

# REAL-TIME DSP LABORATORY4:

## Design and Implementation of Finite Impulse Response (FIR) Filters on the C6713 DSK

### Contents

1	Introduction	1
1.1	Discrete-Time Digital Filters . . . . .	1
1.2	Causal Linear Phase FIR Filters . . . . .	4
2	FIR Filter Design Methods	5
2.1	Designing a Lowpass Filter Using Fourier Methods . . . . .	6
2.2	Windowing Methods for FIR Filter Design . . . . .	8
2.3	Specific FIR Filter Designs Using Windowing Methods . . . . .	10
2.3.1	Example: Discrete-Time Differentiator . . . . .	13
2.3.2	Example: Discrete-Time Hilbert Transformers . . . . .	13
2.4	Zero Placements in a Transfer Function . . . . .	14
3	FIR Filter Design Using MATLAB SPTOOL	14
4	FIR Implementation on the C6713 DSK Using C and Assembly	17
4.1	FIR Filter Design using MATLAB - Equiripple FIR * . . . . .	19
4.2	FIR Filter Design - Notch Filter * . . . . .	21
4.3	FIR Filter Design - Zero Placement * . . . . .	22
4.4	FIR Filter Design - Delay * . . . . .	22
4.5	FIR Filter Design - Differentiator * . . . . .	23
4.6	FIR Filter Design - Hilbert Transformer * . . . . .	23
4.7	C Callable Assembly Code for Implementing FIR Filters * . . . . .	24
4.8	Circular Buffers on the TMS320C6713 . . . . .	25
5	End Notes	29

## 1 Introduction

Digital filtering is a generic term for linear, shift-invariant filtering using time domain convolution or frequency domain multiplication. The basic arithmetic is multiply and accumulate.

Figure 1 illustrates a *Finite Duration Impulse Response* (FIR) filter.

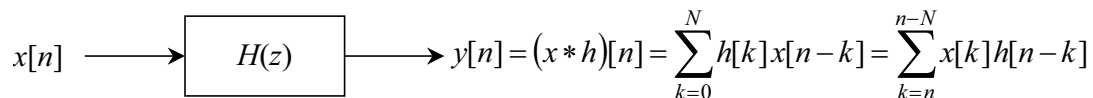


Figure 1: Discrete-Time System

Here, the output of the FIR filter at time  $n$  is a linear combination of delayed unit pulse responses, scaled by the input. In hardware, this means that in order to generate an output in real time, we must multiply two values and add them to a sum (accumulate). This is exactly what the

DSP chip is designed to do. In fact, the ability to multiply and accumulate very fast is one of the main differences between a DSP chip and the CPU found in a personal computer. The filters that we design in the next two labs will be useful later when we implement communication systems on DSP hardware. As a consequence, we will want these filters to operate at a high sampling rate. One way to make these filters execute their adds and multiplies at high rate is to code them in DSP assembly language. In this lab, you will study

- discrete-time digital filters,
- FIR filter design, and
- FIR filter implementation on the DSK, using C and assembly language programs.

## 1.1 Discrete-Time Digital Filters

The output of a general linear and shift-invariant (LSI) discrete-time system is described by either

$$y[n] = \sum_{k=-\infty}^{\infty} h[k]x[n-k] \quad \text{or} \quad Y(z) = H(z)X(z), \quad (1)$$

where  $X(z)$  is the Z-transform of the input sequence  $\{x[n]\}$ ,  $Y(z)$  is the Z-transform of the output sequence  $\{y[n]\}$ , and  $H(z)$  is the Z-transform of the filter impulse response sequence  $\{h[n]\}$ . The first equation in eqn(1) describes exactly what the filter does and the second is an algebraic code for what is done. When evaluated at  $z = e^{j\theta(t)}$ , the second equation shows how the spectrum of the input, namely  $X(e^{j\theta(t)})$ , is shaped by the complex frequency response,  $H(e^{j\theta(t)})$ , to determine the spectrum of the output, namely  $Y(e^{j\theta(t)})$ . Our shorthand way of talking about eqn(1) is to say convolution in time is the equivalent of multiplication in frequency. In digital signal processing, we design  $H(z)$  to meet frequency response specifications.

For an FIR filter of length  $N$ , with input  $x[n]$ , the output  $y[n]$  is described by the difference equation

$$y[n] = h[0]x[n] + h[1]x[n-1] + \dots + h[N]x[n-N] = \sum_{k=0}^N h[k]x[n-k] \quad (2)$$

The first part of eqn(2) illustrates that the output at time  $n$  is a weighted combination of the  $N+1$  most recent inputs, where the weights  $h[k]$  are the filter coefficients. The right side of eqn(2) illustrates that the output at time  $n$  is the convolution of the finite length impulse response of the filter with the input. This is the finite impulse response version of eqn(1).

The Z-transform of  $h[n]$  is

$$h[n] \longleftrightarrow H(z) = \sum_{k=0}^N h[k]z^{-k} \quad (3)$$

The response of the filter  $H(z)$  to the complex exponential input  $\{x[n] = e^{jn\theta}\}$  is  $y[n] = H(e^{j\theta})e^{jn\theta}$ , where  $H(e^{j\theta})$  is the *Discrete Time Fourier Transform* (DTFT) of  $\{h[n]\}$ :

$$h[n] \longleftrightarrow H(e^{j\theta}) = \sum_{k=0}^N h[k]e^{-jk\theta} \quad (4)$$

Thus  $H(e^{j\theta})$ , a complex frequency response, illustrates how the input is complex scaled for each value of  $\theta$  on the Nyquist band  $(-\pi < \theta \leq \pi)$ .

The rotating phasor  $e^{j\theta}$  is  $2\pi$ -periodic in  $\theta$  and therefore so is  $H(e^{j\theta})$ . This is illustrated in Figure 2.

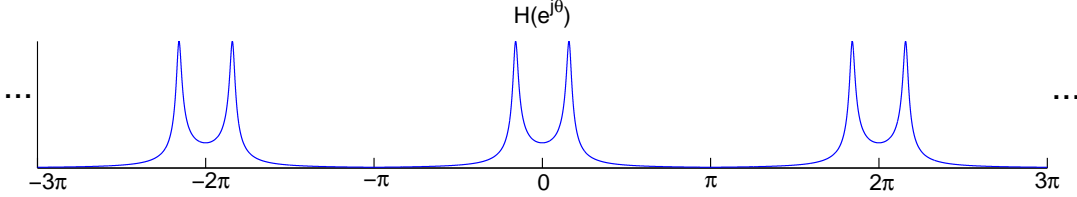


Figure 2: Discrete-Time Fourier Transform  $H(e^{j\theta})$

The DTFT of a discrete-time signal is  $2\pi$  periodic in  $\theta$  which means there are aliases that appear at multiples of  $2\pi$ . These aliasing effects are seen in the DTFT, or complex frequency response, of any digital filter that we design. We design digital filters on the Nyquist band, knowing that these characteristics will be aliased every  $2\pi$  radians.

This representation describes the frequency response of a filter on the Nyquist band, but it does not give any information about the computation rate of the filter. In real-time DSP applications, this rate is known and it must be taken into account. To account for this, we let  $\theta = \omega t_o$ , where  $t_o$  is the time duration between samples of the impulse response  $h[n]$ . In the case of the onboard codec, the incoming signal  $x(t)$  is sampled at  $t_s = .125\text{ms}$  to generate  $x[n]$ . For this, we would design a filter that is running at rate  $t_o = t_s$ . Usually, we will design a filter  $H(z)$  that is running at the same rate as our incoming sampled data ( $t_o = t_s$  where  $t_s$  is the sampling period of the codec). In multi-rate systems, this would not be the case.

If the filter  $H(z)$  is excited by the sampled data sequence  $x[n] = x(nt_o)$ , where  $x(t) = e^{j\omega t}$ , then the output is

$$\begin{aligned} y[n] &= \sum_{k=0}^N h[k]e^{j(n-k)\omega t_o} = e^{jn\omega t_o} \sum_{k=0}^N h[k]e^{-jk\omega t_o} \\ &= e^{jn\omega t_o} H(e^{j\omega t_o}), \end{aligned} \quad (5)$$

where

$$H(e^{j\omega t_o}) = \sum_{k=0}^N h[k]e^{-jk\omega t_o} \quad (6)$$

The function  $H(e^{j\omega t_o})$  is the complex frequency response of the filter  $H(z)$  to a continuous-time complex exponential that has been sampled at rate  $f_o = 1/t_o$ . Of course, it is just the DTFT of  $\{h[n]\}_0^N$  evaluated at  $\theta = \omega t_o$ . When plotted versus radian frequency  $\omega$ ,  $H(e^{j\omega t_o})$  is the complex frequency response, or Bode plot, of the filter. See Figure 3.

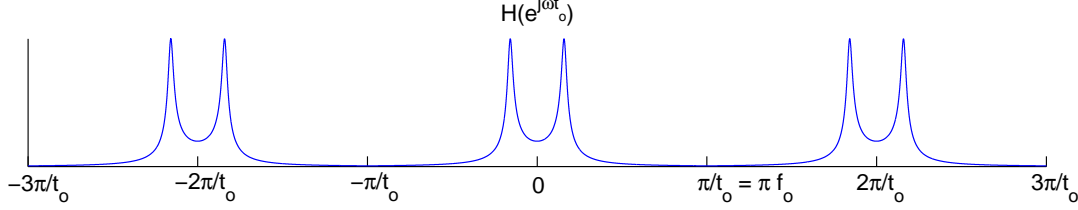


Figure 3: Discrete-Time Fourier Transform  $H(e^{j\omega t_o})$

The only difference between Figures 2 and 3 is in the scaling of the variable  $\theta = \omega t_o$ , and the corresponding scaling of the Nyquist band. The Nyquist band is now  $-\pi/t_o$  to  $\pi/t_o$  in radians/second and is dependent upon the sample period  $t_o$ . As  $t_o$  (time duration between samples) becomes smaller, the sample rate ( $f_o = 1/t_o$ ) will become larger, which means that the Nyquist band will cover a larger frequency range. As the Nyquist band increases, the frequency range over which the filter may be applied will increase. Again, the sample rate of the filter ( $f_o$ ) will generally be equal to the sampling rate of the codec ( $f_s$ ).

On the Nyquist band, we design digital filters that have certain magnitude and phase characteristics. For a magnitude response, we want to amplify certain frequencies while attenuating others. In this lab we will study filters with the following magnitude responses: lowpass, high-pass, bandpass, and bandstop.

