Chapter 6

# Real-Time FIR Digital Filters

## Introduction

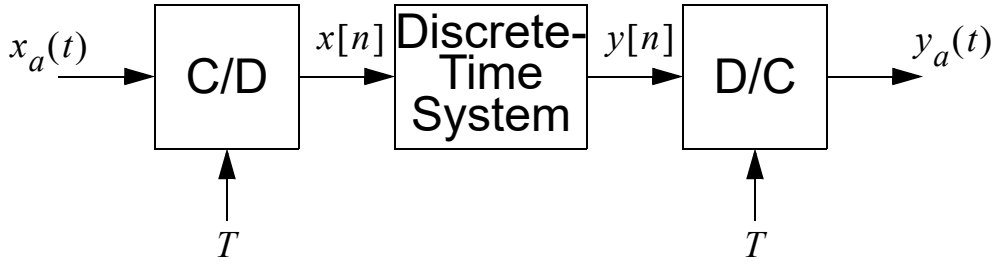
Digital filter design techniques fall into either finite impulse response (FIR) or infinite impulse response (IIR) approaches. In this chapter we are concerned with just FIR designs. We will start with an overview of general digital filter design, but the emphasis of this chapter will be on real-time implementation of FIR filters using C and assembly.

- Basic FIR filter topologies will be reviewed
- To motivate the discussion of real-time implementation we will begin with a brief overview of FIR filter design
- Both fixed and floating-point implementations will be considered
  - The MATLAB signal processing toolbox and filter design toolbox will be used to create quantized filter coefficients
- The use of circular addressing in assemble will be considered
- Implementing real-time filtering using analog I/O will be implemented using the Wolfson WM8731 or WM5102 codec interfaces and associated ISR or DMA routines

# Basics of Digital Filter Design

- A filter is a frequency selective linear time invariant (LTI) system, that is a system that passes specified frequency components and rejects others
- The discrete-time filter realizations of interest here are those LTI systems which have LCCDE representation and are causal
- Note that for certain applications noncausal filters are appropriate
- An important foundation for digital filter design are the *classical* analog filter approximations
- The filter design problem can be grouped into three stages:
  - Specification of the desired system properties (application driven)
  - Approximation of the specifications using causal discrete-time systems
  - System realization (technology driven - hardware/software)

- A common scenario in which one finds a digital filter is in the filtering of a continuous-time signal using an A/D-$H(z)$-D/A system




- Strictly speaking $H(z)$ is a discrete-time filter although it is commonly referred to as a *digital filter*
- Recall that for the continuous-time system described above (ideally)

$$H_{\text{eff}}(j\Omega) = \begin{cases} H(e^{j\Omega T}), & |\Omega| < \pi/T \\ 0, & \text{otherwise} \end{cases} \tag{6.1}$$

- Using the change of variables $\omega = \Omega T$, where $T$ is the sample spacing, we can easily convert continuous-time specifications to discrete-time specifications i.e.,

$$H(e^{j\omega}) = H_{\text{eff}}\left(j\frac{\omega}{T}\right), |\omega| < \pi \tag{6.2}$$

## Overview of Approximation Techniques

- Digital filter design techniques fall into either IIR or FIR approaches
- General Approximation Approaches:
  - Placement of poles and zeros (ad-hoc)
  - Numerical solution of differential equations
  - Impulse invariant (step invariant etc.)
  - Bilinear transformation
  - Minimum mean-square error (frequency domain)
- FIR Approximation Approaches
  - Truncated impulse response of ideal *brickwall* responses using window functions
  - Frequency sampling desired response using transition samples
  - Optimum equiripple approximations (use the *Parks-McClellan algorithm* via the *Remez exchange algorithm*, `firpm()` in MATLAB)
  - Minimum mean-square error in the frequency domain
- In the FIR approaches the additional constraint of *linear phase* is usually imposed

# Basic FIR Filter Topologies

- Recall that a causal FIR filter containing $M+1$ coefficients has impulse response

$$h[n] = \sum_{k=0}^{M} a_k \delta[n-k] \tag{6.3}$$

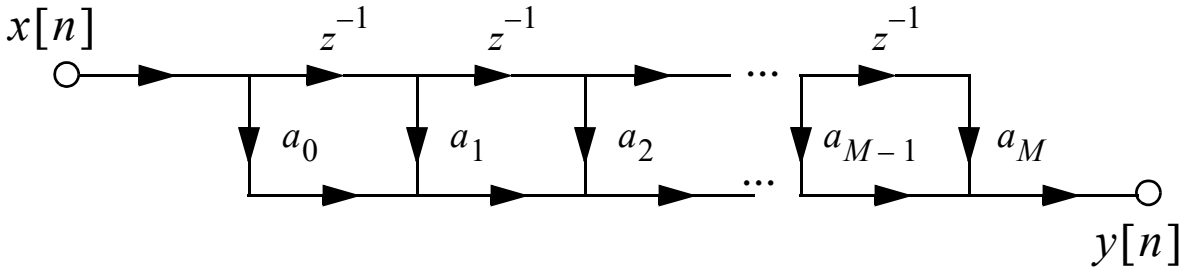
- The corresponding $z$-transform of the filter is

$$H(z) = \sum_{k=0}^{M} a_k z^{-k} = a_0 + a_1 z^{-1} + \ldots + a_{M-1} z^{-M} \tag{6.4}$$

- The frequency response of the filter is

$$H(e^{j\omega}) = \sum_{k=0}^{M} a_k e^{-j\omega k} \tag{6.5}$$

- The classical direct form topology for this filter is shown below

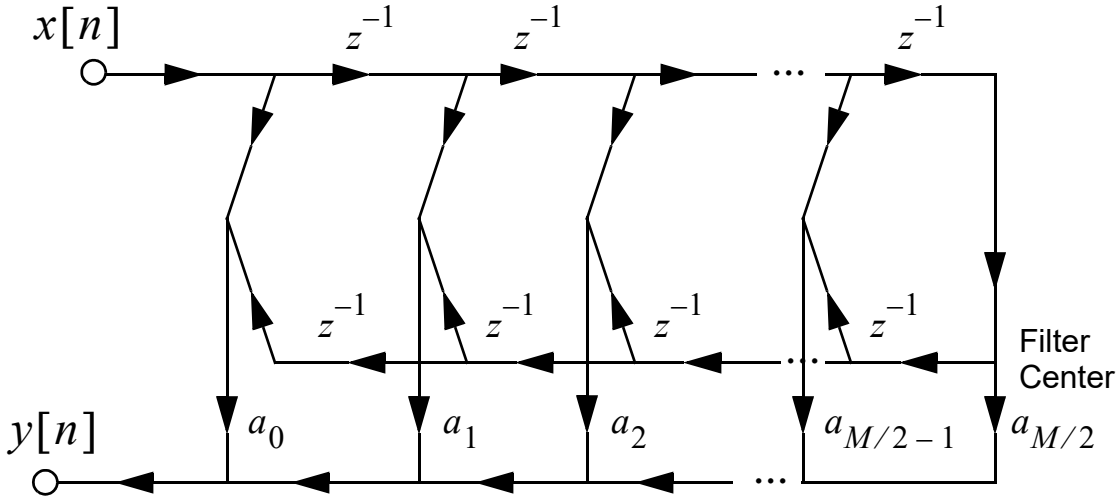



Direct Form FIR Structure

- Often times we are dealing with linear phase FIR designs which have even or odd symmetry, e.g.,

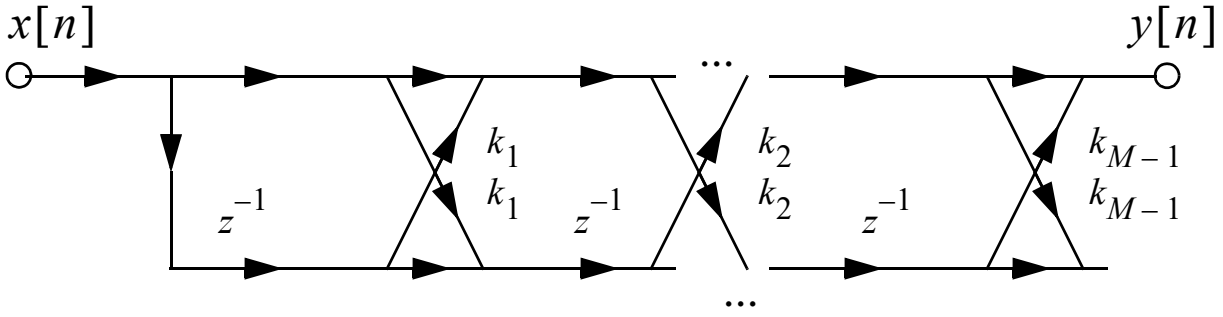
$$h[n] = h[M-n] \text{ even}$$
$$h[n] = -h[M-n] \text{ odd} \tag{6.6}$$

- In this case a more efficient structure, reducing the number of multiplications from $M+1$ to $M/2+1$, is




Modified Direct Form FIR Structure for M Even

- Another structure, not as popular as the first two, is the FIR lattice




FIR Lattice Structure

  – Although not discussed any further here, the *reflection coefficients*, $k_i$, are related to the $a_k$ via a recurrence formula (in MATLAB we use `k=tf2latc(num)`)

