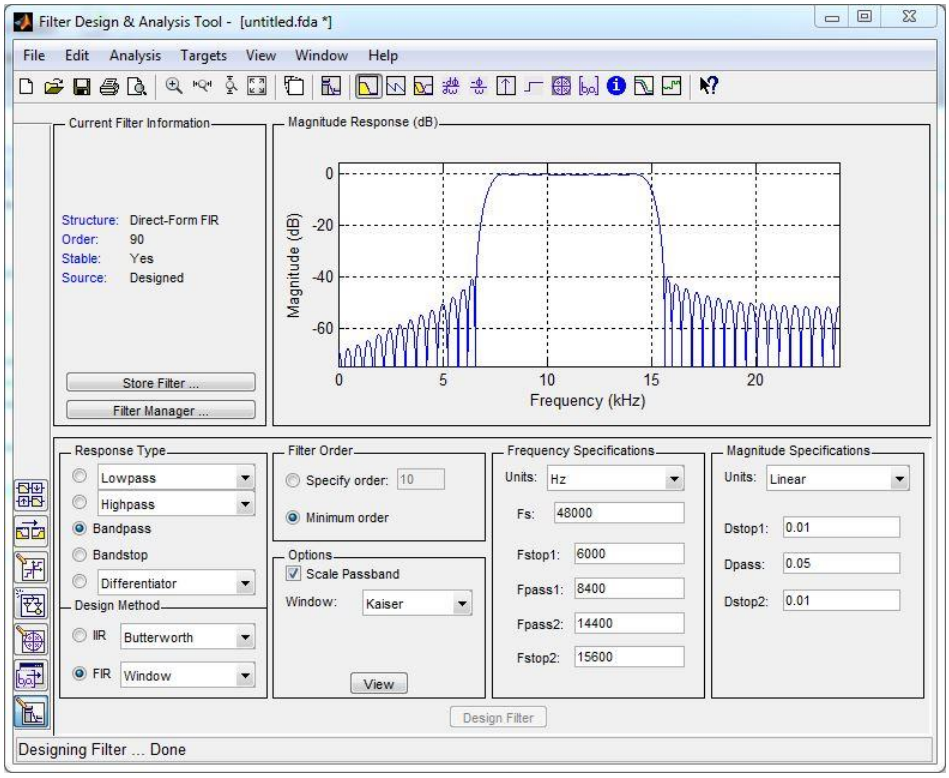

Figure 2: MATLAB fdatool filter

Subpart B

For this subpart, several calculations were performed using a Parks-McClellan algorithm. The first task was to determine the minimum length of the impulse response analytically. The minimum length is defined as M+1 and the following equation is used to determine M:

$$M = \frac{-10\log(\delta_1\delta_2) - 13}{2.324\Delta\omega} = \frac{-10\log(0.01 \cdot 0.05) - 13}{2.324(0.05\pi)} = 54.814$$

Therefore, the rounded M is 55 and the minimum impulse length (M+1) is 56. Next, the firpmord function of MATLAB was used to generate a filter. The code used to this can be found in the appendix and the frequency response is shown below.

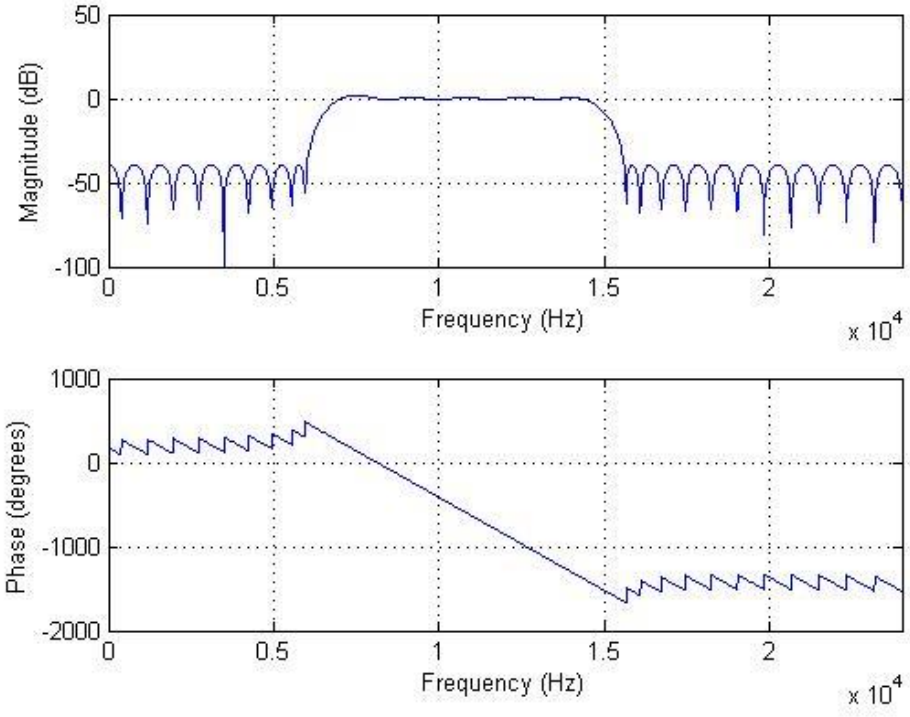

Figure 3: Frequency Response of the PM Bandpass Filter

After performing these steps using individual MATLAB functions, the fdatool was used to design the filter and output the coefficients. A screenshot of the interface with the designed filter is shown below and the coefficients exported from this can be found in the appendix.