When the input to a real filter is real, then the output is real. Consider a special case of eqn(5) when a real filter  $H(z)$  is excited by the real sampled data sequence  $x[n] = x(nt_o)$ , where  $x(t) = 2 \cos(\omega t) = e^{j\omega t} + e^{-j\omega t}$ , then the output is

$$\begin{aligned} y[n] &= \sum_{k=0}^N h[k] e^{j(n-k)\omega t_o} + \sum_{k=0}^N h[k] e^{-j(n-k)\omega t_o} \\ &= e^{jn\omega t_o} \sum_{k=0}^N h[k] e^{-jk\omega t_o} + e^{-jn\omega t_o} \sum_{k=0}^N \overline{h[k] e^{-jk\omega t_o}} \end{aligned} \quad (7)$$

$$\begin{aligned} &= e^{jn\omega t_o} H(e^{j\omega t_o}) + e^{-jn\omega t_o} \overline{H(e^{j\omega t_o})} \\ &= 2 |H(e^{j\omega t_o})| \cos(n\omega t_o + \arg H(e^{j\omega t_o})), \end{aligned} \quad (8)$$

where eqn(7) uses the fact that  $H(z)$  has a real impulse response (i.e.  $h[k] = \overline{h[k]}$ ). Both eqn(5) and eqn(8) give a fundamental insight into signal processing and linear shift-invariant filtering: sending a periodic signal at frequency  $\omega$  through the LSI system  $H(e^{j\omega t_o})$  will produce a signal at the same frequency  $\omega$  that has been scaled and phased by the complex frequency response  $H(e^{j\omega t_o})$ . This is the reason that we use Fourier analysis to analyze signals and systems in the frequency domain.

The phase response of a digital filter,  $\arg H(e^{j\omega t_o})$  in eqn(8), shows how signals will be phased (a little more general concept than time delay) at each frequency. The *phase delay* at each frequency is given by  $\frac{\arg H(e^{j\omega t_o})}{\omega}$ <sup>1</sup>. For no phase distortion, the phase delay must be constant (i.e.  $\arg H(e^{j\omega t_o}) = \alpha\omega t_o$ , where  $\alpha$  is the delay (in samples) for sinusoids at every frequency. We say that  $\alpha$  is the filter latency (in samples) of a phase distortionless, and that its associated time delay is  $\alpha t_o$  seconds. The property  $\arg H(e^{j\omega t_o}) = \alpha\omega t_o$  is the linear phase property that all of the lowpass, highpass, bandpass, and bandstop FIR filters that we design will have.

## 1.2 Causal Linear Phase FIR Filters

FIR filters are used in applications where a linear phase is required. General linear phase is when  $\arg H(e^{j\omega t_o}) = \alpha\omega t_o + \beta$ . It is only when  $\beta = 0$  that the filter will have no phase distortion<sup>2</sup>.

The linear phase of FIR filters is an artifact of the design process. A classical Fourier series (FS) FIR filter design starts with designing an ideal filter with a given  $\beta$ , usually either  $\beta = 0$  or  $\beta = \pm\frac{\pi}{2}$ . For the case  $\beta = 0$ , the ideal filter response  $H(e^{j\omega t_o})$  will be purely *real*, so the impulse response will have *even symmetry* in time. That is, if a signal is real in one domain, it will be Hermitian in the other<sup>3</sup>. In this case, real and even in the time domain means that it is Hermitian in the time domain and is therefore real in the frequency domain. In the case  $\beta = \pm\frac{\pi}{2}$ , the ideal filter response  $H(e^{j\omega t_o})$  will be purely *imaginary*, so the impulse response will have *odd symmetry* in time. These are the two general cases we will use to design causal FIR filters. Once this ideal filter (with either an even or odd impulse response) has been designed, the filter impulse response is truncated using a windowing function to make the filter impulse response finite-duration. This finite-duration impulse response is then delayed by  $\alpha$  samples to make the impulse response  $h[n]$  causal, where  $h[n] = 0$ , for  $n < 0$ . This delay of  $\alpha$  samples accounts for the  $\alpha\omega t_o$  part of the phase response  $\arg H(e^{j\omega t_o}) = \alpha\omega t_o + \beta$  of the causal linear phase FIR filter that is implemented in hardware. NB: Remember from signals and systems that if  $h[n] \longleftrightarrow H(e^{j\omega t_o})$ , then  $h[n - \alpha] \longleftrightarrow e^{j\alpha\omega t_o} H(e^{j\omega t_o})$ .

In this discussion, the causal FIR filter impulse response must have symmetry about the point  $N/2$ . When the order of the filter, namely  $N$ , is an odd integer,  $N/2$  will be a point exactly between two samples. This is perfectly acceptable for guaranteeing a linear phase, except that it is not intuitive to design ideal filters whose impulse response is defined for half-sample points, which would be required in the FS design method. To develop a more intuitive way of designing odd order filters, we will design ideal bandlimited analog filters, create a finite support analog impulse response using windowing, delay in time, and then sampling using impulse invariance to get a causal FIR filter. This method will allow us to design both even and odd order FIR filters without the ambiguity of where the samples of the ideal (un-delayed) impulse response align in time. A point to be clarified in the section on windowing.

---

<sup>1</sup>The phase delay of a filter at a given frequency is the time delay within one period of a sinusoid at that frequency.

<sup>2</sup>One could argue that when the magnitude response switches polarity that the phase response will have  $\pi$  added to it, creating a  $\beta = \pi$  at some frequencies. However, if we unwrap the phase response, these jumps of  $\pi$  will go away.

<sup>3</sup>Hermitian symmetry means that  $H(e^{j\omega t_o})$  has even magnitude ( $|H(e^{j\omega t_o})| = |H(e^{-j\omega t_o})|$ ) and odd phase ( $\arg H(e^{j\omega t_o}) = -\arg H(e^{-j\omega t_o})$ ), or  $H(e^{-j\omega t_o}) = \overline{H(e^{j\omega t_o})}$ . This relationship is analogous for all Fourier transform pairs.

## 2 FIR Filter Design Methods

A simple way to design causal digital linear phase FIR filters is to use a Fourier Series (FS) approximation of a desired frequency response.

In continuous time, when a signal is  $T$ -periodic,  $x(t) = x(t + T)$  for all  $t$ , it has a (discrete) Fourier series  $X[n]$ . The formulas that relate  $x(t)$  and  $X[n]$  are

$$x(t) = \sum_{n=-\infty}^{\infty} X[n]e^{jn\omega_o t} \quad (\text{synthesis}) \quad (9)$$

$$X[n] = \frac{1}{T} \int_0^T x(t)e^{-jn\omega_o t} dt \quad (\text{analysis}) \quad (10)$$

In this representation,  $\omega_o = \frac{2\pi}{T}$  is the fundamental frequency in rad/sec,  $T$  is the fundamental period in seconds, and the values  $X[n]$  are referred to as the *Fourier Series coefficients*. Even if  $x(t)$  has an infinite number of FS coefficients, it can usually be well-approximated by a finite number of its FS coefficients. It is this idea of approximating a function by taking a finite sum over some of its FS coefficients that gives rise to the FS method of designing FIR filters. Examples of this along with a complete treatment of periodic signals and Fourier series can be found in [3].

In the FS approach to FIR digital filter design, the continuous periodic frequency response  $H(e^{j\omega t_o})$  takes the place of the the periodic continuous-time signal  $x(t)$  and the impulse response  $h[n]$  takes the place of the discrete FS coefficients. The DTFT pair  $h[n] \longleftrightarrow H(e^{j\omega t_o})$  is described by the equations

$$h[n] = \frac{t_o}{2\pi} \int_{-\pi/t_o}^{\pi/t_o} H(e^{j\omega t_o}) e^{jn\omega t_o} d\omega \quad (\text{synthesis}) \quad (11)$$

$$H(e^{j\omega t_o}) = \sum_{n=-\infty}^{\infty} h[n] e^{-jn\omega t_o} \quad (\text{analysis}) \quad (12)$$

The idea now is to approximate an ideal frequency response with a finite number of filter coefficients  $h[n]$ . By comparing eqn (9) and eqn (10) with eqn (11) and eqn (12), it is easy to see that these equations are duals of each other.

For real filter coefficients  $h[n]$ ,  $H(e^{j\omega t_o})$  must have Hermitian symmetry. When we design linear phase FIR filters, we force  $H(e^{j\omega t_o})$  to be either purely real or purely imaginary so that our filter will have either an even or odd impulse response, respectively. Once we have our impulse response, we will *window* it (truncate it to make it finite) and then delay it so that it is purely causal, which gives us an FIR filter with linear phase.

### 2.1 Designing a Lowpass Filter Using Fourier Methods

Consider the ideal lowpass filter (LPF) in Figure 4.

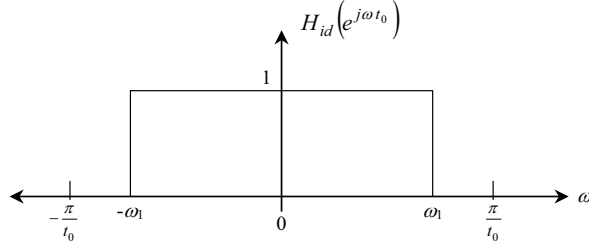


Figure 4: Ideal LPF  $H_{id}(e^{j\omega t_o})$

The filter coefficients (impulse response) of this filter are

$$\begin{aligned}
 h_{id}[n] &= \frac{t_o}{2\pi} \int_{-\pi/t_o}^{\pi/t_o} H_{id}(e^{j\omega t_o}) e^{-jn\omega t_o} d\omega \\
 &= \frac{t_o}{2\pi} \int_{-\omega_1}^{\omega_1} e^{-jn\omega t_o} d\omega, \quad \omega_1 < \frac{\pi}{t_o} \\
 &= \frac{1}{\pi n} \frac{e^{jn\omega_1 t_o} - e^{-jn\omega_1 t_o}}{2j} \\
 &= \frac{1}{\pi n} \sin[\omega_1 t_o n] \\
 &= \frac{\omega_1 t_o}{\pi} \text{sinc}[\omega_1 t_o n], \quad \omega_1 < \frac{\pi}{t_o} \tag{13}
 \end{aligned}$$

$$= \frac{2f_1}{f_o} \text{sinc}\left[\frac{2\pi f_1}{f_o} n\right], \quad f_1 < \frac{f_o}{2} = \frac{1}{2t_o} \tag{14}$$