- The lattice structure is very robust under coefficient quantization

# Overview of FIR Filter Design

## Why or Why Not Choose FIR?

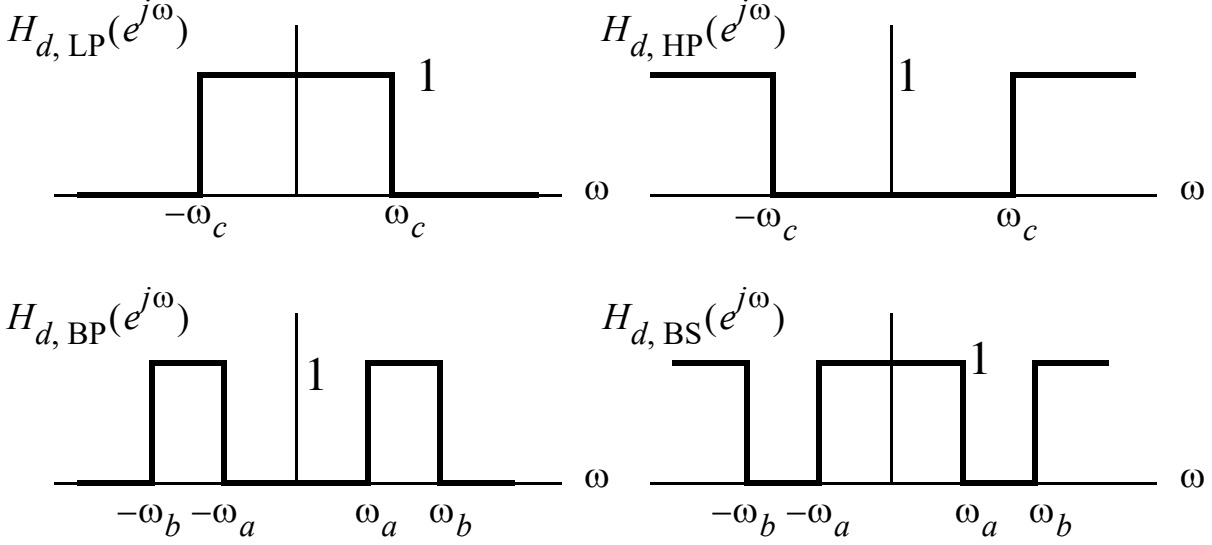
- FIR advantages over IIR
  - Can be designed to have exactly linear phase
  - Typically implemented with nonrecursive structures, thus they are inherently stable
  - Quantization effects can be minimized more easily
- FIR disadvantages over IIR
  - A higher filter order is required to obtain the same amplitude response compared to a similar IIR design
  - The higher filter order also implies higher computational complexity
  - The higher order filter also implies greater memory requirements for storing coefficients

## FIR Design Using Windowing

- The basic approach is founded in the fact that ideal lowpass, highpass, bandpass, and bandstop filters, have the following noncausal impulse responses on $-\infty < n < \infty$

$$
\begin{aligned}
h_{d,\,\mathrm{LP}}[n] &= \frac{\sin(\omega_c n)}{\pi n} \\
h_{d,\,\mathrm{HP}}[n] &= \delta(n) - \frac{\sin(\omega_c n)}{\pi n} \\
h_{d,\,\mathrm{BP}}[n] &= \frac{\sin(\omega_b n) - \sin(\omega_a n)}{\pi n} \\
h_{d,\,\mathrm{BS}}[n] &= \delta(n) - \frac{\sin(\omega_b n) - \sin(\omega_a n)}{\pi n}
\end{aligned}
\tag{6.7}
$$

where the ideal frequency responses corresponding to the above impulse responses are




Ideal Brickwall Filters

- Consider obtaining a causal FIR filter that approximates $h_d[n]$ by letting

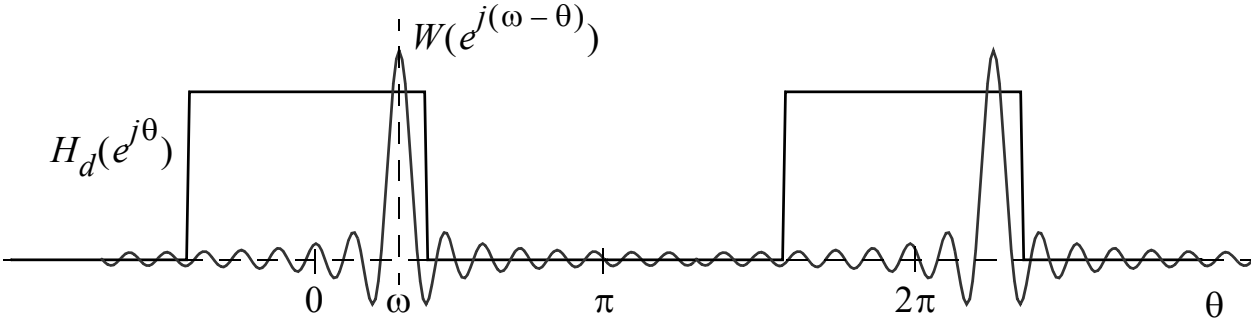
$$
h[n] = \begin{cases} h_d[n - M/2], & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases}
\tag{6.8}
$$

- Effectively we have applied a rectangular window to $h_d[n]$ and moved the saved portion of the impulse response to the right to insure that for causality, $h[n] = 0$ for $n < 0$
  - The applied rectangular window can be represented as

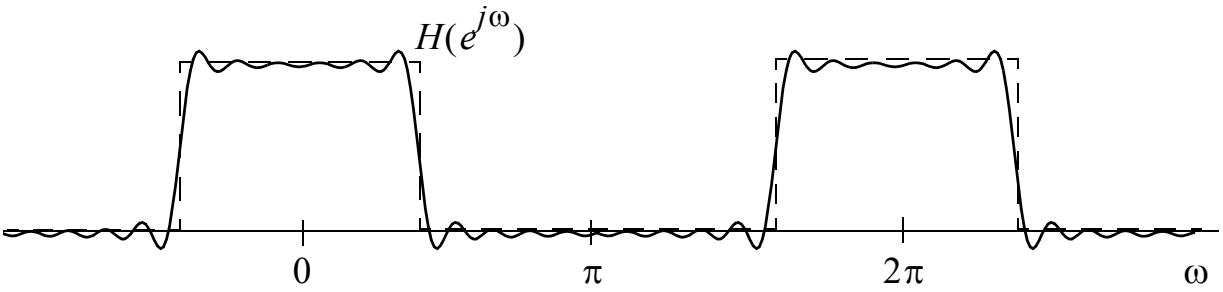
$$h[n] = h_d[n - M/2]w[n] \tag{6.9}$$

  where here $w[n]$ is unity for $0 \le n \le M$ and zero otherwise
- Since the rectangular window very abruptly shuts off the impulse response, Gibbs phenomenon oscillations occur in the frequency domain at the band edges, resulting in very poor stopband response
- This can be seen analytically with the aid of the windowing Fourier transform theorem

$$H(e^{j\omega}) = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{-j\theta M/2} H_{\mathrm{d}}(e^{j\theta}) W(e^{j(\omega-\theta)}) d\theta \tag{6.10}$$




(a) Circular convolution (windowing)




(b) Resulting frequency response of windowed sinc

  - Use of the rectangular window results in about 9% peak ripple in both the stopband and passband; equivalently the peak passband ripple is 0.75 dB, while the peak sidelobe level is down only 21 dB

- To both decrease passband ripple and lower the sidelobes, we may smoothly taper the window to zero at each end
  - Windows chosen are typically symmetrical about the center $M/2$
  - Additionally, to insure linear phase, the complete impulse response must be either symmetric or antisymmetric about $M/2$

## Example: Use of the Hanning Window

- The Hanning window is defined as follows

$$w[n] = \begin{cases} 0.5 - 0.5\cos(2\pi n/M), & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases} \tag{6.11}$$

- Consider an $M = 32$ point design with $\omega_c = \pi/2$
- The basic response $h_d[n]$ for a lowpass is of the form

$$h_d[n] = \frac{\sin(\omega_c n)}{\pi n} = \frac{\omega_c}{\pi}\text{sinc}\left(\frac{\omega_c n}{\pi}\right) \tag{6.12}$$