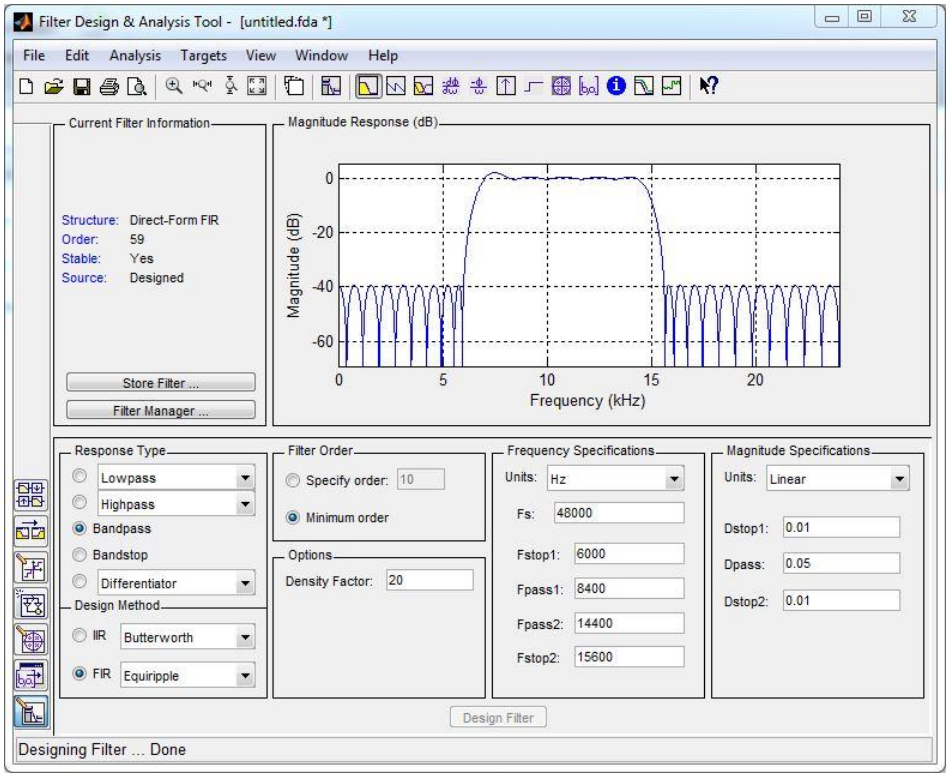

Figure 4: MATLAB fdatool filter

Part 2

Task 1

The first task of part 2 was to use code that implements a filter given an array of coefficients using a linear buffer. The Parks-McClellan coefficients generated using the fdatool as shown above were inserted into the code and the design was tested on the DSP board with a variety of input frequencies. After testing, the filter was found to function as expected. The output of the filter with various input frequencies is shown below.