The last two equations, eqn (13) and eqn (14), designate the cutoff frequency in radians per second and Hertz respectively. Here, the sinc( $\cdot$ ) function is defined to be  $\text{sinc}(x) = \sin(x)/x$ . The cutoff frequency must be less than half the rate of the filter ( $f_o/2$ ). Since  $H_{id}(e^{j\omega t_o})$  is real, the impulse response  $h_{id}[n]$  is even. Also,  $h_{id}[n]$  is an infinite sequence. Before this can be implemented, the impulse response must be truncated in a manner that preserves the even symmetry. There are many ways to truncate this response which we will explore next in windowing. For now, let's define the truncated sequence to be

$$h_{tn}[n] = \begin{cases} h_{id}[n] & : |n| \leq \frac{N}{2} \\ 0 & : \text{otherwise} \end{cases}, \tag{15}$$

where  $N$  is a positive *even* integer. The final step is to make this filter causal by delaying it  $N/2$  (an integer number of) samples in time. Our causal linear phase FIR lowpass filter will have the following impulse response

$$h[n] = h_{tn}\left[n - \frac{N}{2}\right] = \begin{cases} \frac{2f_1}{f_o} \text{sinc}\left(\frac{2\pi f_1}{f_o} \left(n - \frac{N}{2}\right)\right) & : 0 \leq n \leq N \\ 0 & : \text{otherwise} \end{cases} \tag{16}$$

The impulse response,  $h[n]$ , will have even symmetry ( $h[n] = h[n - N]$ ) about the point  $N/2$ . This symmetry will guarantee that the FIR filter will have a constant phase delay (no phase distortion), which is the desired property of the designed filter. It can be argued that when  $N$  is a positive *odd* integer, that eqn (16) will also produce a linear phase FIR filter. This may be justified by the fact that the samples of  $h[n]$  are calculated by evaluation the continuous-time function  $\text{sinc}(\cdot)$ . Therefore, to get a non-integer delay, one could use bandlimited construction to create the continuous-time function  $\text{sinc}(\cdot)$ , delay it by  $t_o/2$  seconds, and then re-sample to get a digital signal with a non-integer delay  $n - \frac{1}{2}$ . This is a confusing point that we will circumvent by designing analog filters, truncating their continuous-time impulse, and sampling to generate a discrete-time FIR filter. A point to be clarified in the next section.

## 2.2 Windowing Methods for FIR Filter Design

Windowing is multiplying an (infinite-duration) ideal filter impulse response  $h_{id}(t)$  with a finite-duration windowing function. To gain insight into this idea, let's consider designing an ideal *bandlimited* analog filter, windowing, delaying, and then sampling its impulse response to create a discrete-time FIR filter. Specifically, this filter must be bandlimited to the Nyquist band ( $-\frac{\pi}{t_o} < \omega < \frac{\pi}{t_o}$ , where  $t_o$  is the sample rate of the DSP system). This will allow us to create a digital filter that greatly minimizes the affects of aliasing.

To begin, we design an ideal analog filter  $H_{id}(j\omega)$  that has a desired frequency response on the band  $-\frac{\pi}{t_o} < \omega < \frac{\pi}{t_o}$  and is zero outside this band. This will lead to a filter whose impulse response is infinite in duration with either even or odd symmetry. Then, this impulse response is truncated by multiplying it (in the time domain) by an evenly symmetric windowing function. Next, the truncated impulse response is delayed in time to make it causal (equal to zero for time less than zero). This time delay is chosen to be the minimum time delay required to make the truncated impulse response causal, which means that the time delay will be half the duration of the windowing function. At this point, our finite duration analog filter is essentially bandlimited to half the sample rate of the DSP system and is causal. Therefore, we can sample this impulse response to create the causal discrete-time impulse response of our digital filter. This idea is illustrated more carefully in the next discussion.

In the LPF example previously, our windowing function was a *rectangular window*. Our associated analog filter had impulse response

$$h_{id}(t) = \frac{\omega_1}{\pi} \text{sinc}(\omega_1 t) \quad (17)$$

and the windowing function was

$$w(t) = \begin{cases} 1, & -\frac{N}{2}t_o \leq t \leq \frac{N}{2}t_o \\ 0, & \text{otherwise} \end{cases}, \quad (18)$$

where  $N$  is the filter order.

This gives us our windowed impulse response  $h_w(t) = h_{id}(t)w(t)$  which has frequency response



$$H_w(j\omega) = H_{id}(j\omega) * W(j\omega) = \int_{-\infty}^{\infty} H_{id}(j\nu) W(j(\omega - \nu)) \frac{d\nu}{2\pi}, \quad (19)$$

where  $W(j\omega) = Nt_o \text{sinc}(\frac{N\omega t_o}{2})$ . This frequency response is NOT bandlimited, but for large  $N$  will nearly be so.

This finite-duration impulse response  $h_w(t)$  is made causal by delaying this impulse response by  $\frac{N}{2}t_o$  seconds to create  $h(t) = h_w(t - \frac{N}{2}t_o)$ , which has frequency response

$$H(j\omega) = e^{-j\frac{N}{2}\omega t_o} H_w(j\omega). \quad (20)$$

Notice that  $H_w(j\omega)$  is real function, so this is a polar representation of  $H(j\omega)$ . Note that the phase delay is  $\frac{\arg H(j\omega)}{\omega} = \frac{N}{2}t_o$  seconds, which gives us the phase distortionless linear phase causal filter that we are trying to design. The windowing function causes a smearing effect in the frequency domain. This is illustrated in Figure 5.

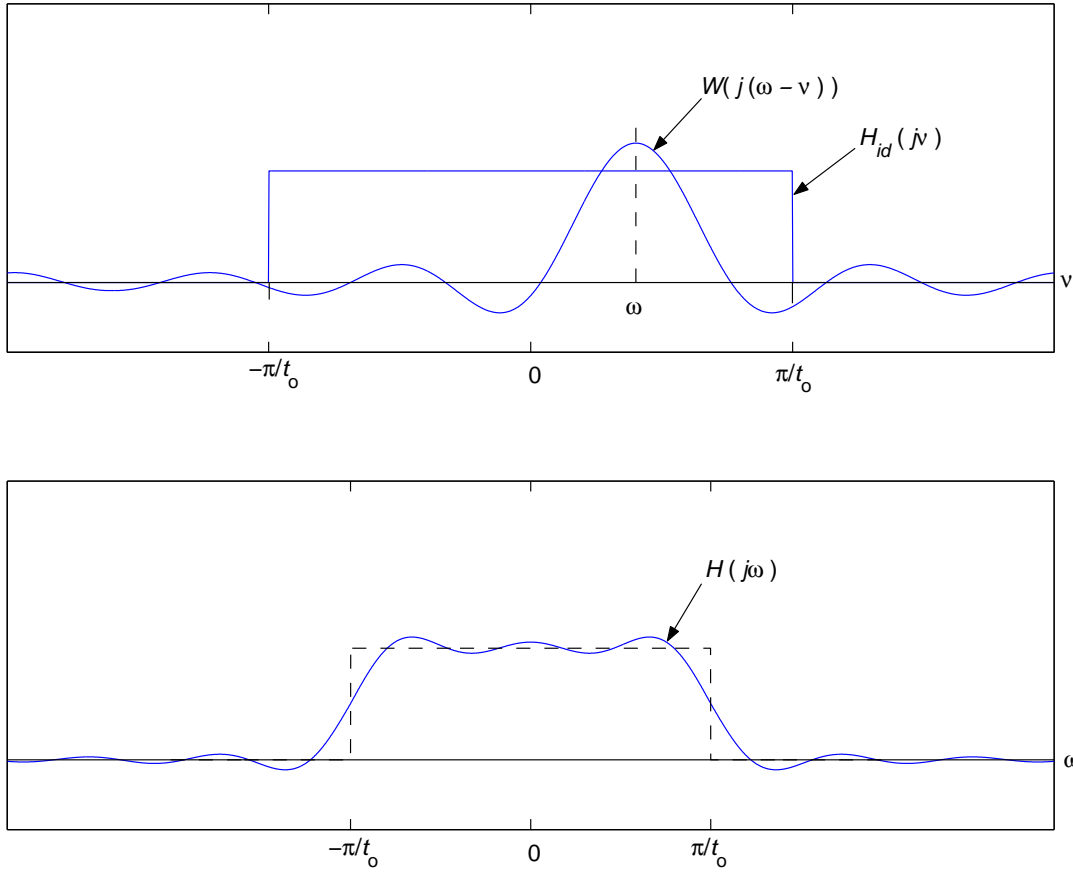


Figure 5: Convolution in Frequency

### *Impulse Invariance*

Once we have our causal analog finite-duration filter, we convert it to a discrete-time filter by using an impulse-invariance technique. Suppose we create the sampled sequence  $h(nt_o)$  from the finite duration impulse response  $h(t)$ . From the Shannon-Whittaker sampling theorem, we have [2]

$$H_s(e^{j\omega t_o}) = \frac{1}{t_o} \sum_{n=-\infty}^{\infty} H(j(\omega - n\omega_o)). \quad (21)$$

In the context of this section,  $H(j\omega)$  will be the spectrum defined in eqn (20), and  $H_s(e^{j\omega t_o})$  is the *scaled* frequency response of our intended digital FIR filter. Notice that frequency response  $H_s(e^{j\omega t_o}) = \frac{1}{t_o} H(j\omega)$  on the Nyquist band. This scaling of  $\frac{1}{t_o}$  can be rather large, so to compensate for this, we multiply our sampled sequence,  $h(nt_o)$ , by  $t_o$ . This scaling factor,  $t_o$ , will give the same power spectrum on the Nyquist band for both  $H(e^{j\omega t_o}) = t_o H_s(e^{j\omega t_o})$  and  $H(j\omega)$ . Therefore, the FIR filter that we implement in hardware,  $H(e^{j\omega t_o})$ , will have impulse response

$$h[n] = \begin{cases} t_o h(nt_o), & 0 \leq n \leq N \\ 0, & \text{otherwise} \end{cases}, \quad (22)$$

where the continuous-time function  $h(t)$  is the causal, windowed, impulse response of an ideal bandlimited analog filter. The most important idea here is that the spectrum  $H(j\omega)$  is essentially bandlimited to the Nyquist band, making  $H(e^{j\omega t_o})$  virtually void of aliasing effects.

## 2.3 Specific FIR Filter Designs Using Windowing Methods

In this section, we will explore digital filters with the four basic types of desired magnitude responses, namely lowpass, highpass, bandstop, and bandpass. Figure 6 shows each of these filters in terms of their bandlimited analog filter frequency responses.