since $\text{sinc}(x) = \sin(\pi x)/(\pi x)$

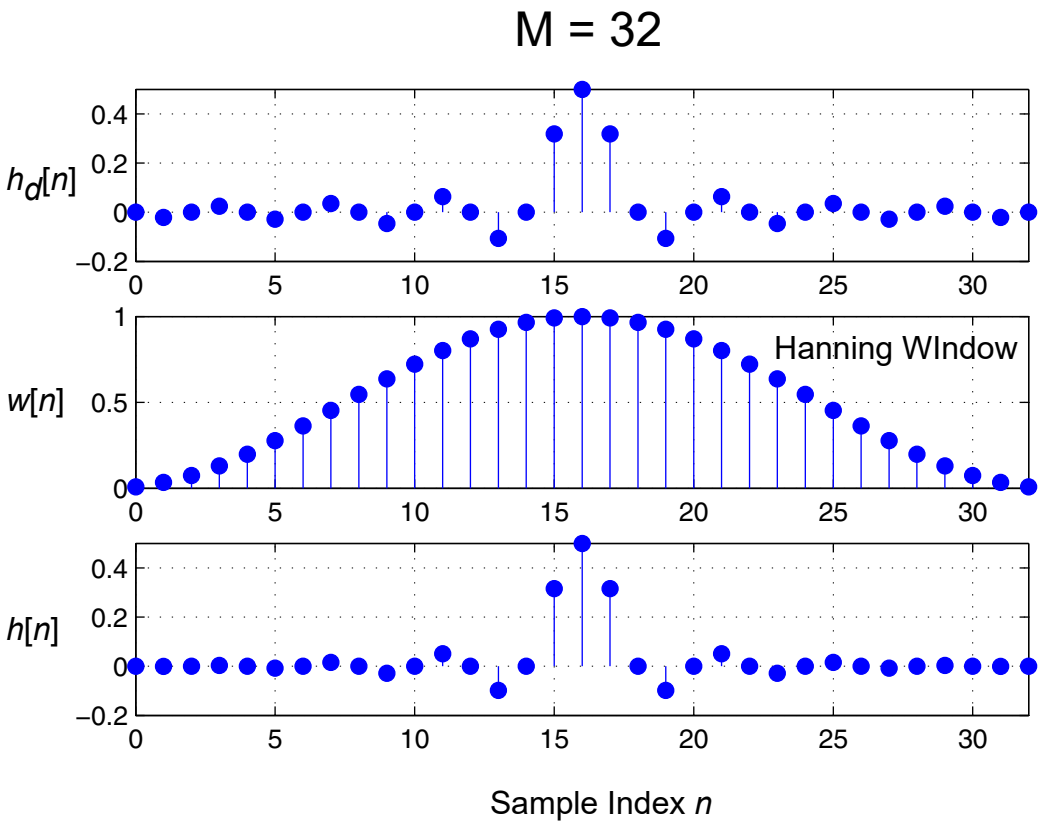
- Ultimately we will use MATLAB filter design functions, but for now consider a step-by-step approach:

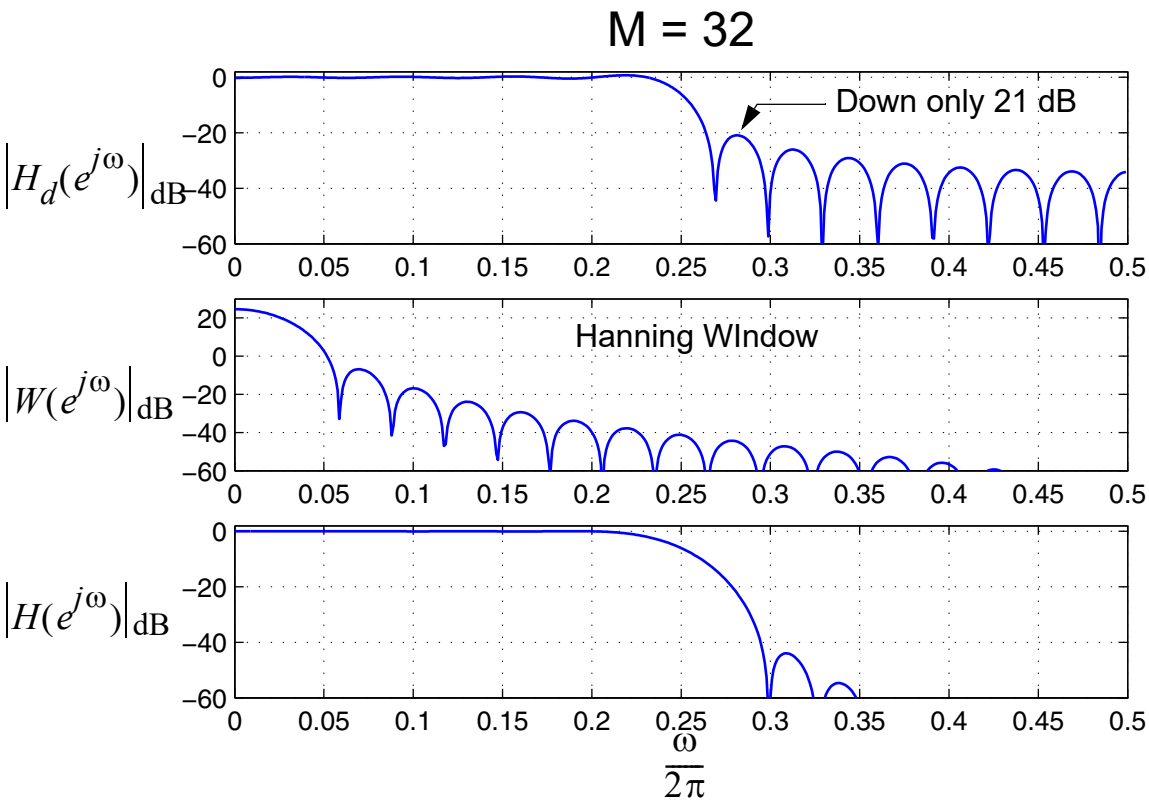
```
>> n = 0:32;
>> hd = pi/2/pi*sinc(pi/2*(n-32/2)/pi);
>> w = hanning(32+1);
>> h = hd.*w';
>> [Hd,F] = freqz(hd,1,512,1);
>> [W,F] = freqz(w,1,512,1);
>> [H,F] = freqz(hd.*w',1,512,1);
```

- The time domain waveforms:
- The frequency response:
- In general for an even symmetric design the frequency response is of the form

$$H(e^{j\omega}) = A_e(e^{j\omega})e^{-j\omega M/2} \tag{6.13}$$

where $A_e(e^{j\omega})$ is a real and even function of $\omega$


M = 32
$h_d[n]$
$w[n]$
Hanning WIndow
$h[n]$
Sample Index $n$

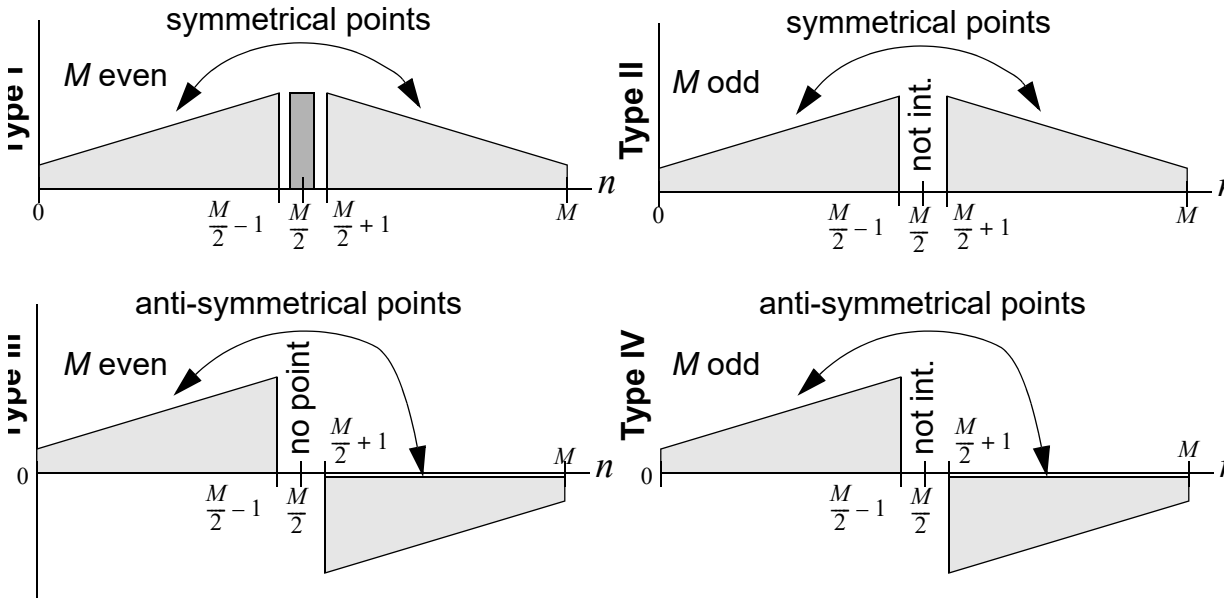

M = 32
$|H_d(e^{j\omega})|_{\text{dB}}$
Down only 21 dB
$|W(e^{j\omega})|_{\text{dB}}$
Hanning WIndow
$|H(e^{j\omega})|_{\text{dB}}$
$\frac{\omega}{2\pi}$

- For an odd symmetric design

$$H(e^{j\omega}) = jA_o(e^{j\omega})e^{-j\omega M/2} \tag{6.14}$$

where $A_e(e^{j\omega})$ is a real and even function of $\omega$

- Note that the symmetric case leads to type I or II generalized linear phase, while the antisymmetric case leads to type III or IV generalized linear phase