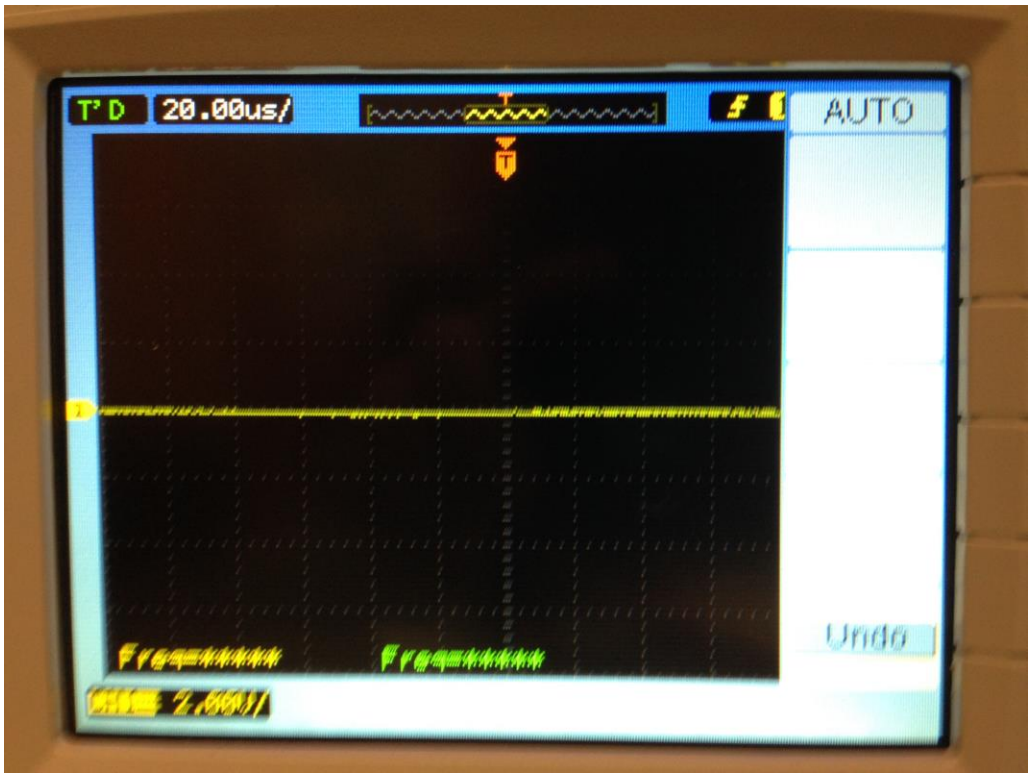

Figure 5: Filter output with 6 kHz input signal

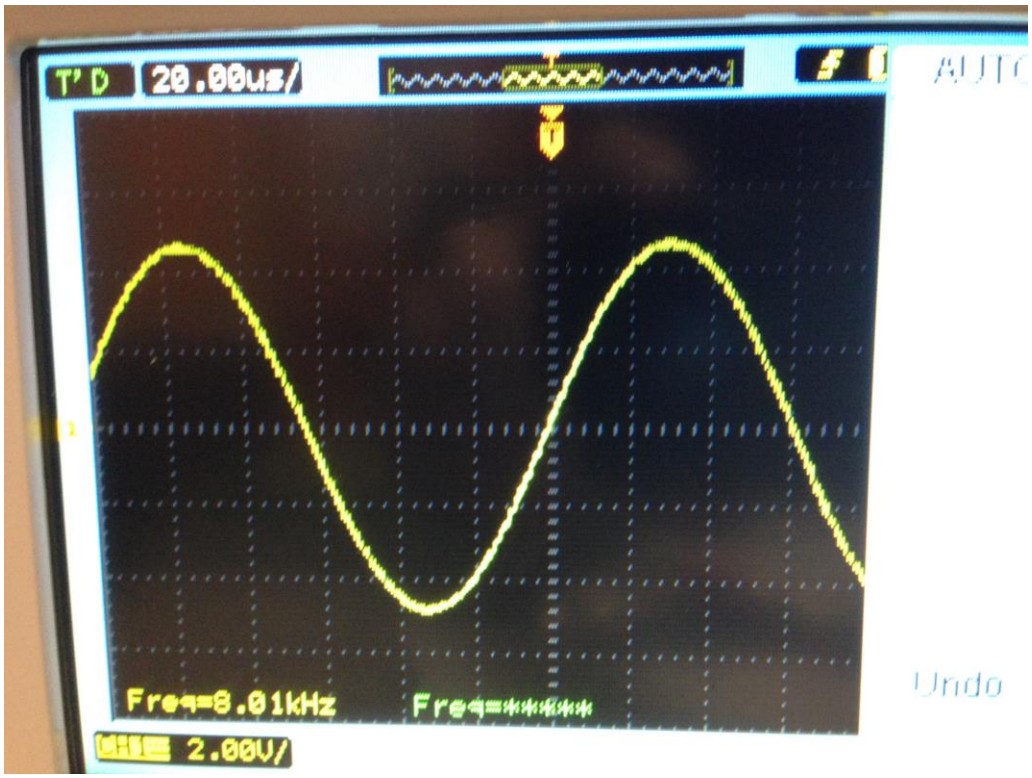

Figure 6: Filter output with 8 kHz input signal

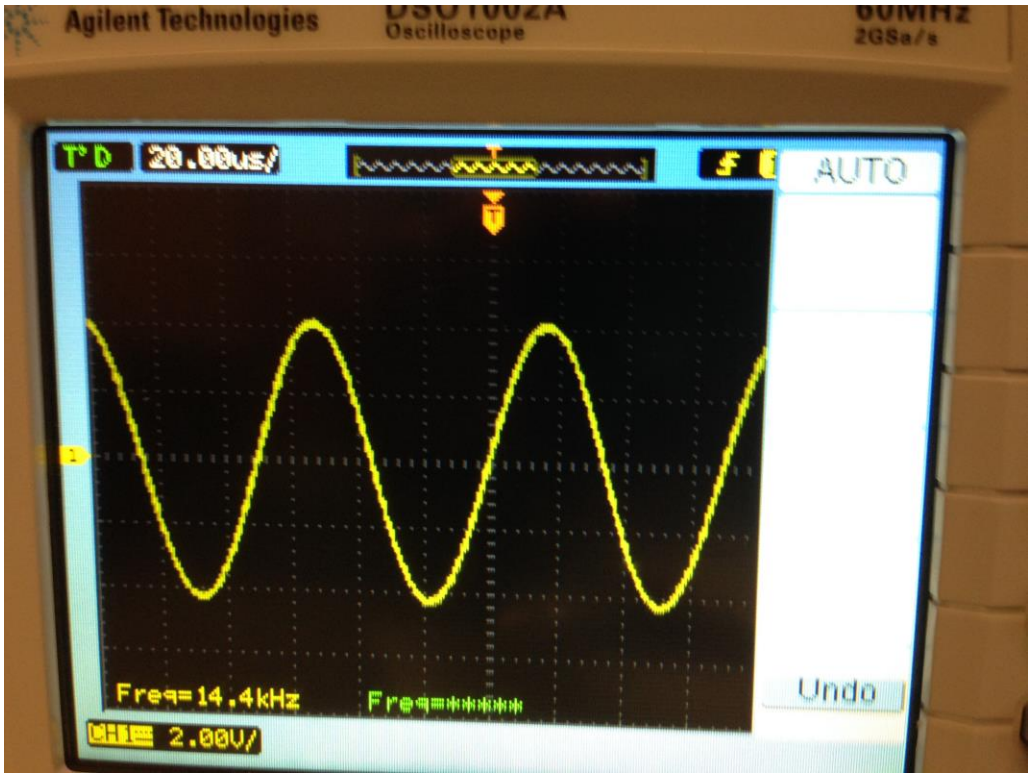

Figure 7: Filter output with 14.4 kHz input signal

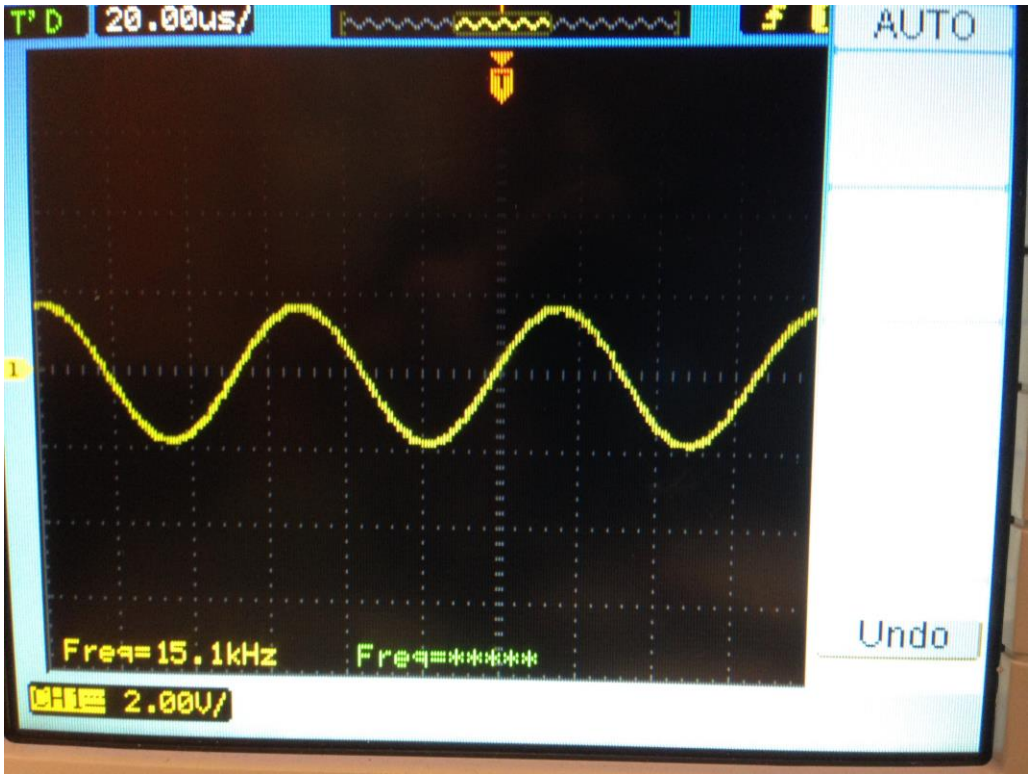

Figure 8: Filter output with 15.1 kHz input signal

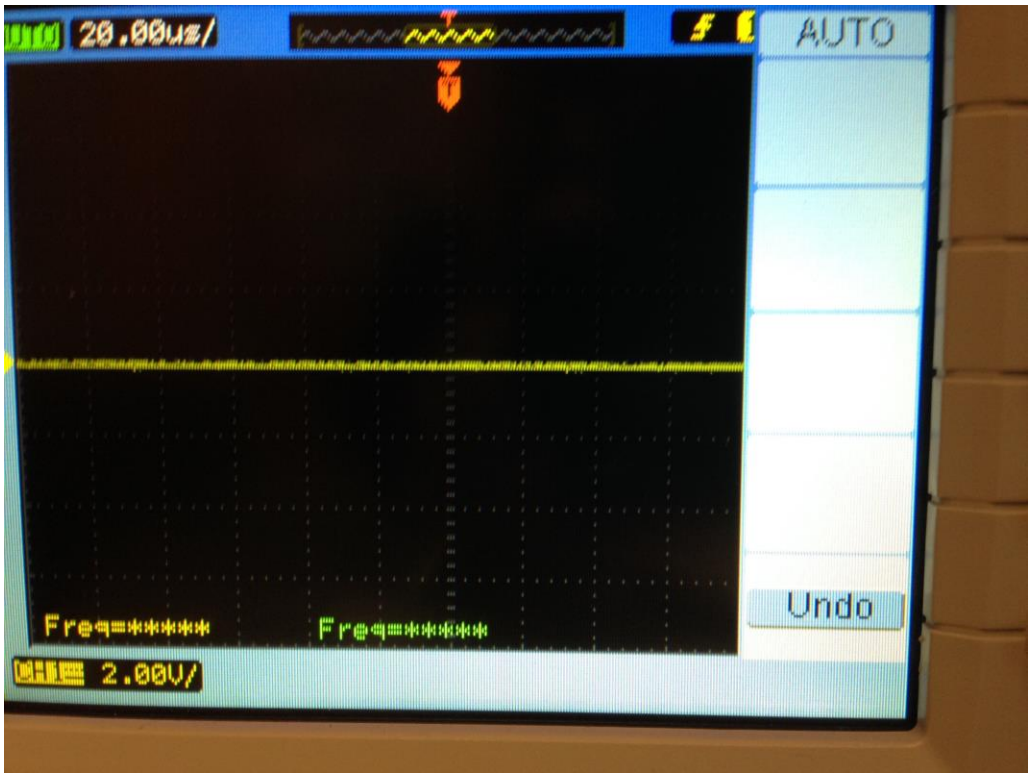

Figure 9: Filter output with 18 kHz input signal

The output shown on the oscilloscope aligns very nicely with the theoretical frequency response. The only differences are the first transition band being a little more restrictive than the design requires since the width of the transition bands is constant and the upper one needs to be more tight and the output voltage of the signal is higher than the input signal voltage. The input signal is 500 mV p-p and the output signal is generally between 5-10 V p-p. This is most likely due to the amplifiers in the output buffers of the codec and does not affect the functionality of the filter since all of the voltage are correct proportionally.