These impulse responses of these filters may be calculated using the inverse Fourier transform identity

$$x(t) = \int_{\omega=-\infty}^{\infty} X(j\omega) e^{j\omega t} \frac{d\omega}{2\pi}. \quad (23)$$

Table 1 below shows the bandlimited analog impulse responses for each of the filters in Figure 6.

The frequency variables in Table 1 are in Hertz, with  $2\pi f_i = \omega_i$ ,  $i \in \{1, 2\}$ . This is done for ease of calculating filter impulse responses. When designing filters, we tend to think of frequency in Hertz, but when we do mathematical manipulations of Fourier transforms, we think of frequency,  $\omega$ , in radians per second.

So far, we have only used rectangular windows. Rectangular windows will give the sharpest transition at cutoff frequencies, but this will come at the expense of large oscillations in the magnitude response at cutoff frequencies. This effect is known as the *Gibbs phenomenon* in Fourier series analysis, and it is due to the non-uniform convergence in eqn (9) at values of  $t$

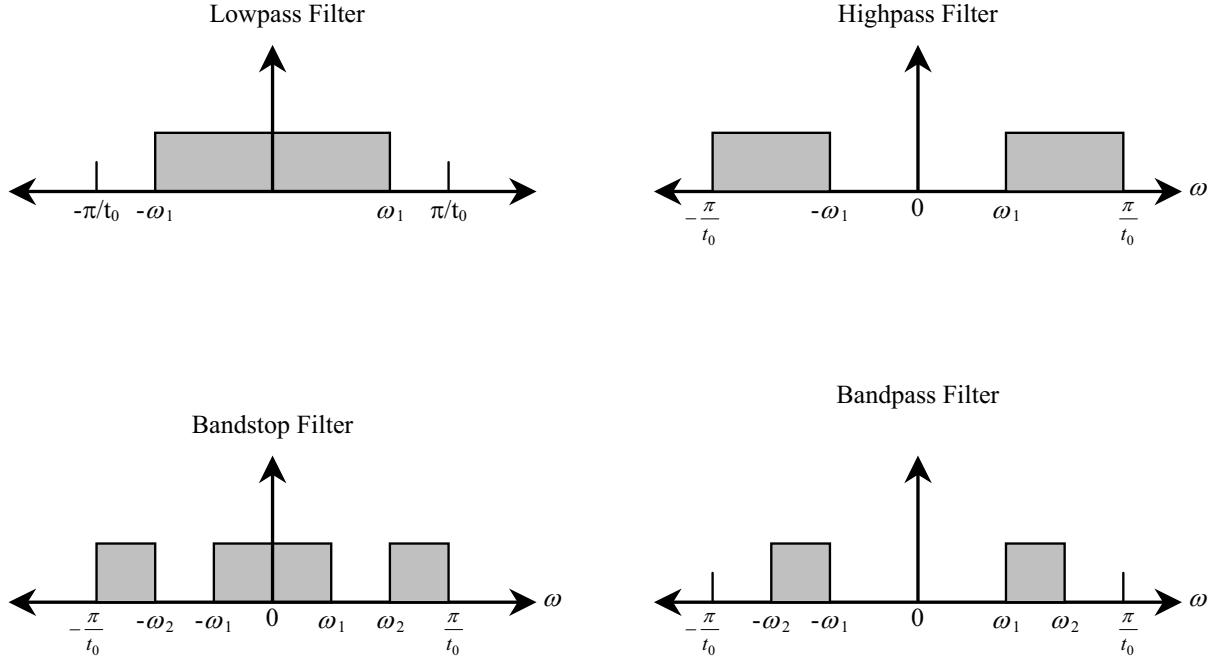


Figure 6: Ideal Analog Lowpass, Highpass, Bandstop, and Bandpass Filter Frequency Responses

Filter Type	$h(t)$
Lowpass filter	$2f_1 \text{sinc}(2\pi f_1 t)$
Highpass filter	$\frac{\sin(\frac{\pi}{t_0} t) - \sin(2\pi f_1 t)}{\pi t}$
Bandstop filter	$\frac{\sin(\frac{\pi}{t_0} t) - \sin(2\pi f_2 t) + \sin(2\pi f_1 t)}{\pi t}$
Bandpass filter	$\frac{\sin(2\pi f_2 t) - \sin(2\pi f_1 t)}{\pi t}$

Table 1: Ideal Lowpass, Highpass, Bandstop, and Bandpass impulse responses bandlimited to the Nyquist band.

where  $x(t)$  is discontinuous. To help alleviate this problem, other windows have been developed that truncate the delayed ideal impulse response in a less abrupt manner. These windows come at the expense of creating a larger transition band at the cutoff frequencies of the magnitude response of  $H(j\omega)$ .

When we talk about windowing functions, there are two characteristics, *main lobe width* and *relative first (peak) side lobe amplitude*, that concern us. The main lobe width determines the transition between pass and stop band frequencies (the smaller the main lobe width, the sharper the transition) and the relative peak side lobe reveals the Gibbs phenomenon (the lower the relative side lobe is to the main lobe, the less oscillation will be present in the magnitude response of  $H(j\omega)$ ). Unfortunately, this is a “no free lunch” situation. If you want a better transition at cutoff frequencies, you have to allow for more oscillations at the transition regions of your magnitude response, and if you want less oscillations in your magnitude response, then you have to tolerate a longer transition band between pass and stop bands. Here is partial list of windows available.

- *Rectangular Window*

The rectangular window is given in eqn (18).

The approximate width of the main lobe is  $\frac{4\pi}{(N+1)}f_o$  (rad/sec) and the relative peak side lobe amplitude is -13dB less than the amplitude of the main lobe.

- *Bartlett (Triangular) Window*

The Bartlett window is

$$w(t) = \begin{cases} \frac{2}{Nt_o}t + 1, & -\frac{N}{2}t_o < t < 0 \\ -\frac{2}{Nt_o}t + 1, & 0 \leq t < \frac{N}{2}t_o \\ 0, & \text{otherwise} \end{cases} \quad (24)$$

The approximate width of the main lobe is  $\frac{8\pi}{N}f_o$  (rad/sec) and the relative peak side lobe amplitude is -25dB less than the amplitude of the main lobe.

- *Hanning Window*

The Hanning window is

$$w(t) = \begin{cases} 0.5 + 0.5 \cos\left(\frac{2\pi}{Nt_o}t\right), & -\frac{N}{2}t_o < t < \frac{N}{2}t_o \\ 0, & \text{otherwise} \end{cases} \quad (25)$$

The approximate width of the main lobe is  $\frac{8\pi}{N}f_o$  (rad/sec) and the relative peak side lobe amplitude is -31dB less than the amplitude of the main lobe.

- *Hamming Window*

The Hamming window is

$$w(t) = \begin{cases} 0.54 + 0.46 \cos\left(\frac{2\pi}{Nt_o}t\right), & -\frac{N}{2}t_o < t < \frac{N}{2}t_o \\ 0, & \text{otherwise} \end{cases} \quad (26)$$

The approximate width of the main lobe is  $\frac{8\pi}{N}f_o$  (rad/sec) and the relative peak side lobe amplitude is -41dB less than the amplitude of the main lobe.

- *Blackman Window*

The Blackman window is

$$w(t) = \begin{cases} 0.42 + 0.5 \cos\left(\frac{2\pi}{Nt_o}t\right) - 0.08 \cos\left(\frac{4\pi}{Nt_o}t\right), & -\frac{N}{2}t_o < t < \frac{N}{2}t_o \\ 0, & \text{otherwise} \end{cases} \quad (27)$$

The approximate width of the main lobe is  $\frac{12\pi}{N}f_o$  (rad/sec) and the relative peak side lobe amplitude is -57dB less than the amplitude of the main lobe.

The windows above have been listed in order of smallest main lobe and highest peak side lobe to largest main lobe and lowest peak side lobe. These are not the only windows available, but they are some of the more common ones. We will experiment with a few of these later in the lab. In addition to the window functions above is the *Kaiser window*. This window has two parameters (filter length and a shape parameter) that are used to trade off side lobe amplitude for main lobe width. In this lab, we will use MATLAB to design a Kaiser window, but for some examples of Kaiser window designs, see [6, pg. 465–485] .

### 2.3.1 Example: Discrete-Time Differentiator

When doing digital signal processing of continuous-time signals, we may want to estimate the derivative of the continuous-time signal that we are processing. In continuous-time Fourier analysis, we have the following property

$$\begin{aligned} x(t) &\longleftrightarrow X(j\omega) \\ \frac{d}{dt}x(t) &\longleftrightarrow j\omega X(j\omega) \end{aligned} \tag{28}$$

In the context of digital FIR filter design using windowing, we begin by designing an ideal bandlimited analog filter with the frequency response

$$H(j\omega) = \begin{cases} j\omega, & -\frac{\pi}{t_o} < t < \frac{\pi}{t_o} \\ 0, & \text{otherwise} \end{cases}, \tag{29}$$

which is bandlimited to the Nyquist band. The corresponding impulse response is

$$h(t) = \frac{1}{t} \left( \frac{\cos(\frac{\pi}{t_o}t)}{t_o} - \frac{\sin(\frac{\pi}{t_o}t)}{\pi t} \right). \tag{30}$$

Differentiators are used for frequency discriminators in FM (frequency modulation) demodulators, which we will explore in more depth when we use DSP techniques to implement classical communication systems. Notice that the frequency response of  $H(j\omega)$  in eqn (28) is purely imaginary, so the corresponding impulse response in eqn (30) has odd symmetry ( $h(t) = -h(-t)$ ). This is what we expect out of an impulse response that mimics the differencing of a differentiator.

### 2.3.2 Example: Discrete-Time Hilbert Transformers

Another type of filter that is used in classical analog communication systems is a Hilbert transformer. In analog systems, a Hilbert transformer is an all-pass filter (magnitude of one at all frequencies) with the frequency response  $H(j\omega) = -j\text{sgn}(\omega)$ , where  $\text{sgn}(\cdot)$  is the signum function (returns the sign of a number). When designing a digital FIR Hilbert transformer using windowing, we start with the impulse response of the ideal bandlimited analog filter with spectrum

$$H(j\omega) = \begin{cases} j, & -\frac{\pi}{t_o} < \omega < 0 \\ -j, & 0 < \omega < \frac{\pi}{t_o} \\ 0, & \text{otherwise} \end{cases}, \quad (31)$$

which is bandlimited to the Nyquist band. The corresponding impulse response is

$$h(t) = \frac{2 \sin^2(\frac{\pi}{2t_o}t)}{\pi t}. \quad (32)$$

This filter has a purely imaginary frequency response and corresponding odd impulse response. Both the Hilbert transformer and the differentiator have a general linear phase, but they will cause phase distortion. We will explore this with square waves later in the lab. The phase distortion in the Hilbert transformer will be useful when we study classical analog communication systems, especially single-sideband (SSB) modulation.