- For $h[n]$ symmetric about $M/2$ we can write the magnitude response $A_e(e^{j\omega})$ as

$$A_e(e^{j\omega}) = \int_{-\infty}^{\infty} H_e(e^{j\theta})W_e(e^{j(\omega-\theta)})d\theta \tag{6.15}$$

where $H_e(e^{j\omega})$ and $W_e(e^{j\omega})$ are real functions correspond-

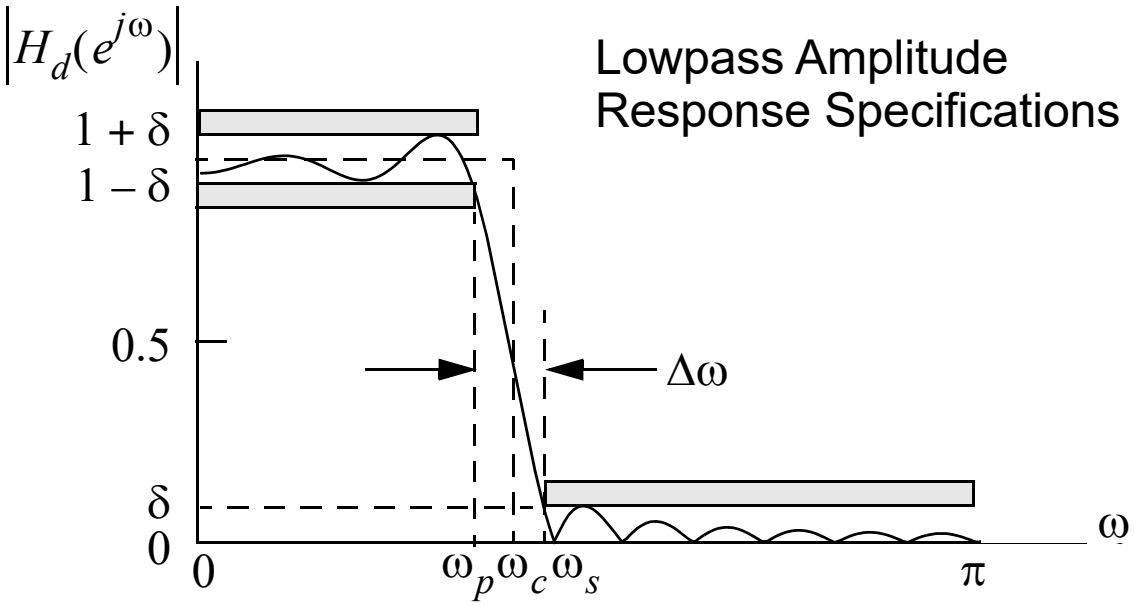
ing to the magnitude responses of the desired filter frequency response and window function frequency response respectively

## Lowpass Design

- For a lowpass design having linear phase we start with

$$h[n] = \frac{\sin[\omega_c(n - M/2)]}{\pi(n - M/2)} w[n] \qquad (6.16)$$

- The lowpass amplitude specifications of interest are




- The stopband attenuation is thus $A_s = -20\log\delta$ dB and the peak ripple is $\varepsilon_{dB} = 20\log(1 + \delta)$
- For popular window functions, such as rectangular, Bartlett, Hanning, Hamming, and Blackman, the relevant design data is contained in the following table (the Kaiser window function is defined in (6.19))

Table 1.1: Window characteristics for FIR design.

| Type | Transition Bandwidth $\Delta\omega$ | Minimum Stopband Attenuation | Equivalent Kaiser $\beta$ |
|---|---|---|---|
| Rectangle | $1.81\pi/M$ | 21 dB | 0 |
| Bartlett | $1.80\pi/M$ | 25 dB | 1.33 |
| Hanning | $5.01\pi/M$ | 44 dB | 3.86 |
| Hamming | $6.27\pi/M$ | 53 dB | 4.86 |
| Blackman | $9.19\pi/M$ | 74 dB | 7.04 |

**The general design steps:**

1. Choose the window function, $w[n]$, that just meets the stopband requirements as given in the Table 7.1
2. Choose the filter length, $M$, (actual length is $M+1$) such that

$$\Delta\omega \leq \omega_s - \omega_p \tag{6.17}$$

3. Choose $\omega_c$ in the truncated impulse response such that

$$\omega_c = \frac{\omega_p + \omega_s}{2} \tag{6.18}$$

4. Plot $\left|H(e^{j\omega})\right|$ to see if the specifications are satisfied
5. Adjust $\omega_c$ and $M$ if necessary to meet the requirements; If possible reduce $M$

- By using the Kaiser window method some of the *trial and error* can be eliminated since via parameter $\beta$, the stopband attenuation can be precisely controlled, and then the filter length for a desired transition bandwidth can be chosen

**Kaiser window design steps:**

1. Let $w[n]$ be a Kaiser window, i.e.,

$$w[n] = \begin{cases} I_0\left[\beta\left(1-\left[\dfrac{(n-M/2)}{M}\right]^2\right)^{1/2}\right], & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases} \tag{6.19}$$

2. Chose $\beta$ for the specified $A_s$ as

$$\beta = \begin{cases} 0.1102(A_s - 8.7), & A_s > 50\text{dB} \\ 0.5842(A_s-21)^{0.4} + 0.07886(A_s-21), & 21 \le A_s \le 50 \\ 0.0, & A_s \le 21 \end{cases} \tag{6.20}$$

3. The window length $M$ is then chosen to satisfy

$$M = \frac{A_s - 8}{2.285\Delta\omega} \tag{6.21}$$

4. The value chosen for $\omega_c$ is chosen as before

## Optimum Approximations of FIR Filters

- The window method is simple yet limiting, since we cannot obtain individual control over the approximation errors in

each band

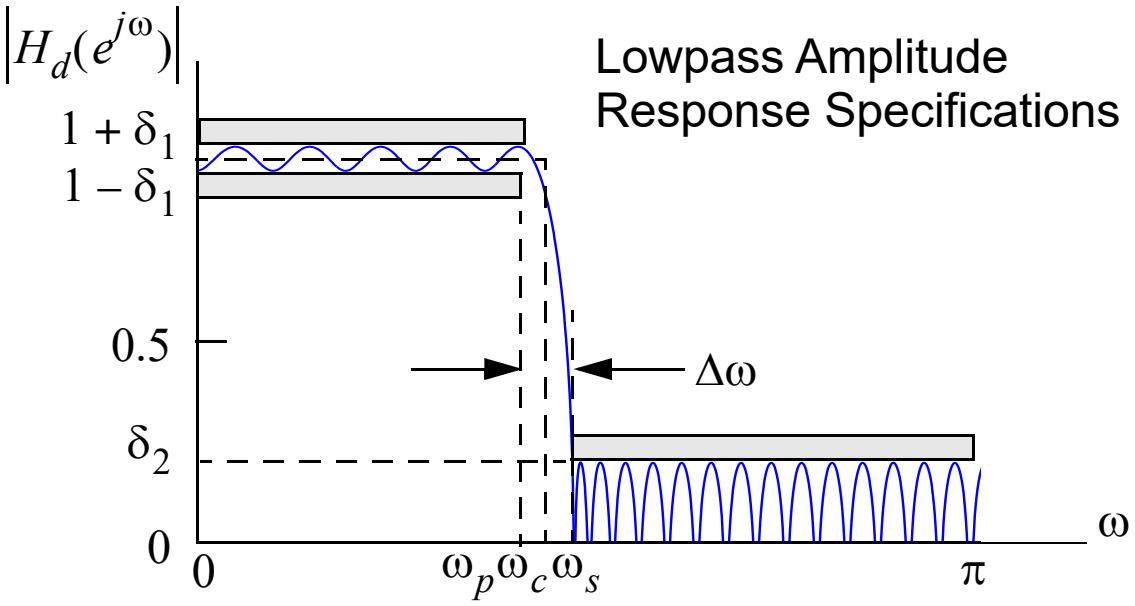
- The optimum FIR approach can be applied to type I, II, III, and IV generalized linear phase filters
- The basic idea is to minimize the weighted error between the desired response $H_d(e^{j\omega})$ and the actual response $H(e^{j\omega})$

$$E(\omega) = W(\omega)[H(e^{j\omega}) - H_d(e^{j\omega})] \tag{6.22}$$

- The classical algorithm for performing this minimization in an equiripple sense, was developed by Parks and McClellan in 1972
  - Equiripple means that equal amplitude error fluctuations are introduced in the minimization process
  - In MATLAB the associated function is `firpm()`
- Efficient implementation of the Parks McClellan algorithm requires the use of the *Remez exchange algorithm*
- For a detailed development of this algorithm see Oppenheim and Schafer[1]
- Multiple pass- and stopbands are permitted with this design formulation

1. A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, second edition, Prentice Hall, Upper Saddle River, New Jersey, 1999.

- Each band can have its own tolerance, or ripple specification, e.g., in a simple lowpass case we have




- An equiripple design example will be shown later

## MATLAB Basic Filter Design Functions

The following function list is a subset of the filter design functions contained in the MATLAB signal processing toolbox useful for FIR filter design. The function groupings match those of the toolbox manual.

| Filter Analysis/Implementation | |
|---|---|
| y = filter(b,a,x) | Direct form II filter vector x |
| [H,w] = freqz(b,a) | z-domain frequency response computation |

| Filter Analysis/Implementation (Continued) | |
|---|---|
| [Gpd,w] = grpdelay(b,a) | Group delay computation |
| h = impz(b,a) | Impulse response computation |
| unwrap | Phase unwrapping |
| zplane(b,a) | Plotting of the z-plane pole/zero map |

| Linear System Transformations | |
|---|---|
| residuez | z-domain partial fraction conversion |
| tf2zp | Transfer function to zero-pole conversion |
| zp2sos | Zero-pole to second-order biquadratic sections conversion |
| `tf2latc` | Transfer function to lattice conversion |

| FIR Filter Design | |
|---|---|
| fir1 | Window-based FIR filter design with standard response |
| kaiserord | Kaiser window FIR filter design estimation parameters with `fir1` |

| FIR Filter Design (Continued) | |
|---|---|
| fir2 | Window based FIR filter design with arbitrary sampled frequency response |
| firpm | Parks-McClellan optimal FIR filter design; optimal (equiripple) fit between the desired and actual frequency responses |
| firpmord | Parks-McClellan optimal FIR filter order estimation |

## Windowed FIR Design From Amplitude Specifications

$$\omega_p = 0.4\pi \text{ , } \omega_s = 0.6\pi \text{ and } \delta = 0.0032 \Rightarrow A_s = 50 \text{ dB}$$