Task 2

Task 2 was to complete a similar activity to task 1, except using circular buffering instead of linear buffering. The circular buffer implementation was compiled and run on the DSP board and tested with the same input frequencies as the first task and was found to still function correctly. The results are shown below:

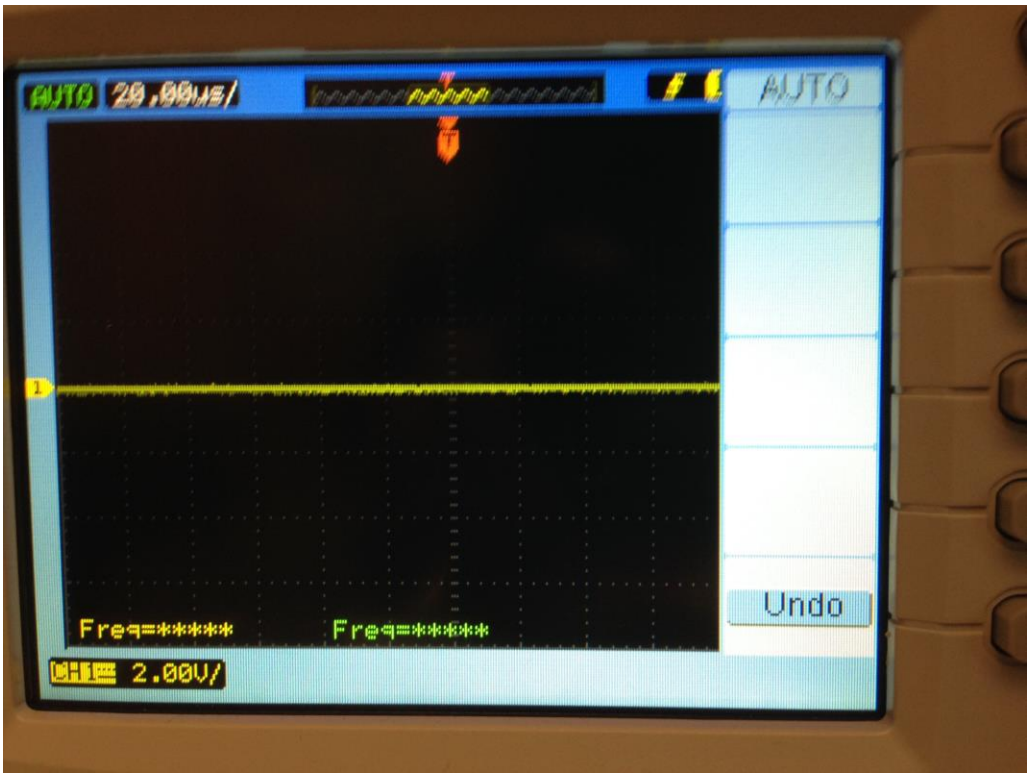

Figure 10: Filter output with 6 kHz input signal

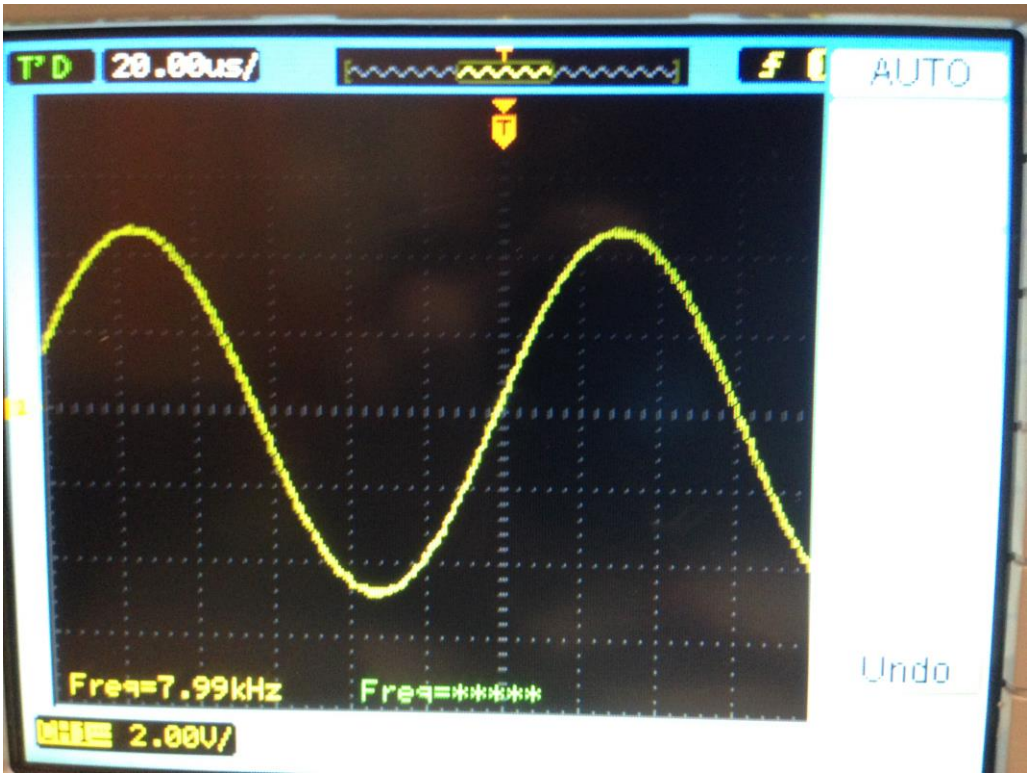

Figure 11: Filter output with 8 kHz input signal

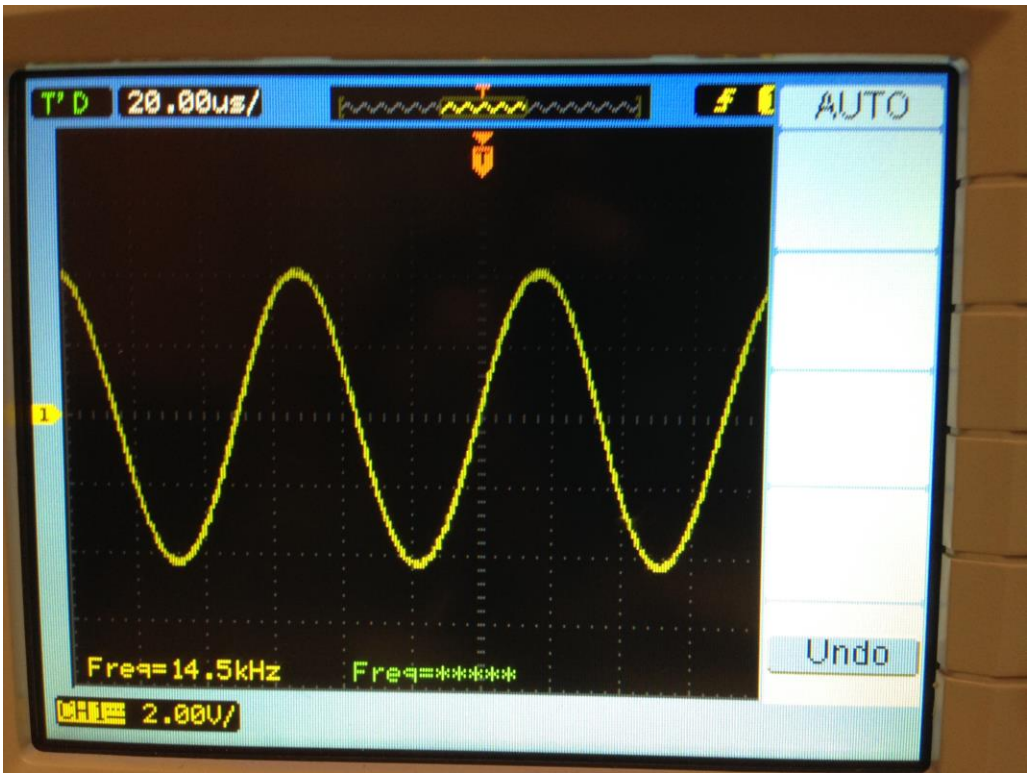

Figure 12: Filter output with 14.4 kHz input signal

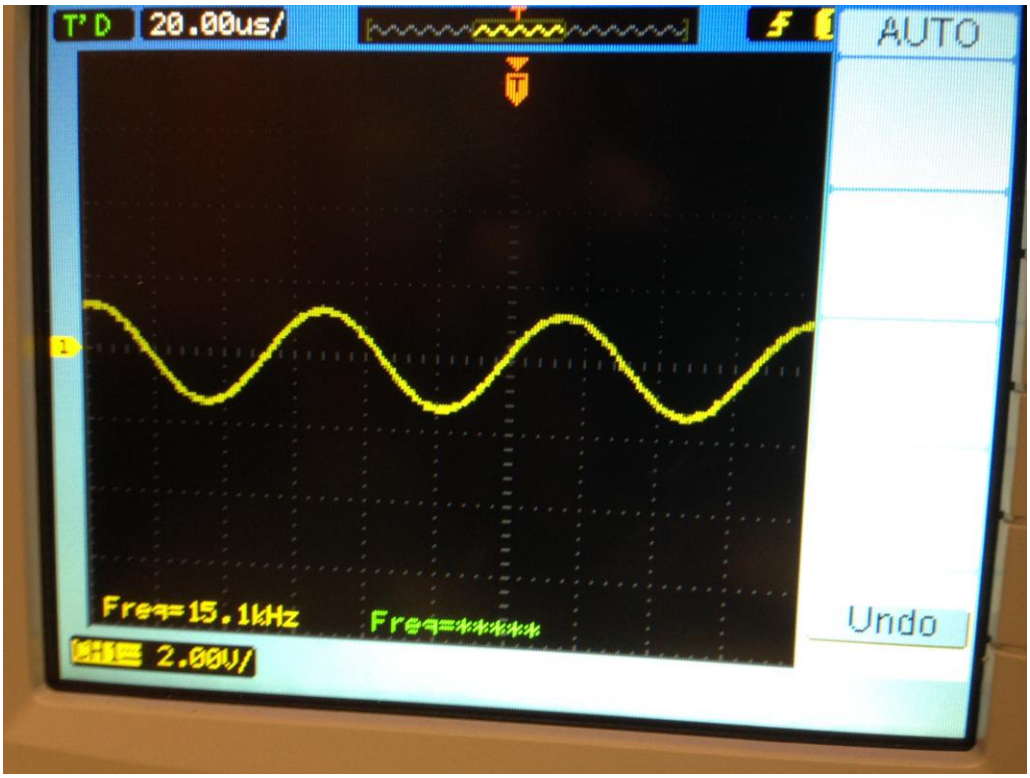

Figure 13: Filter output with 15.1 kHz input signal

Figure 14: Filter output with 18 kHz input signal

This implementation has the same results as the implementation in task 1 and still has the same differences in the magnitude of the output signal. As part of this task, the performance of the linear and circular buffer was compared. With a PM filter with 60 taps, the linear buffer implementation was found to use 4,047 clock cycles to service each interrupt. The circular buffer was found to use 3,597 clock cycles to service each interrupt, an improvement of 11%. These performance metrics were measured with compiler optimization disabled and could be further improved by enabling optimization.

Task 3

Task 3 involved recognizing that the FIR filter coefficients have some symmetry to them. Because of this, the number of iterations and memory accesses required to implement the filter can be reduced, further improving the performance. The linear buffer algorithm was rewritten to take advantage of this property and was tested to verify that the functionality was not changed at all. It was found to take 3,742 clock cycles per interrupt in this implementation. This represents a 7.5% improvement to accomplish the same exact task. The circular buffer implementation was rewritten to take advantage of this property and was then tested to make sure the functionality had not changed and was found to now take 3,167 clock cycles per interrupt, a 12% improvement.