## 2.4 Zero Placements in a Transfer Function

When we analyze digital filters, we examine their poles and zeros in the Z-domain. All analyzable filters are of the form  $H(z) = \frac{B(z)}{A(z)}$ . In the case of FIR filters,  $A(z) = 1$  and  $H(z) = B(z)$ . This means that FIR filters have no poles, only zeros. This also means that the coefficients of the polynomial  $B(z)$  are the filter coefficients  $\{h[n]\}_0^N$ . For real filter coefficients, the zeros of  $H(z)$  must appear in complex conjugate pairs. A zeros that occur on the unit circle at  $z = e^{j\omega_1 t_o}$  will translate to zeros in the magnitude response of the filter at frequency  $\omega_1$ . Frequency components that neighbor a zero on the unit circle will be attenuated. Zeros that are not on the unit circle will also shape the magnitude response of the filter  $H(z)$ . The closer a frequency on the unit circle gets to a zero, the more it will be attenuated. We will explore this later.

## 3 FIR Filter Design Using MATLAB SPTOOL

SPTOOL is a signal processing tool that is available for MATLAB<sup>4</sup>. SPTOOL provides a graphical user interface (GUI) for designing FIR and IIR filters. In this section, we are going to design a lowpass filter using MATLAB that you will use in the first FIR filter programming example.

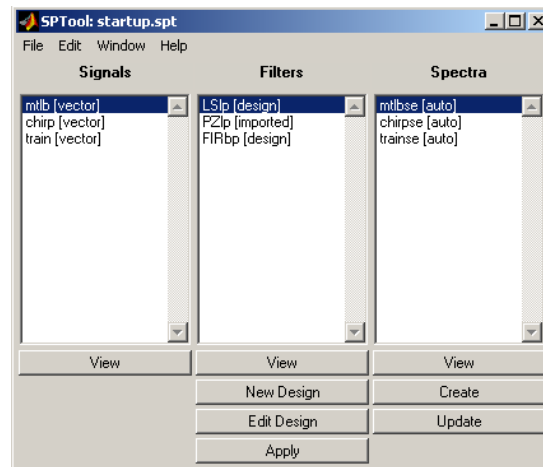
---

<sup>4</sup>The signal processing tool (SPTOOL) is a part of the MATLAB “Signal Processing Toolbox”, which is available on the engineering network services (ENS). The signal processing toolbox may also be purchased directly from Mathworks at [www.mathworks.com](http://www.mathworks.com).

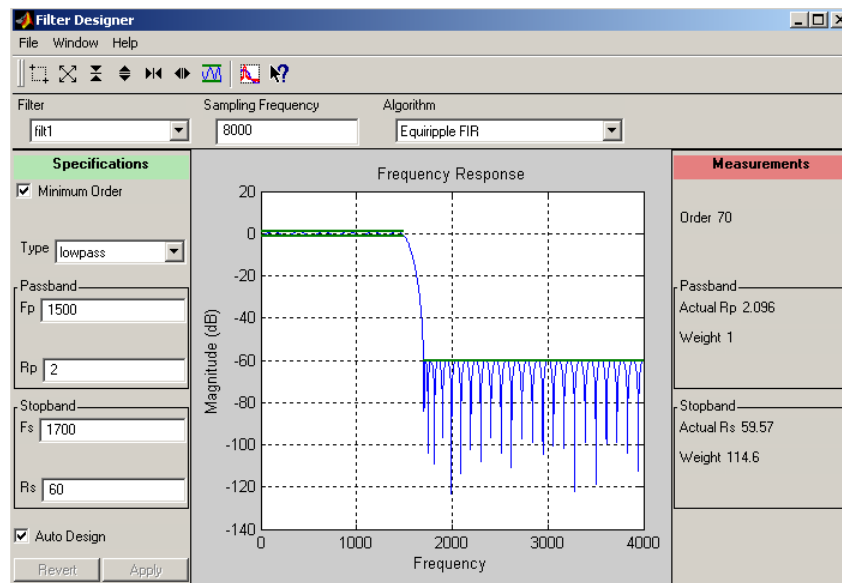
1. From the MATLAB command prompt, type

```
>> sptool
```

This will open the “SPtool: startup.spt” window (shown below). Keep this window open until told to close it.



2. In the middle column labelled “Filters”, select “New Design”. This will open up the “Filter Designer” window (shown below).



3. In the “Filter Designer” window, make sure the following are selected:
  - Select Response Type: “lowpass”
  - Make sure that “Equiripple FIR” is selected in the Design Method box.
  - In the Filter Order box, make sure “Minimum order” is checked.
  - In the Frequency Specification box, set:
    - Fs (sampling frequency) to 8000,
    - Fpass to 1500,
    - Fstop to 1700.
  - In the Magnitude Specification box, set:
    - Apass to 2

– Astop to 60.

4. Your filter has been designed by MATLAB. To access this filter from the workspace, go back to the window “SPtool: startup.spt”. Go to ‘File’ and select ‘Export’. In the window ‘Export from SPtool’, select **Filter: filt1 [design]** and then click on the button “Export to Workspace”.
5. Your filter is now available in the workspace. To see the filter coefficients, type the following

```
>> h=filt1.tf.num;  
>> stem(h)  
>> length(h)
```

in the MATLAB workspace.

When the above SPTOOL design was exported to the MATLAB workspace, the structured array “filt1” was created. To see what is included in this structured array, type `filt1` in the MATLAB workspace. The part of the structured array that we will be most interested in is the transfer function. To see the structure of the transfer function, type `filt1.tf` in the MATLAB workspace. You should see that the numerator (`num` in the MATLAB workspace) is a `1x71` array of type `double` (double precision floating-point numbers) and the denominator (`den` in the MATLAB workspace) is `1`. This is what we expect since FIR filters contain *no poles*, only zeros. As we have shown above, the numerator coefficients of the transfer function are access by typing the `filt1.tf.num` in the MATLAB workspace. These numerator coefficients are the FIR filter coefficients, namely  $h[n]$ , that we will use to implement our digital FIR filter in hardware.

To further analyze this filter, download the MATLAB m-file `plotZTP_FIR.m` from the class webpage. This m-file is a contains a function that will plot the Z-transform (evaluated at  $z = e^{j\omega t_o}$ ), the pole-zero diagram and the impulse response in a four-panel plot. This function is a modified version of `plotZTP.m` that is used to analyze all rational filters (taken from [4]). The modifications are that the denominator ( $A(z)$  in the representation  $H(z) = B(z)/A(z)$ ) is always 1, and the impulse response is always finite, so the entire impulse is displayed (This will not be the case for IIR filters). To analyze the filter, `filt1`, type the following

```
>> h=filt1.tf.num;  
>> fs=filt1.Fs;  
>> plotZTP_FIR(h,fs)
```

in the MATLAB workspace.

SPTOOL has three FIR filter options: Equiripple FIR, Least Squares FIR, and Kaiser Window FIR. The passband and stopband each have two sets of parameters `Fp`, `Rp`, `Fs`, and `Rs`. The parameters `Fp` and `Fs` define the boundaries of the passband and stopband frequencies, respectively. The parameter `Rp` determines the ripple (in dB) of the passband and `Rs` is the stopband rejection parameter that denotes the relative height of the stopband compared to the passband (i.e. `Rs = 60` means that the amplitude of the stop band is -60 dB less than the passband.) In the case of bandpass and bandstop filters, there will be two passband and stopband parameters, namely `Fp1`, `Fp2`, `Fs1`, and `Fs2`. When designing filters using SPTOOL, a transition band of frequencies must be included between `Fp1` and `Fs1` as well as `Fp2` and `Fs2`. The tighter these bands are, the larger your filter order will be.

At this point, your FIR filter should have been designed and exported to the MATLAB workspace. There is the homebrew MATLAB function, namely `FIR_cof_gen.m`, available for download from the webpage, which you will want to download into the directory on your u: drive for this lab. This MATLAB m-file is a function that takes the filter coefficients  $h[n]$  and creates a formatted



.cof that can be included into a C program. Specifically, the .cof file that you create here will be included in the C coded FIR filter in the next section. This file extension, .cof, is chosen to give the file name meaning. We will treat .cof files as though they were header files with a .h extension and include them in the beginning part of our C coded algorithms.

In the SPTOOL design example above, a lowpass filter with cutoff frequency 1500Hz was created. Once this filter was exported to the workspace as `filt1`, the filter coefficients were found by typing `h=filt1.tf.num`. These filter coefficients are formatted into a .cof by typing

```
>> FIR_cof_gen('LPF1500', h, 'fixed')
```

in the MATLAB workspace. (NB: The m-file `FIR_cof_gen.m` must be stored in your current working directory in MATLAB.) This command will create the file `LPF1500.cof`.

When you plotted the filter coefficients of the lowpass FIR filter designed in SPTOOL, you may noticed that the coefficients were all less than one. In fixed-point arithmetic, all of the filter coefficients must be signed-integers, so the numbers were all scaled by  $2^{15}$  and rounded to the nearest integer. Therefore, the scaling of the filter coefficients will result in a scaling of the output, which would overdrive the codec. To account for this, the output must be scaled down by multiplying by  $2^{-15}$ . This will be addressed in more detail in the next section.

One final comment about `FIR_cof_gen.m`. This function performs  $H_\infty$  (pronounced “H infinity”) scaling to scale the frequency response  $H(e^{j\omega t_o})$ . This scaling makes the gain  $\leq 1$  for all frequencies. This is done to help keep you from overdriving the codec.

In this lab, SPTOOL, Windowing, and zero placements will be the three methods that we will use to design FIR filters. In all three design methods, the MATLAB function `FIR_cof_gen.m` will be used to create the formatted filter coefficient (.cof) file that we include into our C code.

## 4 FIR Implementation on the C6713 DSK Using C and Assembly

One of the most common ways to describe digital filters is to use hardware diagrams to describe the input - output relationships. For FIR filters, one possible hardware diagram is illustrated in Figure 7. As the input stream  $x[n]$  enters the FIR filter it is delayed by the  $z^{-1}$  delay operator and scaled by the filter coefficient  $h[k]$ . The output  $y[n] = \sum_{k=0}^N h[k]x[n-k]$ . This hardware diagram shows that the output of FIR filter is calculated by doing explicit convolution.