- A Hamming window or a Kaiser window with a particular $\beta$ value will achieve the desired stop band attenuation

$$\Delta\omega = 0.2\pi \geq \frac{6.27\pi}{M}$$

or

$$M \geq \lceil 31.35 \rceil = 32$$

- The cutoff frequency is

$$\omega_c = \frac{0.2 + 0.3}{2} \times 2\pi = 0.25 \times 2\pi$$

- Another approach is to use a Kaiser window with

$$\beta = 0.5842(50-21)^{0.4} + 0.07886(50-21) = 4.5335$$

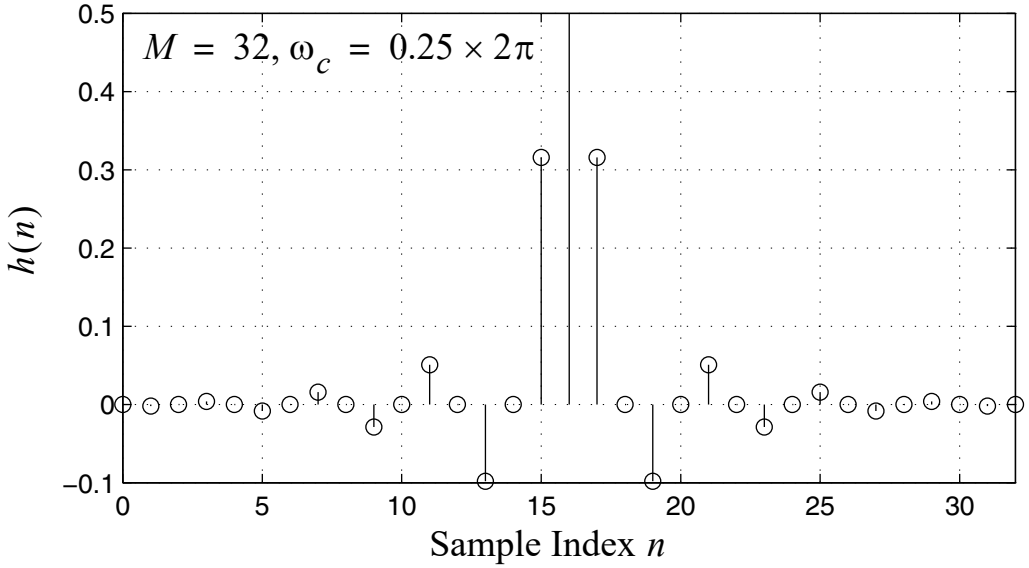
and

$$M = \left\lceil \frac{50-8}{2.285(0.2\pi)} \right\rceil = \lceil 29.25 \rceil = 30$$

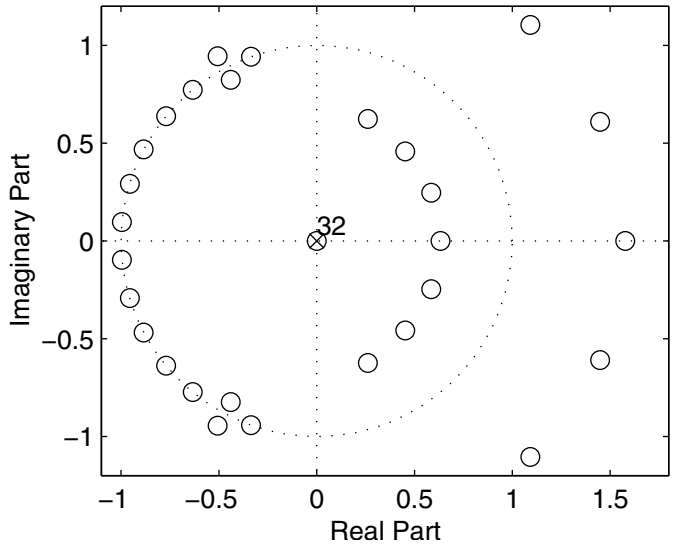
- Verification of these designs can be accomplished using the MATLAB filter design function `fir1()`

```
» % Hamming window design
» % (note fir1 uses hamming  by default)
» b = fir1(32,2*.25,hamming(32+1));
» stem(0:length(b)-1,b)
» grid
» zplane(b,1)
» [H,F] = freqz(b,1,512,1);
» plot(F,20*log10(abs(H)))
» axis([0 .5 -70 2])
» grid
» % Kasier window design
» bk = fir1(30,2*.25,kaiser(30+1,4.5335));
» [Hk,F] = freqz(bk,1,512,1);
» plot(F,20*log10(abs(Hk)))
» axis([0 .5 -70 2])
```

```
» grid
```



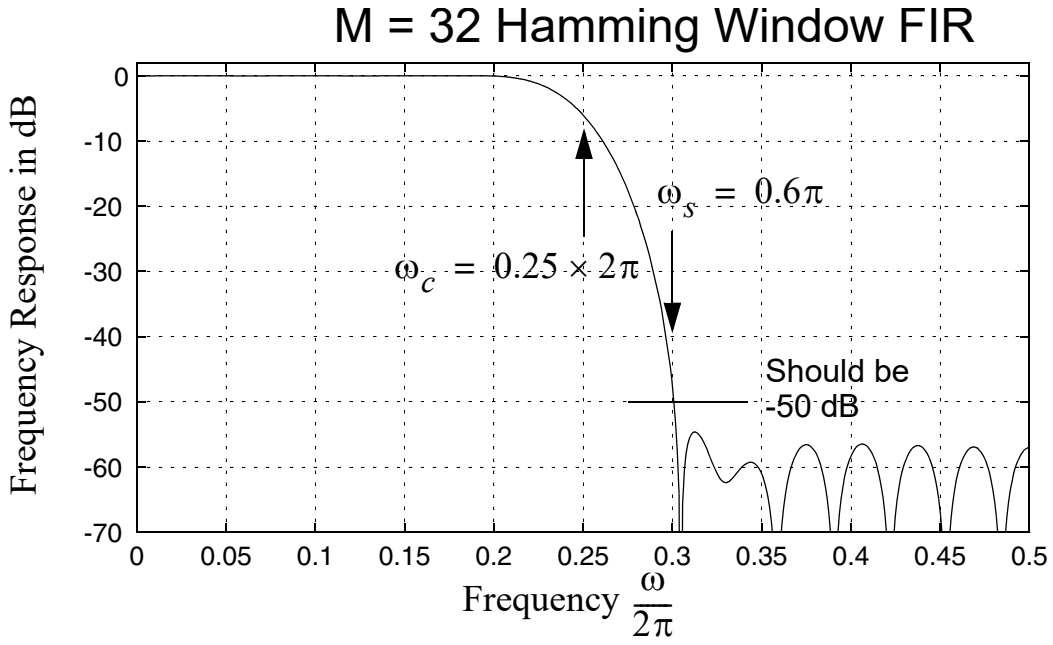

M = 32 impulse response with Hamming window



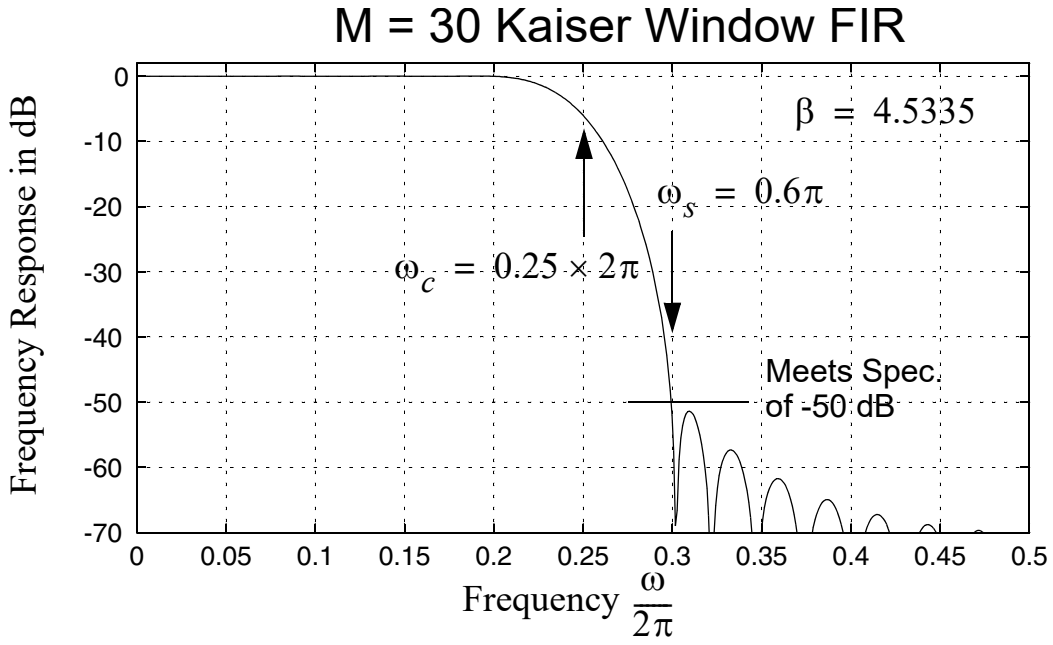

Note: There are few misplaced zeros in this plot due to MATLAB's problem finding the roots of a large polynomial.

Note: The many zero quadruplets that appear in this linear phase design.