Part 3

For this report, several additional questions needed to be answered. First, how many taps can you have before you run out of time for FIR filtering when using linear buffering? To do this, the maximum amount of allowable time to service each interrupt needed to be found. Because the codec is sampling at 48 kHz, each interrupt needs to finish in less than 20,833 ns. Each clock cycle of the DSP takes 4.44 ns, so the maximum number of clock cycles that each interrupt can take is 4,692 cycles, including any overhead time. Ignoring the overhead time, a 60 tap FIR filter using linear buffering and taking advantage of the symmetry uses 3,742 cycles, or approximately 62 cycles per tap. Using this value, the maximum number of taps was found to be 75. Note that this number of clock cycles is for unoptimized code and could be improved significantly with optimization.

The next question is why circular buffers. Circular buffers eliminate the need to shift the contents of the buffer on ever sample, which reduces the number of memory accesses drastically. Since memory accesses are especially expensive, this can have a significant performance impact.

Finally, a major pro of Kaiser Windows is the relative simplicity of the calculations to find the parameters, but this also leads to their main con, which is the large and varying size of the side nodes. The Parks-McClellan algorithm changes this by taking the opposite approach, it reduces the size of the side nodes and keeps their size constant (hence the name equiripple) but requires much more complex equations that need to be done with a computer instead of by hand.

Conclusion

In this lab, I learned about how to calculate certain parameters for Kaiser Window filter and Parks-McClellan filters, how to use MATLAB tools and functions to find all of the parameters and coefficients, how to implement linear buffer filtering, how to implement circular buffer filtering, and how to improve upon the performance of the filtering by using the symmetry of the coefficients. I feel like I also better understand how to design and use FIR filters in general.

Appendix

Part 1

part_1_a_iv.m

```
%% Kaiserord function
fsamp = 48000;
fcuts = [6000 8400 14400 15600];
mags = [0 1 0];
devs = [0.01 0.05 0.01];
```

```
[n,Wn,beta,ftype] = kaiserord(fcuts,mags,devs,fsamp);
n = n + rem(n,2);   % make sure n is even
window = kaiser(n+1,beta);
hh = fir1(n,Wn,ftype,window,'noscale');
freqz(hh,1,1024,fsamp);
```

part_1_b_ii.m

```
fsamp = 48000;
fcuts = [6000 8400 14400 15600];
mags = [0 1 0];
devs = [0.01 0.05 0.01];

[n,fo,ao,w] = firpmord(fcuts,mags,devs,fsamp);
b = firpm(n,fo,ao,w);
freqz(b,1,1024,fsamp);
```

Common To All Other Tasks

coeff.h

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/* coeff.h                          */
/* FIR filter coefficients          */
/* exported by MATLAB using FIR_DUMP2C */


#define N 59

extern float B[];
```

coeff.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/* coeff.c                          */
/* FIR filter coefficients          */
/* exported by MATLAB using FIR_DUMP2C */

#include "coeff.h"

float B[N+1] = {
-0.0049123250992086183,
-0.014619079916495251,
 0.0060757570503416243,
 0.01952535804916462,
 0.0024331187695853683,
-0.0098746099102310358,
-0.0037963320758576859,
-0.012010346542919192,
-0.01035742425347112,
 0.01992326384135619,
 0.021487608632075595,
```

```
-0.0036831076567773926,
-0.0025285456210165994,
-0.010532837642683505,
-0.034386903509944826,
-0.0011850529678652522,
 0.039757652237396482,
 0.014026994706631228,
 0.0043585313963652086,
 0.014292489095961029,
-0.04562980713703095,
-0.062041754213086692,
 0.02997771785352504,
 0.041487703139491801,
 0.0066936154158850807,
 0.087583480119140356,
 0.030241608546776121,
-0.23544107929841201,
-0.15849062460484048,
 0.26627195921759944,
 0.26627195921759944,
-0.15849062460484048,
-0.23544107929841201,
 0.030241608546776121,
 0.087583480119140356,
 0.0066936154158850807,
 0.041487703139491801,
 0.02997771785352504,
-0.062041754213086692,
-0.04562980713703095,
 0.014292489095961029,
 0.0043585313963652086,
 0.014026994706631228,
 0.039757652237396482,
-0.0011850529678652522,
-0.034386903509944826,
-0.010532837642683505,
-0.0025285456210165994,
-0.0036831076567773926,
 0.021487608632075595,
 0.01992326384135619,
-0.01035742425347112,
-0.012010346542919192,
-0.0037963320758576859,
-0.0098746099102310358,
 0.0024331187695853683,
 0.01952535804916462,
 0.0060757570503416243,
-0.014619079916495251,
-0.0049123250992086183,
};
```

Task 1 – FIRmono_ISRs.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011
```

```
////////////////////////////////////////////////////////////////////////
// Filename: FIRmono_ISRs.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "coeff.h"   // load the filter coefficients, B[n] ... extern

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
       Uint32 UINT;
       Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
float xLeft[N+1];
float yLeft;
Int32 i;


interrupt void Codec_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       /* add any local variables here */


       if(CheckForOverrun())                                  // overrun error occurred
(i.e. halted DSP)
              return;                                                    // so serial
port is reset to recover