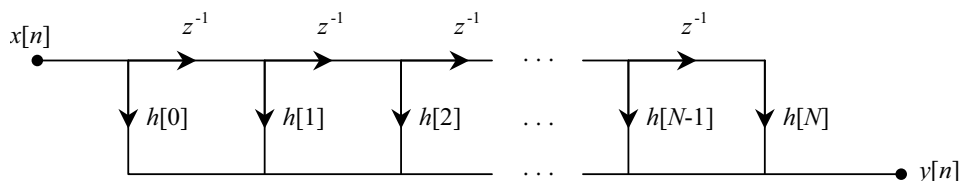


Figure 7: Delay Tap Hardware Diagram of an FIR Filter

Another way to describe FIR filters is to say that the output at time  $n$  is the inner product of a filter coefficient vector with an input vector containing the  $N+1$  most recent inputs. Lets define the two column vectors  $\underline{h} = [h[0], h[1], \dots, h[N]]^T \in \mathbb{R}^{N+1}$  and  $\underline{x}_n = [x[n], x[n-1], \dots, x[n-N]]^T \in \mathbb{R}^{N+1}$ . The output is defined to be  $y[n] = \langle \underline{h}, \underline{x}_n \rangle = \underline{h}^T \underline{x}_n = \sum_{k=0}^N h[k]x[n-k]$ , which, as we saw in the last lab, can be implemented on our DSP hardware.

To make this a real-time digital filter, the input vector  $\underline{x}_n$  must be updated as each sample is read in from the codec. After the next sample has been read in, the new input vector containing the  $N+1$  most recent inputs is  $\underline{x}_{n+1} = [x[n+1], x[n], \dots, x[(n+1)-N]]^T \in \mathbb{R}^{N+1}$  and the output is  $y[n+1] = \langle \underline{h}, \underline{x}_{n+1} \rangle$ . The vector  $\underline{x}_{n+1}$  is created by removing the oldest sample of  $\underline{x}_n$  from the bottom, shifting all of the elements down in vector (remember  $\underline{x}_n$  is a column vector), and storing the newest sample  $x[n+1]$  at the top of the vector. This is illustrated below.

Input Array	Time $n$	Time $n+1$	Time $n+2$	$\dots$
$x[0]$ $x[1]$ $x[2]$ $\vdots$ $x[N-2]$ $x[N-1]$ $x[N]$	$\begin{bmatrix} x[n] \\ x[n-1] \\ x[n-2] \\ \vdots \\ x[n-(N-2)] \\ x[n-(N-1)] \\ x[n-N] \end{bmatrix}$	$\begin{bmatrix} x[n+1] \\ x[n] \\ x[n-1] \\ \vdots \\ x[n-(N-3)] \\ x[n-(N-2)] \\ x[n-(N-1)] \end{bmatrix}$	$\begin{bmatrix} x[n+2] \\ x[n+1] \\ x[n] \\ \vdots \\ x[n-(N-4)] \\ x[n-(N-3)] \\ x[n-(N-2)] \end{bmatrix}$	$\dots$

The column on the left represents the C coded array that holds the  $N+1$  most recent inputs and the three columns on the right illustrate how the incoming samples are stored in the C coded array. This is not the most efficient way to organize the data in the input vector  $\underline{x}$ , but it provides a good starting point for coding digital FIR filters in C.

Create a project `FIR_c`, download the file `FIR_c.c` from the class webpage and open it in CCS. Be sure that you have created the file `LPF1500.cof` from the previous section, and put a copy of it in the folder (on your u: drive) for this project. Examine the code listing for `FIR_c.c` shown in Figure 8.

Within the ISR (lines 7 through 23), note the following:

- A local variable `k`, of type `short`, is created as a loop counter, the current sample is read in from the code and stored at the top of the array buffer `x[0]`, and the accumulating variable for the current output, global variable `yn`, is initialized to zero (lines 9 through 12).
- The explicit convolution (multiply and accumulate operations) of the  $N+1$  filter coefficients stored in the vector `h[]` with the  $N+1$  most recent inputs stored in the vector `x[]` is accumulated in `yn` (lines 14 and 15).
- The input buffer is shifted down starting at the bottom, where the last value of the array is no longer stored in memory (lines 17 and 18).
- Finally, the output is scaled (fixed point implementation, see below) and sent to the codec, and the ISR returns program control back to the `main()` function (lines 20 and 22).

---

```

/*1 */ // FIR_c.c  FIR filter. Include file of N order FIR filter coefficients
/*2 */ #include "DSK6713_aic23.h"
/*3 */ #include "LPF1500.cof"                // coefficient file LPF @ 1500Hz
/*4 */ int yn = 0;                          // initialize filter's output
/*5 */ short x[N+1];                        // input samples (N+1 samples)
/*6 */ Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;    // sampling frequency of codec
/*7 */
/*8 */ interrupt void c_int11()              // ISR
/*9 */ {
/*10*/     short k;
/*11*/
/*12*/     x[0] = input_left_sample(); // new input @ beginning of buffer
/*13*/     yn = 0;                      // initialize filter's output
/*14*/
/*15*/     for (k = 0; k<= N; k++)
/*16*/         yn += (h[k] * x[k]);      // y(n) += h(k)* x(n-k)
/*17*/
/*18*/     for (k = N; k > 0; k--)        // starting @ end of buffer
/*19*/         x[k] = x[k-1];            // update delays with data move
/*20*/
/*21*/     output_left_sample((short)(yn >> 15)); // scale output filter
/*22*/     return;
/*23*/ }
/*24*/
/*25*/ void main()
/*26*/ {
/*27*/     comm_intr();                    // init DSK, codec, McBSP
/*28*/     while(1);                      // infinite loop
/*29*/ }

```

Figure 8: Listing of FIR\_c.c

In fixed-point algorithms, this type of scaling must be done. This leads to a few additional comments. First, the input samples and the filter coefficients are 16-bit signed integers, and the multiplication of two 16-bit numbers will result in a 32-bit number. Therefore, the accumulation variable `yn` is a 32-bit signed integer, which, in C, means that the variable is of type `int` `yn` (instead of type `short yn`). The scaling up of the filter coefficients by  $2^{15}$  was done by `FIR_cof_gen.m` when the option ‘fixed’ was selected, and the down scaling was done in line 20 of Figure 8. In binary, multiplying or dividing a number by a power of two means shifting bits of that number either to the left or right with respect to a decimal point. In this case, we are dividing `yn` by  $2^{15}$ , so we need to shift the bits of `yn` fifteen bits to the right, which is done in C by using the right bit shift operator `>>` (see line 20).

#### 4.1 FIR Filter Design using MATLAB - Equiripple FIR \*

The filter implemented in Figure 8 is the equiripple FIR filter designed in SPTOOL. Look in the folder “include” in CCS and open the file `LPF1500.cof`. Observe that it contains the 71 filter

coefficients of the 70th order FIR filter created earlier. Also, the filter order  $N=70$  is defined. By including a separate file that contains the filter coefficients and filter order, we can keep the same basic FIR program and include different `.cof` files to change our FIR filter.

### ***Assignment***

1. Redo the SPTOOL design and run `FIR_cof_gen.m` to create `LPF1500.cof`. Implement this filter on the DSK. Use the signal generator and oscilloscope to verify that the filter is working properly by making a plot of normalized amplitude vs frequency for a 500mVpp input. Discuss the amplitude response plot. (HINT: Use the MATLAB function `plotZTP_FIR.m` to examine the filter.)
2. Make a new project `FIR_float_c` and download `FIR_float_c.c` from the class webpage. Using the homebrew function `FIR_cof_gen.m`, create a floating-point `.cof` file containing the 71 (unscaled) filter coefficients from the lowpass filter designed in SPTOOL earlier. Assuming that the filter coefficients are stored in the array `h` in MATLAB, typing the following command (at the MATLAB command prompt)

```
>> FIR_cof_gen('LPF1500_float', h, 'float')
```

will create the `.cof` file `LPF1500_float.cof`. You will need this file (with the name `LPF1500_float.cof`) to implement this project. In CCS, built this project and implement it on the DSK, then plot its frequency behavior.

Compare the files `FIR_c.c` and `FIR_float_c.c`. In your lab write-up, explain what the difference is between a fixed-point and floating-point FIR filtering algorithm.

### ***Assignment***

3. Create two new projects that use the *left channel* of the codec I/O paths. One project should implement a fixed-point arithmetic FIR filter with scaling and the other should use floating-point arithmetic. Name these fixed-point and floating-point projects `FIR_c` and `FIR_float_c`, respectively. Design a filter of your choice. Implement fixed-point and floating-point versions of this filter on the DSK. Use a sampling rate of the 8KHz, and take this into account when you design your filter in MATLAB. Use the homebrew MATLAB function `plotZTP_FIR.m` to analyze your filter. Include a copy of the figure generated by `plotZTP_FIR.m` and copies of your fixed-point and floating-point C coded algorithms. Comment on the changes that were required to implement your filter, and comment on the quality of the filter that you designed.
4. Once you have the programs in question 3 working, create a least squares 8KHz lowpass filter. using SPTOOL. Using the signal generator and oscilloscope, filter a square wave using the LPF LPF. Start with a low frequency square wave and increase the frequency gradually. As the frequency increases, more Fourier series coefficients of the square wave get filtered out. This is the same effect seen when using a rectangular window to design FIR filters. Due to the sampling and reconstruction of the codec, the waveform might not be exactly what you expect, but it should be similar to what one might expect. Find a pair of headphones and listen to the effects of dropping FS coefficients. Comment on what you see on the scope and hear. Give as much intuition as possible as to the affects of truncating FS coefficients. What intuition do you get about digital communication over bandlimited channels?
5. Create an audio (two-channel) versions of the fixed-point project of question 3. Name this project `FIR_audio_c`.

#### *Design Tips:*

The command `input_sample()` will read in a 32-bit binary number from the codec that contains two 16-bit signed integer (`short`) values. To split these two channels up, do the following:

```
int input_value;  
short input_left;  
short input_right;  
input_value = input_sample();  
input_left = (short)input_value;  
input_right = (short)(input_value >> 16);
```

Once you have the two channels split up, filter both channels and then use the function `output_left_right_sample(short left_sample, short right_sample);` to send the two channels worth of data to the codec.