M = 32 pole-zero plot with Hamming window




M = 32 frequency response with Hamming window




M = 30 frequency response with Kaiser window

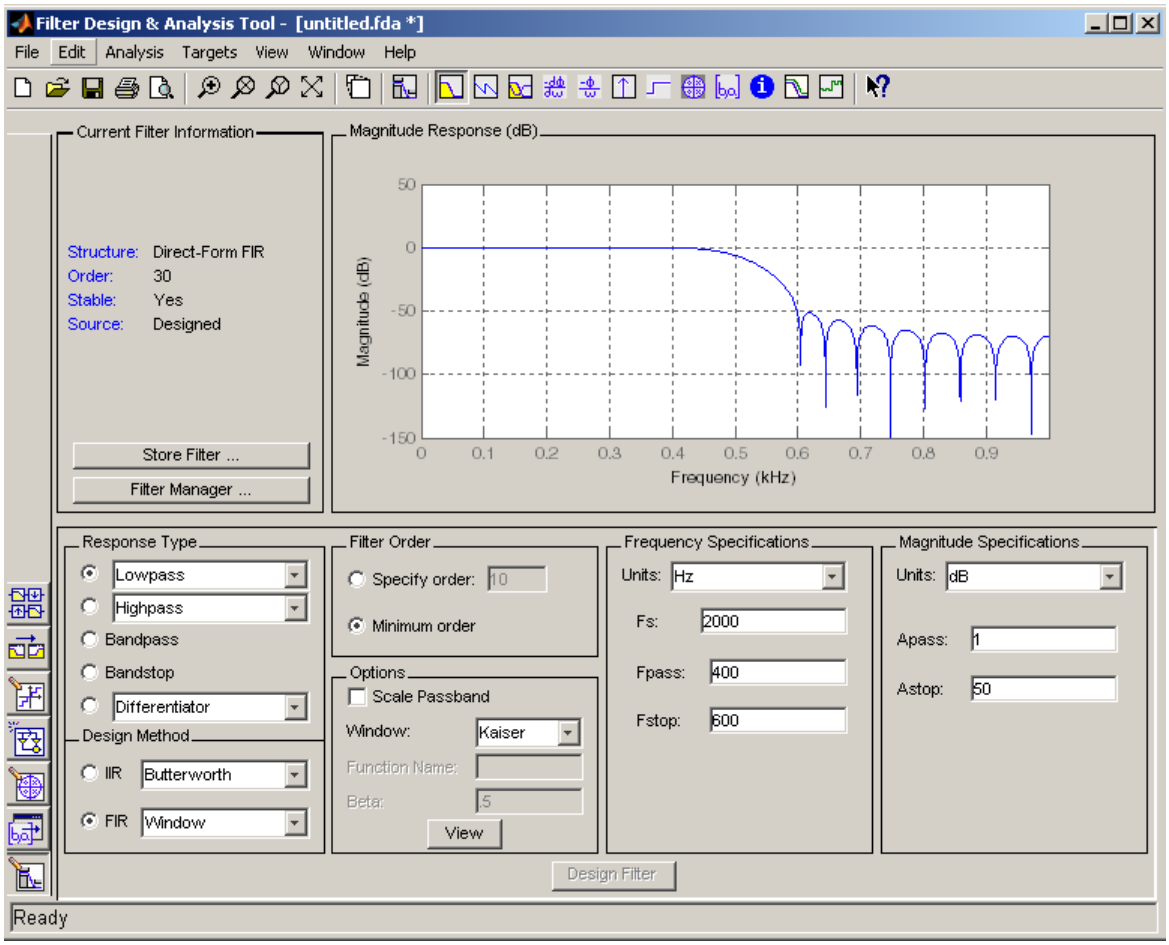
# Using MATLAB's `fdatool`

- Digital filter design from amplitude specifications to quantized filter coefficients can be accomplished using the MATLAB tool `fdatool`
  - Note: The basic `fdatool` is included with the signal processing toolbox, and when adding the associated *Fixed-Point Tool Box*, gives one the capability to perform additional filter designs and perform quantization design and analysis

## Example: Windowed FIR Design

- Consider an example that repeats the previous windowed FIR design using the Kaiser window
- In this example a Kaiser window linear phase FIR filter is designed assuming a sampling rate of 2000 Hz, so the folding frequency is 1000 Hz and the critical frequencies are

$f_p = 400\,\text{Hz}$ and $f_s = 600\,\text{Hz}$




- Many other filter viewing options are available as well
  - Display phase, Gain & phase, Group delay
  - Impulse response, Step response
  - Pole-zero plot
  - Filter coefficients

# Using scipy.signal for FIR Filter Design

- The Scipy `signal` package contains a number of useful FIR filter design functions
- A sampling taken from the documentation at `http://docs.scipy.org/doc/scipy/reference/generated/scipy.signal` is:

## scipy.signal.firwin

**scipy.signal.firwin(***numtaps, cutoff, width=None, window='hamming', pass_zero=True, scale=True, nyq=1.0***)** [source]

FIR filter design using the window method.

This function computes the coefficients of a finite impulse response filter. The filter will have linear phase; it will be Type I if *numtaps* is odd and Type II if *numtaps* is even.

Type II filters always have zero response at the Nyquist rate, so a ValueError exception is raised if firwin is called with *numtaps* even and having a passband whose right end is at the Nyquist rate.

**Parameters:**

**numtaps** *: int*
Length of the filter (number of coefficients, i.e. the filter order + 1). *numtaps* must be even if a passband includes the Nyquist frequency.

**cutoff** *: float or 1D array_like*
Cutoff frequency of filter (expressed in the same units as *nyq*) OR an array of cutoff frequencies (that is, band edges). In the latter case, the frequencies in *cutoff* should be positive and monotonically increasing between 0 and *nyq*. The values 0 and *nyq* must not be included in *cutoff*.

**width** *: float or None*
If *width* is not None, then assume it is the approximate width of the transition region (expressed in the same units as *nyq*) for use in Kaiser FIR filter design. In this case, the *window* argument is ignored.

**window** *: string or tuple of string and parameter values*
Desired window to use. See scipy.signal.get_window for a list of windows and required parameters.

**pass_zero** *: bool*
If True, the gain at the frequency 0 (i.e. the "DC gain") is 1. Otherwise the DC gain is 0.

**scale** *: bool*

Set to True to scale the coefficients so that the frequency response is exactly unity at a certain frequency. That frequency is either:

- 0 (DC) if the first passband starts at 0 (i.e. pass_zero is True)
- *nyq* (the Nyquist rate) if the first passband ends at *nyq* (i.e the filter is a single band highpass filter); center of first passband otherwise

**nyq** *: float*

Nyquist frequency. Each frequency in *cutoff* must be between 0 and *nyq*.

**Returns:**

**h** *: (numtaps,) ndarray*

Coefficients of length *numtaps* FIR filter.

**Raises:**

**ValueError**

If any value in *cutoff* is less than or equal to 0 or greater than or equal to *nyq*, if the values in *cutoff* are not strictly monotonically increasing, or if *numtaps* is even but a passband includes the Nyquist frequency.

## scipy.signal.firwin2

**scipy.signal.firwin2(***numtaps, freq, gain, nfreqs=None, window='hamming', nyq=1.0, antisymmetric=False***)** [source]

FIR filter design using the window method.

From the given frequencies *freq* and corresponding gains *gain*, this function constructs an FIR filter with linear phase and (approximately) the given frequency response.

See the on-line scipy.signal help for more information

## scipy.signal.kaiserord

**scipy.signal.kaiserord(***ripple, width***)** [source]

Design a Kaiser window to limit ripple and width of transition region.

See the on-line scipy.signal help for more information

## scipy.signal.kaiser_beta

**scipy.signal.kaiser_beta(***a***)** [source]

Compute the Kaiser parameter *beta*, given the attenuation *a*.

See the on-line scipy.signal help for more information

## scipy.signal.kaiser_atten

**scipy.signal.kaiser_atten(***numtaps, width***)** [source]

Compute the attenuation of a Kaiser FIR filter.

Given the number of taps *N* and the transition width *width*, compute the attenuation *a* in dB, given by Kaiser's formula:

```
a = 2.285 * (N - 1) * pi * width + 7.95
```

See the on-line scipy.signal help for more information

# scipy.signal.remez

**scipy.signal.remez(***numtaps, bands, desired, weight=None, Hz=1, type='bandpass', maxiter=25, grid_density=16***)** [source]

Calculate the minimax optimal filter using the Remez exchange algorithm.

Calculate the filter-coefficients for the finite impulse response (FIR) filter whose transfer function minimizes the maximum error between the desired gain and the realized gain in the specified frequency bands using the Remez exchange algorithm.

**Parameters:**

**numtaps** *: int*
The desired number of taps in the filter. The number of taps is the number of terms in the filter, or the filter order plus one.

**bands** *: array_like*
A monotonic sequence containing the band edges in Hz. All elements must be non-negative and less than half the sampling frequency as given by *Hz*.

**desired** *: array_like*
A sequence half the size of bands containing the desired gain in each of the specified bands.

**weight** *: array_like, optional*
A relative weighting to give to each band region. The length of *weight* has to be half the length of *bands*.

**Hz** *: scalar, optional*
The sampling frequency in Hz. Default is 1.

**type** *: {'bandpass', 'differentiator', 'hilbert'}, optional*
The type of filter:
'bandpass' : flat response in bands. This is the default.
'differentiator' : frequency proportional response in bands.

***'hilbert' : filter with odd symmetry, that is, type III***
(for even order) or type IV (for odd order) linear phase filters.