       CodecDataIn.UINT = ReadCodecData();             // get input data samples

       /* I added my FIR filter routine here */
       xLeft[0]  = CodecDataIn.Channel[LEFT];   // current LEFT input value
       yLeft = 0;                          // initialize the output value

       for (i = 0; i <= N; i++) {
```

```
		yLeft += xLeft[i]*B[i];			// perform the dot-product
	}

	for (i = N; i > 0; i--) {
		xLeft[i] = xLeft[i-1];			// shift for the next input
	}

	CodecDataOut.Channel[LEFT]  = yLeft;	// output the value
	CodecDataOut.Channel[RIGHT] = yLeft;
	/* end of my routine */


	WriteCodecData(CodecDataOut.UINT);	// send output data to  port
}
```

Task 2 – FIRmono_ISRs.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

///////////////////////////////////////////////////////////////////////
// Filename: ISRs.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
///////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "coeff.h"   // load the filter coefficients, B[n] ... extern

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
	Uint32 UINT;
	Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
float xLeft[N+1], *pLeft = xLeft;
Int32 i;


interrupt void Codec_ISR()
///////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
```

```
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
       /* add any local variables here */
       float output, *p;

       if(CheckForOverrun())                                    // overrun error occurred
(i.e. halted DSP)
              return;                                                       // so serial
port is reset to recover

       CodecDataIn.UINT = ReadCodecData();             // get input data samples

       /* I added my mono FIR filter routine here */
       *pLeft  = CodecDataIn.Channel[LEFT];     // store LEFT input value

       output = 0;                                                          // set up for LEFT
channel
       p = pLeft;                                                           // save current
sample pointer
       if(++pLeft > &xLeft[N])                                  // update pointer, wrap if
necessary
              pLeft = xLeft;                                                // and store
       for (i = 0; i <= N; i++) {                       // do LEFT channel FIR
              output += *p-- * B[i];            // multiply and accumulate
              if(p < &xLeft[0])                 // check for pointer wrap around
                 p = &xLeft[N];
       }

       CodecDataOut.Channel[LEFT]  = output; // store filtered value
       CodecDataOut.Channel[RIGHT] = output; // store filtered value
       /* end of my mono FIR filter routine */

       WriteCodecData(CodecDataOut.UINT);       // send output data to  port
}
```

Task 3

FIRmono_ISRs.c – Linear

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

////////////////////////////////////////////////////////////////////////
// Filename: FIRmono_ISRs.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "coeff.h"   // load the filter coefficients, B[n] ... extern

// Data is received as 2 16-bit words (left/right) packed into one
```

```
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
	Uint32 UINT;
	Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
float xLeft[N+1];
float yLeft;
Int32 i;


interrupt void Codec_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
////////////////////////////////////////////////////////////////////////
{
	/* add any local variables here */


	if(CheckForOverrun())                                   // overrun error occurred
(i.e. halted DSP)
		return;                                                         // so serial
port is reset to recover

	CodecDataIn.UINT = ReadCodecData();             // get input data samples

	/* I added my FIR filter routine here */
	xLeft[0]  = CodecDataIn.Channel[LEFT];    // current LEFT input value
	yLeft = 0;                          // initialize the output value

	for (i = 0; i < (N+1)/2; i++) {
		yLeft += xLeft[i]*B[i];
		yLeft += xLeft[N-i]*B[i];          // perform the dot-product
	}

	if( ( N % 2 ) == 0 )
	{
		yLeft += ( xLeft[ ( N/2 )] * B[ ( N/2 )] );
	}


	for (i = N; i > 0; i--) {
```

```
            xLeft[i] = xLeft[i-1];             // shift for the next input
        }

        CodecDataOut.Channel[LEFT]  = yLeft;      // output the value
        /* end of my routine */


        WriteCodecData(CodecDataOut.UINT);        // send output data to  port
}
```

FIRmono_ISRs.c – Circular

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

////////////////////////////////////////////////////////////////////////
// Filename: ISRs.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "coeff.h"   // load the filter coefficients, B[n] ... extern

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
        Uint32 UINT;
        Int16 Channel[2];
} CodecDataIn, CodecDataOut;


/* add any global variables here */
float xLeft[N+1]; //, *pLeft = xLeft;
Int32 i;
int sampleInd = 0;



interrupt void Codec_ISR()
////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
```

```
////////////////////////////////////////////////////////////////////////
{
	/* add any local variables here */
	float output; //, *p;
	int indLeft, indRight;

	if(CheckForOverrun())                                  // overrun error occurred
(i.e. halted DSP)
		return;                                                    // so serial
port is reset to recover

	CodecDataIn.UINT = ReadCodecData();              // get input data samples

	/* I added my mono FIR filter routine here */
	xLeft[sampleInd] = CodecDataIn.Channel[LEFT];

	output = 0;                                                // set up for LEFT
channel
	indLeft = sampleInd;
	indRight = sampleInd + 1;
	if(++sampleInd > N)
		sampleInd = 0;
	if(indRight > N)
		indRight = 0;
	for (i = 0; i < (N+1)/2; i++) {                        // do LEFT channel FIR
			output += ( ( xLeft[indLeft--] * B[i] ) + ( xLeft[indRight++] * B[i]
) );
			if(indLeft < 0)
				indLeft = N;
			if(indRight > N)
				indRight = 0;
	}

	if( ( N % 2 ) == 0 )
	{
		output += ( xLeft[ indLeft-- ] * B[ ( N/2 )] );
	}

	CodecDataOut.Channel[LEFT]  = output; // store filtered value
	CodecDataOut.Channel[RIGHT] = output; // store filtered value
	/* end of my mono FIR filter routine */

	WriteCodecData(CodecDataOut.UINT);       // send output data to  port
}
```