## **4.2 FIR Filter Design - Notch Filter \***

Suppose that we would like to design a simple notch filter that knocks out a specific tone. This could be done by designing a filter with frequency response

$$\begin{aligned}
H(z) &= (1 - z^{-1}e^{j\omega_1 t_o})(1 - z^{-1}e^{-j\omega_1 t_o}) \\
&= 1 - 2 * \cos(\omega_1 t_o)z^{-1} + z^{-2} \\
&= 1 - 2 * \cos(2\pi f_1 t_o)z^{-1} + z^{-2}.
\end{aligned} \tag{33}$$

#### Assignment

- Design a notch filter for the codec that knocks out a 1KHz tone (or a tone of your choice). Use `plotZTP_FIR.m` to examine this filter. Plot the response of the filter near the notch frequency. Using a pair of headphones, listen to sinusoids at and near the notch frequency. How well does this filter suppress tones at the notch frequency? Comment on the filter.

### 4.3 FIR Filter Design - Zero Placement \*

#### Assignment

- Design a digital filter by placing zeros in a transfer function (e.g. define the real and/or complex conjugate pair zeros of some transfer function  $H(z)$ ). Create a transfer function  $H(z) = H_1(z)H_2(z)H_3(z)$ , where  $H_i(z) = 1 - \alpha_i 2 \cos(2\pi \frac{f_i}{f_0})z^{-1} + \alpha_i^2 z^{-2}$  is the transfer function of a complex conjugate zero pair at frequency  $f_i$  with magnitude  $\alpha_i$ . Use `plotZTP_FIR.m` to analyze your filter. Intuitively, explain how these zeros affect the frequency response of your filter. Implement this filter on the DSK. Include a copy of the MATLAB figure generated by `plotZTP_FIR.m`, give the magnitude and frequency of each complex conjugate zero pair, and comment on the output of this filter for input sinusoids of various frequencies. Give as much intuition as possible.

### 4.4 FIR Filter Design - Delay \*

Delay is an echoing effect that can be found on many guitar effects boards. It produces an effect that sounds much like the effect of hearing music at a sporting event. This effect is described by the difference equation  $y[n] = x[n] + a[m]x[n - m]$ , which is an FIR filter of the form  $H(z) = 1 + a[m]z^{-m}$ . Here  $|a[m]| \leq 1$  controls the amplitude of the echo and  $m$ , usually large, controls the delay (e.g.  $m = 400$  using the on-board codec translates to a 50ms delay). An example impulse response with  $a[m] = .7$  and  $m = 8$  would be coded in MATLAB as `h=[1 0 0 0 0 0 0 0 .7];`. Use `plotZTP_FIR.m` to examine this filter.

*Caveat:* For the delay filters that we will examine next, the frequency responses will take MATLAB too long to compute, so we will not explore them.

#### Assignment

- Design a digital delay filter for the with the parameters  $a[m] = .9$  and  $m = 400$  (a 50ms delay). This will require a buffer that will store the 400 most recent inputs, but will *only* require *two* multiply and accumulate operations. Create your C code in a way that does (at most) two multiply and accumulate operations. Play music through the DSK and listen to the effects. Try various values of  $a[m]$  and  $m$  to get a feel for how they affect the input. Specifically, try comment on delays of 50ms ( $m = 400$ ) and 100ms ( $m = 800$ )? Does this filter have a linear phase? Comment on your results and give as much intuition as possible.

## 4.5 FIR Filter Design - Differentiator \*

A discrete-time differentiator approximates the continuous-time derivative of the continuous signal that was sampled by the codec. In the ideal case, a square wave input will produce a delta-train output. For the discrete-time case, the inputs will be bandlimited. In the case of the square wave, this means that most of its higher order Fourier series coefficients will be removed by the anti-aliasing filter. This will produce an output that will look like sinc( $\cdot$ ) functions instead of Dirac delta functions.

### *Assignment*

9. Design a discrete-time differentiator using windowing. Use `plotZTP_FIR.m` to examine this filter. Implement it on the DSK. Use a sinusoidal input at various frequencies and comment on the results. Then, use a low frequency (100Hz to 400Hz) square wave as the input. Listen to the output. Also, observe the output wave form on the oscilloscope. Comment on what you see and here. Give a mathematical argument as to what is happening at the discontinuities. Give as much intuition as possible. Include a copy of the Figure generated by `plotZTP_FIR.m`, and your derivation of your impulse response commenting on the windowing function used and the filter order.

## 4.6 FIR Filter Design - Hilbert Transformer \*

The Hilbert transformer is used to shift the phase of positive frequencies by  $-90^\circ$  and the phase negative frequencies by  $90^\circ$ . Hilbert transformers are used in single-sideband communications. To the human ear, the effects of phase shifting are not noticeable, but the time response may be appear very different. We will observe this with a square wave input to a Hilbert transformer.

### *Assignment*

10. Design a discrete-time Hilbert transformer using windowing. Use `plotZTP_FIR.m` to examine this filter. Implement it on the DSK. Use a sinusoidal input at various frequencies and comment on the observed waveform. HINT: in order to see a fixed 90 degree phase shift between the output of the Hilbert Transformer and an unfiltered reference waveform, you will need to compensate in the reference channel for the delay added to the Hilbert Transformer to make it causal. This can only be done by putting the reference channel through the other stereo channel, unfiltered. Play music through the Hilbert transformer. Comment on what you hear. Then, use a lower frequency ( $\leq 1\text{kHz}$ ) square wave as the input with a limited amplitude (500mV to 600mV peak-to-peak). Listen to the output. Observe the output waveform on the oscilloscope. Comment on what you see and hear. Explain what is happening. Give as much intuition as possible. HINT: Think of a square wave as the sum of an infinite number of sinusoids, whose frequencies are odd multiples of the fundamental frequency. What effect does the Hilbert transformer have on these sinusoids in the time domain? Include a copy of the Figure generated by `plotZTP_FIR.m`, and your derivation of your impulse response commenting on the windowing function used and the filter order.

## 4.7 C Callable Assembly Code for Implementing FIR Filters \*

As we saw in the last lab, it is possible to call an assembly coded function from a C program. The FIR filters that we have implemented so far can be coded in assembly in the same way the inner product function `inprod_asm_func.asm` was coded. The only difference is that now the input sample vector `x[]` needs to be updated as the FIR filter output is being calculated. An example of a fixed-point FIR filter coded in assembly can be seen in Figure 9.

---

```
; FIR_asm_func.asm
; asm function called from C to implement fixed-point FIR
; A4 = x[n] address, B4 = h[0] address, A6 = filter order N
; input samples organized as x(n)...x(n-N)
; coefficients as h[0]...h[N]

1.)          .def      _FIR_asm_func
2.)_FIR_asm_func:      ; asm function called from C
3.)          MV        .L1    A6,A1      ; setup loop count in A1
4.)          ZERO      .S1    A8        ; init A8 for accumulation
5.)          LDH        .D1    *A4++,A2   ; x[n]
6.)          LDH        .D2    *B4++,B2   ; h[0]
7.)          NOP                4
8.)          MPY        .M1x   A2,B2,A7   ; A7=x[n]*h[0]
9.)          NOP
10.)         ADD        .L1    A7,A8,A8   ; accumulate in A8
11.)
12.)LOOP:      ; start of FIR loop
13.)         MV        .L1    A2,A3      ; used to update input vector x
14.)         LDH        .D1    *A4,A2     ; A2=x[n-k] k=1,...,N
15.)         LDH        .D2    *B4++,B2   ; B2=h[k] k=1,...,N
16.)         NOP                4
17.)         STH        .D1    A3,*A4++   ; update input vector, inc x[k]
18.)         MPY        .M1x   A2,B2,A7   ; A7=h[k]*x[n-k]
19.)         NOP
20.)         ADD        .L1    A7,A8,A8   ; accumulate in A8
21.)         SUB        .S1    A1,1,A1    ; decrement loop count A1
22.)
23.) [A1] B        .S2    LOOP           ; branch to loop if A1 # 0
24.)         NOP                5
25.)
26.)         MV        .L1    A8,A4      ; result returned in A4
27.)         B         .S2    B3        ; return addr to calling routine
28.)         NOP                5
```

Figure 9: Listing of `FIR_asm_func.asm`

The function `FIR_asm_func.asm` is called from the C program `FIR_asm.c` in Figure 10.

The FIR filter `FIR_asm_func.asm` may be optimized in the same way as the inner product function was optimized in the last lab. An optimized version of this program can be found on the class webpage as the `FIR_asm_opt.asm`. It is left to the student to examine the optimized



---

```

/*1 */ //FIR_asm.c FIR C program calling ASM function FIR_asm_func.asm
/*2 */ #include "DSK6713_aic23.h"
/*3 */ #include "LPF1500.cof"                // LPF 1500 Hz cutoff
/*4 */ int yn = 0;                          // initialize filter's output
/*5 */ short x[N+1];                        // input samples (N+1 samples)
/*6 */ Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;    // sampling frequency
/*7 */ interrupt void c_int11()              // ISR
/*8 */ {
/*9 */     x[0] = input_left_sample();        // newest sample @ top of buffer
/*10*/     yn = FIR_asm_func(x,h,N);          // to asm func through A4,B4,A6
/*11*/     output_left_sample((short)(yn >> 15)); // scale output filter
/*12*/
/*13*/     return;
/*14*/ }
/*15*/
/*16*/ void main()
/*17*/ {
/*18*/     short i;
/*19*/
/*20*/     for (i = 0; i<N; i++)
/*21*/         x[i] = 0;                      // init buffer for delays
/*22*/
/*23*/     comm_intr();                         // init DSK, codec, McBSP
/*24*/     while(1);                          // infinite loop
/*25*/ }

```

Figure 10: Listing of FIR\_asm.c

assembly code. Create the projects `FIR_asm` and `FIR_asm_opt`; download and examine the the accompanying files from the class webpage

### **Assignment**

11. Build the executable files `FIR_asm.out` and `FIR_asm_opt.out`. Implement one of the fixed-point FIR filters designed earlier to make sure that both algorithms are working properly. Briefly explain what changes were made to `FIR_asm_func.asm` to optimize it.
12. Briefly explain how each CPU core register (e.g. A0-A15 and B0-B15) is being used in `FIR_asm_func.asm`. Create a linear assembly version of `FIR_asm_func.asm` using meaningful names for each register. Implement it on the DSK. Include a copy of your linear assembly function and an explanation of how each register is being used.

## **4.8 Circular Buffers on the TMS320C6713**

The most efficient way to code FIR filters is to use circular buffers. The idea of circular buffers is that once you get to the end of a buffer, you wrap around to the first element of the buffer. We will use a circular buffer to hold our inputs. The advantage to this is that we will not need to update the input buffer (`x[n]`) after each output has been calculated. Instead, as a new sample

is read in, it is stored in the memory location of the oldest sample, and then the current output is calculated and sent to the codec. The memory management of circular buffers is as follows.