**maxiter** *: int, optional*
Maximum number of iterations of the algorithm. Default is 25.

**grid_density** *: int, optional*
Grid density. The dense grid used in remez is of size `(numtaps + 1) * grid_density`. Default is 16.

**Returns:**

**out** *: ndarray*
A rank-1 array containing the coefficients of the optimal (in a minimax sense) filter.

## Example: Design in Python for float32_t and int16_t Implementation

- We can also use Python, specifically `scipy.signal` to design digital filters
- Once the filter is design we can convert it to `float32_t` or `int16_t` as needed
- Functions for writing coefficient header files are also available
- All work is done in an Jupyter notebook (ECE 5655 Chapter 6) available on the course Web Site
- Design a windowed FIR lowpass filter having 31 taps:

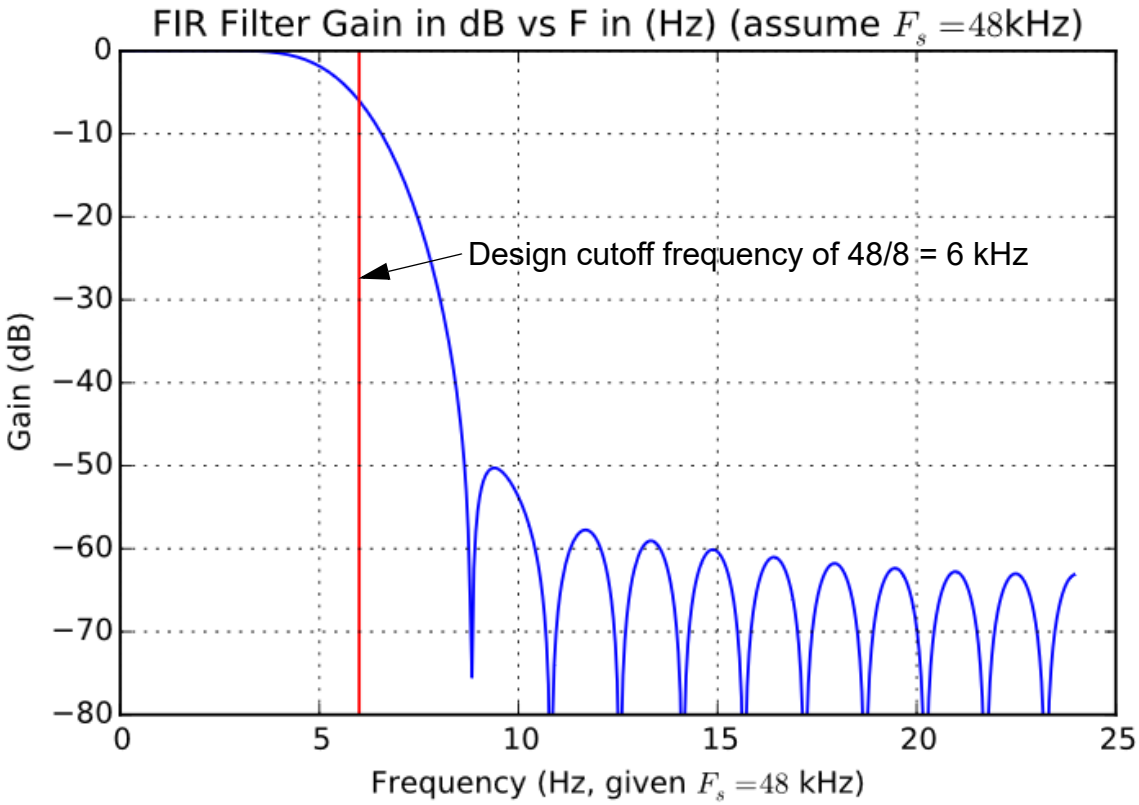
```
# N_taps, 2*Fc/Fs = 2*f_c
b = signal.firwin(31,2*1/8)
b
```

```
array([ -1.20388000e-03,  -2.05336094e-03,  -2.07962901e-03,
         1.64050411e-18,   4.76490069e-03,   9.89603364e-03,
         9.97846427e-03,  -4.80775224e-18,  -1.89637895e-02,
        -3.62933212e-02,  -3.47620350e-02,   8.28597812e-18,
         6.86322821e-02,   1.53265764e-01,   2.23458463e-01,
         2.50720214e-01,   2.23458463e-01,   1.53265764e-01,
         6.86322821e-02,   8.28597812e-18,  -3.47620350e-02,
        -3.62933212e-02,  -1.89637895e-02,  -4.80775224e-18,
         9.97846427e-03,   9.89603364e-03,   4.76490069e-03,
         1.64050411e-18,  -2.07962901e-03,  -2.05336094e-03,
        -1.20388000e-03])
```

- Using the full `float64_t` precision the filter frequency

response in dB, assuming $F_s = 48\,\text{kHz}$, is:

```
f = arange(0,0.5,.001)
w,B = signal.freqz(b,1,2*pi*f)
plot(f*48,20*log10(abs(B)));
plot([48/8,48/8],[-80,0],'r')
ylim([-80,0])
title(r'FIR Filter Gain in dB vs F in (Hz) (assume $F_s = 48$kHz)
ylabel(r'Gain (dB)')
xlabel(r'Frequency (Hz, given $F_s = 48$ kHz)')
grid();
```




- Fixed-point coefficients using 16-bit signed integers can be obtained using scaling, rounding, and quantizing:

```
b = signal.firwin(31,2*1/8)
int16(rint(b*2**15))
```

```
array([  -39,   -67,   -68,     0,   156,   324,   327,     0,  -621,
       -1189, -1139,     0,  2249,  5022,  7322,  8216,  7322,  5022,
        2249,     0, -1139, -1189,  -621,     0,   327,   324,   156,
           0,   -68,   -67,   -39], dtype=int16)
```

## Implementation In C

- CMSIS-DSP provided the needed functions for `float32_t` and `int16_t` implementations
- In this first example we have written custom functions in the code module `FIR_filters.c`/`FIR_filters.h`

```
//FIR_filters.h
// FIR Filters header
// Mark Wickert April 2015

#define ARM_MATH_CM4
#include <s6e2cc.h>
#include "arm_math.h"

/*
Structures to hold FIR filter state information. Two direct form FIR
types are implemented at present: (1) float32_t and (2) int16_t.
Each type requires both an initialization function and a filtering
function. The functions feature both sample-based and frame-based
capability.

More FIR implementation types can/should be added, say for a folded design,
use of intrinsics in the fixed point version. The use of a circular buffer
in all versions.
*/

struct FIR_struct_float32
{
  int16_t M_taps;
  float32_t *state;
  float32_t *b_coeff;
};

struct FIR_struct_int16
{
  int16_t M_taps;
  int16_t *state;
  int16_t *b_coeff;
};

void FIR_init_float32(struct FIR_struct_float32 *FIR, float32_t *state,
                      float32_t *b_coeff, int16_t Nb);
void FIR_filt_float32(struct FIR_struct_float32 *FIR, float32_t *x_in,
                      float32_t *x_out, int16_t Nframe);
void FIR_init_int16(struct FIR_struct_int16 *FIR, int16_t *state,
                    int16_t *b_coeff, int16_t Nb);
```

```
void FIR_filt_int16(struct FIR_struct_int16 *FIR, int16_t *x_in,
                    int16_t *x_out, int16_t Nframe, int16_t scale);

//FIR_filters.c
// FIR Filters Implementation
// Mark Wickert April 2015

#include "FIR_filters.h"

void FIR_init_float32(struct FIR_struct_float32 *FIR, float32_t *state,
                      float32_t *b_coeff, int16_t Nb) {
  // Load the filter coefficients and initialize the filter state.
  int16_t iFIR;
  FIR->M_taps = Nb;
  FIR->b_coeff = b_coeff;
  FIR->state = state;
  for (iFIR = 0; iFIR < FIR->M_taps; iFIR++)
  {
    FIR->state[iFIR] = 0;
  }
}

void FIR_filt_float32(struct FIR_struct_float32 *FIR, float32_t *x_in,
                      float32_t *x_out, int16_t Nframe) {
 int16_t iframe;
 int16_t iFIR;
 float32_t accum;

  //Process each sample of the frame with this loop
  for (iframe = 0; iframe < Nframe; iframe++)
  {
    // Clear the accumulator/output before filtering
    accum = 0;
    // Place new input sample as first element in the filter state array
    FIR->state[0] = x_in[iframe];
    //Direct form filter each sample using a sum of products
    for (iFIR = 0; iFIR < FIR->M_taps; iFIR++)
    {
       accum += FIR->state[iFIR]*FIR->b_coeff[iFIR];
    }
    x_out[iframe] = accum;
    // Shift filter states to right or use circular buffer
    for (iFIR = FIR->M_taps-1; iFIR > 0; iFIR--)
    {
      FIR->state[iFIR] = FIR->state[iFIR-1];
    }
  }
}
```

```
void FIR_init_int16(struct FIR_struct_int16 *FIR, int16_t *state,
                        int16_t *b_coeff, int16_t Nb) {
  // Load the filter coefficients and initialize the filter state.
  int16_t iFIR;
  FIR->M_taps = Nb;
  FIR->b_coeff = b_coeff;
  FIR->state = state;
  for (iFIR = 0; iFIR < FIR->M_taps; iFIR++)
  {
    FIR->state[iFIR] = 0;
  }
}

void FIR_filt_int16(struct FIR_struct_int16 *FIR, int16_t *x_in,
                     int16_t *x_out, int16_t Nframe, int16_t scale) {
 int16_t iframe;
 int16_t iFIR;
 int32_t accum; // Accumulate sum of products in long integer

  for (iframe = 0; iframe < Nframe; iframe++)
  {
    // Clear the accumulator/output before filtering
    accum = 0;
    // Place new input sample as first element in the filter state array
    FIR->state[0] = x_in[iframe];
    for (iFIR = 0; iFIR < FIR->M_taps; iFIR++)
    {
       accum += (int32_t) FIR->state[iFIR]*FIR->b_coeff[iFIR];
    }
    x_out[iframe] = (int16_t)(accum>>scale); //nominally 15
    // Shift filter states to right or use circular buffer
    for (iFIR = FIR->M_taps-1; iFIR > 0; iFIR--)
    {
      FIR->state[iFIR] = FIR->state[iFIR-1];
    }
  }
}
```