Array Index	$h[]$	$x[]$
0	$h[0]$	$x[n-\textit{newest}]$
1	$h[1]$	$x[n-\textit{newest}+1]$
$\vdots$	$\vdots$	$\vdots$
		$x[n-1]$
<i>newest</i>		$x[n]$
<i>oldest</i>		$x[n-N+1]$
		$x[n-N+2]$
$\vdots$	$\vdots$	$\vdots$
N-2	$h[N-2]$	$x[n-\textit{newest}-21]$
N-1	$h[N-1]$	$x[n-\textit{newest}-1]$

Memory Management Using Circular Buffers. Adapted from [8].

The twist on this story is that we want to increment up in memory locations through our circular buffer  $x[]$ , but the first input sample that we will access will be  $x[n-N]$ . This involves working backwards (in relation to the other FIR algorithms) through our input array. As a result, we must go backwards (in relation to the other FIR algorithms) through the the coefficient array,  $h[]$ . This is done by starting at the bottom of the coefficient array,  $h[N]$ , and decrementing our way back to the top of the array. At the same time, we increment forward through the input array. This is done since we are not storing our input samples in reverse order (as we did in `FIR.c`, see Figure 8). To see this, let's examine the an assembly coded FIR filter that uses circular buffers. See Figure 11 below.

In Figure 11, lines 13, 15 and 16 are used to align the filter coefficient register with the bottom of the coefficient array  $h[n]$ . Line 13 accounts for the 16-bit signed integer filter coefficients

---

```

; FIR_asm_circ_func.asm
; asm function called from C to implement fixed-point FIR filter
; using circular addressing
; A4=newest sample, B4=coefficient address, A6=filter order
; input samples organized: x[n]...x[n-N]
; coefficients as h[0]...h[N]

1 .)          .def          _FIR_asm_circ_func
2 .)          .def          last_addr
3 .)          .def          delays
4 .)          .sect         "circdata"      ; circular data section
5 .)          .align        256             ; align delay buffer 256-byte boundary
6 .) delays    .space       256             ; init 256-byte buffer with 0's
7 .) last_addr .int         last_addr-1     ; point to input buffer
8 .)
9 .)          .text          ; code section
10.) _FIR_asm_circ_func:    ; FIR function using circ addr
11.)          ADD           A6,1,A6         ; duration N+1 samples
12.)          MV            A6,A1          ; setup loop count
13.)          MPY           A6,2,A6         ; 2-byte filter coeff.
14.)          ZERO         A8             ; init A8 for accum. (NOP for MPY)
15.)          ADD           A6,B4,B4        ; go to bottom of filter buffer
16.)          SUB           B4,1,B4        ; align with bottom of filter buffer
17.)          MVKL          0x00070040,B6   ; select A7 as pointer and BK0
18.)          MVKH          0x00070040,B6   ; BK0 for 256 bytes (128 shorts)
19.)          MVC           B6,AMR         ; set address mode register AMR
20.)          MVK           last_addr,A9    ; A9=last circ addr(lower 16 bits)
21.)          MVKH          last_addr,A9    ; last circ addr (higher 16 bits)
22.)          LDW           *A9,A7         ; A5=last circ addr
23.)          NOP           4
24.)          STH           A4,*A7++       ; newest sample to last address
25.) LOOP:      ; start of FIR loop
26.)          LDH           .D1  *A7++,A2   ; A2=x[n-k] k=1,...,N
27.)          ||          LDH           .D2  *B4--,B2 ; B2=h[k] k=1,...,N
28.)          SUB           .S1  A1,1,A1    ; decrement loop count
29.)          [A1]         B           .S2  LOOP ; branch to loop if count # 0
30.)          NOP           2              ; 3rd and 4th NOPs for LDH
31.)          MPY           .M1x A2,B2,A6   ; A7=h[k]*x[n-k]
32.)          NOP
33.)          ADD           .L1  A6,A8,A8   ; accumulate in A8
34.)          STW           A7,*A9         ; store last circ addr to last_addr
35.)          B            .S2  B3         ; return addr to calling routine
36.)          MV           .L1  A8,A4      ; result returned in A4
37.)          NOP           4

```

Figure 11: Listing of FIR\_asm\_circ\_func.asm

(2 bytes = 16 bits) and lines 15 and 16 search to the end of the buffer. From here, the loop works exactly as it did before. Here, register A5 has been configured to be a circular buffer of length 256 bytes (this configuration will be explained shortly). When the register A5 is at the last memory location in the buffer  $x[n]$ , it will automatically reset to the first memory location

in the buffer when the it is incremented (see the command `*A5++` in line 26).

Circular buffers require extra initialization to work properly. First, the memory address of the first memory location of a circular buffer must be aligned with an address boundary. In our case, we have 128 16-bit filter coefficients that take up 256 bytes of memory. By aligning with a 256-bit address boundary<sup>5</sup>, the first memory address will be `X100` in hexadecimal, where `X` is some number<sup>6</sup>. Now as a program increments through the memory addresses, it will reach the last element of the buffer at memory location `X1FE` (hex). Incrementing this address would yield `X200` (hex), but a circular buffer will not allow this addition to carryover into bit 9 of the memory address, so the result will be `X100` (hex) and not `X200` (hex). This will align the pointer in `A5` to the top of the buffer without programmer intervention. Basically, circular addressing freezes a certain number of most significant bits in a register and ignores the carryovers from addition. These types of registers are referred to as *circular registers*, while non-circular registers are referred to as *linear registers*.

In assembly code, the input buffer that holds `x[n]` is aligned with a 256-bit boundary in line 12. Next, the register `A5` must be initialized to act as circular buffer for a 256-bit buffer. This is done by setting the address mode register (AMR) in the CPU core. In this example, we loaded a hexadecimal number with this information into the register `B6` by using the *move constant* commands `MVK` and `MVKH`, where `MVK` moves the lower 16-bits of the constant into `B6` (line 26) and `MVKH` moves the higher 16-bits into `B6` (line 27). This 32-bit number in `B6` is then stored in the AMR register using the move constant command `MVC` in line 29. NB: The move constant command is the only way to write to the specialized CPU core registers like the AMR.

The next step in this program is to store the current (read in) data sample into the memory location of the oldest sample in the circular buffer. Here, a global variable `last_addr`, which is defined on line 13, is used to hold the location of the last input sample. When this program is called, only the new sample (16-bit signed integer) is passed to the function. Lines 34 through 36 are used to store the current sample in the memory location of the oldest sample. The pointer in the circular register `A5` is then incremented to the next valid memory location within the circular buffer.

This program requires a slightly different C program. The C program no longer needs an array to hold the inputs, and therefore, only needs to send the current input sample to the function `FIR_asm_circ_func.asm`. Figure 12 contains a listing of the C function that calls a FIR assembly coded program that uses circular buffers.

This circular buffer has been coded assuming the the filter impulse response is of duration 128 samples (127th order filter). For this code to work, the filter duration must be a power of two. For example, if the filter duration were changed to 32 samples, the program would need to be altered so that the input samples were aligned to a 64-bit boundary and the AMR would need to be configured to handle a 64-bit circular register (instead of a 256-bit register). Only registers `A4-A7` and `B4-B7` may be initialized to due circular addressing. To learn more about circular addressing and initializing the AMR, see [7], [1], and [8].

Circular buffers are the fastest way to implement FIR filters, but this speed will not be necessary for the projects implemented in this class. Nonetheless, you should try to implement these whenever possible.

---

<sup>5</sup>Each byte of memory is addressed by one bit in a memory address.

<sup>6</sup>In binary, this number is `XXXX XXX1 0000 0000`, where bit 9 ( $2^9 = 256$ ) is the boundary that the memory address has been aligned to.

---

```

/*1 */ // FIR_asm_circ.c C program calling asm function using a circular buffer
/*2 */ #include "DSK6713_aic23.h"
/*3 */ #include "LPF1500_128.cof"           // LPF, 1500 Hz cutoff
/*4 */ int yn = 0; short yn_short=0;        // init filter's output
/*5 */ Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;  // sampling frequency
/*6 */ interrupt void c_int11()             // ISR
/*7 */ {
/*8 */   short sample_data;
/*9 */
/*10*/   sample_data = input_left_sample(); // newest input sample data
/*11*/   yn = FIR_asm_circ_func(sample_data,h,N);
/*12*/   yn_short = (short) (yn>>15);      // asm func passing to A4,B4,A6
/*13*/   output_left_sample(yn_short);      // filter's output
/*14*/   return;                           // return to calling function
/*15*/ }
/*16*/
/*17*/ void main()
/*18*/ {
/*19*/   comm_intr();                        // init DSK, codec, McBSP
/*20*/   while(1);                          // infinite loop
/*21*/ }

```

Figure 12: Listing of FIR\_asm\_circ.c

## 5 End Notes

In this lab, we explored some simple techniques for designing and implementing FIR filters. In general, FIR filters are used when a linear phase is required. Also FIR filters give an insight to how the zeros of a transfer function affect the frequency response of a filter. We will use these filters in later labs when Hilbert transformers and differentiators are required.

### *Lab Suggestions for Course Project*

1. Assembly code FIR filters that use the floating-point instructions available for the C6713.
2. Implement circular buffers for an  $N$  order filter, where  $N + 1$  is not a power of two.
3. Design Least-Squares FIR filters (Farden/Scharf designs [5]) by hand. Explain what the weighting functions do and how they affect the frequency response of your filter.

## References

- [1] R. Chassaing, *Digital Signal Processing and Applications with the C6713 and C6416 DSK*. Wiley, New Jersey, 2005.
- [2] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 10: Sampling, Reconstruction, and Rate Conversion*, 2001.
- [3] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 5: Periodic Signals and Fourier Series*, 2001.
- [4] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 9: The Z Transform, the DTFT, and Digital Filters*, 2001.
- [5] D. C. Farden and L.L. Scharf. Statistical design of nonrecursive digital filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 3(22):188–196, June 1974.
- [6] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-Time Signal Processing*. Prentice Hall, Upper Saddle River, NJ, 1989.
- [7] Texas Instruments, Dallas, TX, *TMS320C6000 CPU and Instruction Set Reference, SPRU189G*, 2006.
- [8] Steven A. Tretter. *Communication Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C6701 and TMS320C6711*. Kluwer Academic/Plenum Publishers, New York, 2003.