- A data structure is used to conveniently hold the filter coefficients and filter state required for the direct form structure topology
- The heart of the implementation for both code versions, float and fixed, are the actual filtering functions `FIR_filt_float32` and `FIR_filt_int16`

- Homework problems will require some enhancements to both versions in an attempt to increase performance
- More ... TBD
- A top level module that makes use of the FIR filters is shown below:

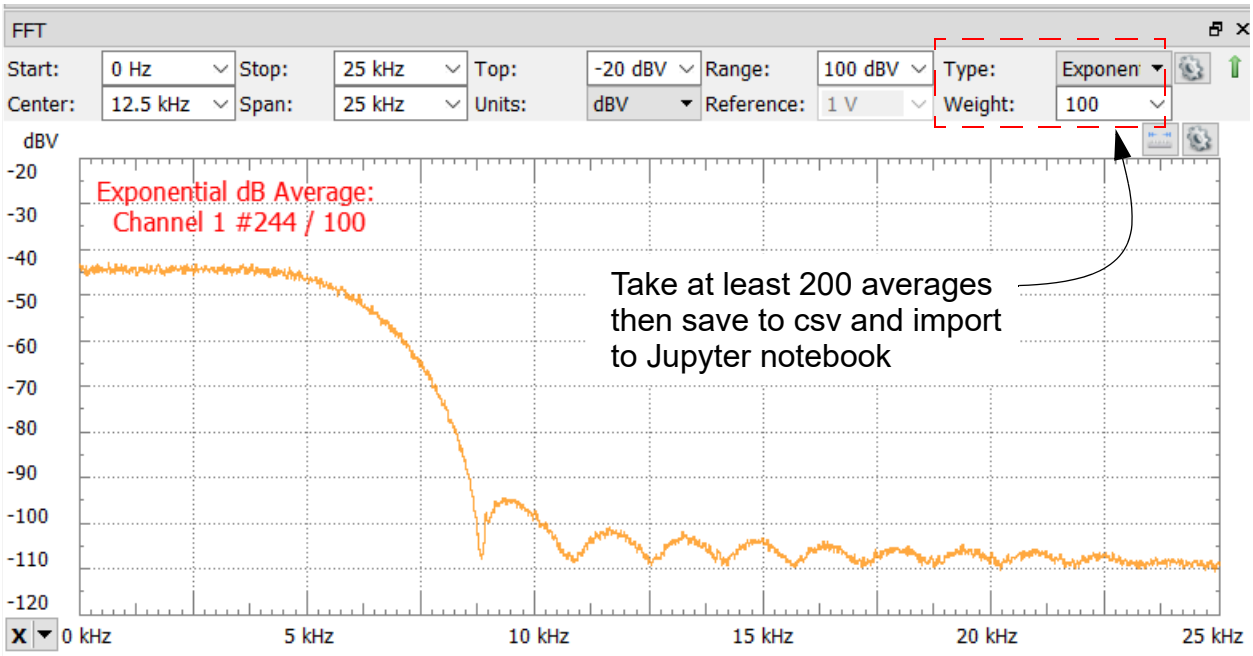
```
// FM4_FIR_intr.c

#include "audio.h"
#include "FM4_slider_interface.h"

#include "FIR_filters.h"
//#include "s4_p1_remez.h"
//#include "s4_p1_remez_f32.h"

int32_t rand_int32(void);


// Create (instantiate) GUI slider data structure
struct FM4_slider_struct FM4_GUI;

// Create CMSIS-DSP filter instances
struct FIR_struct_float32 FIR1;
//struct FIR_struct_int16 FIR2;
arm_fir_instance_q15 ARM_FIR2;
//float32_t state4[4];
//float32_t state8[8];
float32_t state31[31];
//float32_t b_MA4[4] = {1.0f,1.0f,1.0f,1.0f};
//float32_t b_MA8[8] = {0.125,0.125,0.125,0.125,0.125,0.125,0.125,0.125};

float32_t b_31[31] = { -1.20388000e-03,  -2.05336094e-03,  -2.07962901e-03,
                        1.64050411e-18,   4.76490069e-03,   9.89603364e-03,
                        9.97846427e-03,  -4.80775224e-18,  -1.89637895e-02,
                       -3.62933212e-02,  -3.47620350e-02,   8.28597812e-18,
                        6.86322821e-02,   1.53265764e-01,   2.23458463e-01,
                        2.50720214e-01,   2.23458463e-01,   1.53265764e-01,
                        6.86322821e-02,   8.28597812e-18,  -3.47620350e-02,
                       -3.62933212e-02,  -1.89637895e-02,  -4.80775224e-18,
                        9.97846427e-03,   9.89603364e-03,   4.76490069e-03,
                        1.64050411e-18,  -2.07962901e-03,  -2.05336094e-03,
                       -1.20388000e-03};

//int16_t state8q[8];
//int16_t state31q[31];
```

```
//float32_t state78[78];
//int16_t state78q[78];
int16_t b_MA8q[8] = {1024,1024,1024,1024,1024,1024,1024,1024};
/*
int16_t b_31q[31] = {  -39,  -67,  -68,    0,  156,  324,  327,    0, -621,
                     -1189,-1139,    0, 2249, 5022, 7322, 8216, 7322, 5022,
                      2249,    0,-1139,-1189, -621,    0,  327,  324,  156,
                         0,  -68,  -67,  -39};
*/

void I2S_HANDLER(void) {   /****** I2S Interruption Handler *****/

  int16_t left_out_sample = 0;
  int16_t right_out_sample = 0;
  int16_t left_in_sample = 0;
  int16_t right_in_sample = 0;
  float32_t x, y;

      //gpio_toggle(TEST_PIN);
      //gpio_set(TEST_PIN, HIGH);
      gpio_set(P1C, HIGH);

      audio_IN = i2s_rx();
      //Separate 16 bits channel left
      left_in_sample = (audio_IN & 0x0000FFFF);
      //Separate 16 bits channel right
      right_in_sample = ((audio_IN >>16)& 0x0000FFFF);

      if (FM4_GUI.P_vals[3] < 1)
      {
            x = (float32_t) left_in_sample;
      }
      else
      {
            x = (float32_t)(((int16_t)rand_int32())>>2);
      }
      gpio_set(P1B, HIGH);
      FIR_filt_float32(&FIR1,&x,&y,1);
      gpio_set(P1B, LOW);
      //FIR_filt_int16(&FIR2,&left_in_sample,&left_out_sample,1,15);
      //FIR_filt_int16_folded(&FIR2,&left_in_sample,&left_out_sample,1,15);
      //FIR_filt_int16_cb(&FIR2,&left_in_sample,&left_out_sample,1,15);
      //arm_fir_fast_q15(&ARM_FIR2,&left_in_sample,&left_out_sample,1);
      //arm_fir_q15(&ARM_FIR2,&left_in_sample,&left_out_sample,1);
      left_out_sample = (int16_t) (FM4_GUI.P_vals[0]*y);
      //left_out_sample = (int16_t) (P_vals[0]*left_out_sample);

      // No filtering on right input/output samples
      right_out_sample = (int16_t)(FM4_GUI.P_vals[1]*right_in_sample);
      //Put the two channels together again
```

```
      audio_OUT = ((right_out_sample<<16 & 0xFFFF0000))
                  + (left_out_sample & 0x0000FFFF);
      i2s_tx(audio_OUT);
      gpio_set(P1C, LOW);
}

int main(void)
{
      gpio_set_mode(TEST_PIN,Output); //TEST_PIN = P10
      gpio_set_mode(P1B,Output); //D0 = P1B
      gpio_set_mode(P1C,Output); //D1 = P1C
      gpio_set_mode(PF7,Output); //D2 = PF7

      // Initialize the slider interface by setting the baud rate
      // (460800 or 921600)
      // and initial float values for each of the 6 slider parameters
      init_slider_interface(&FM4_GUI,460800, 1.0, 1.0, 0.0,
                            0.0, 0.0, 0.0);

      // Send a string to the terminal
      write_uart0("Hello FM4 World!\n");
      //Initialize Filter(s)
      //FIR_init_float32(&FIR1,state4,b_MA4,4);
      //FIR_init_float32(&FIR1,state8,b_MA8,8);
      FIR_init_float32(&FIR1,state31,b_31,31);
      //FIR_init_float32(&FIR1,state78,h_FIR,78);
      //FIR_init_int16(&FIR2,state78q,h_FIR,M_FIR);
      //arm_fir_init_q15(&ARM_FIR2,78,h_FIR,state78q,1);

      //audio_init ( hz48000, mic_in, intr, I2S_HANDLER);
      audio_init ( hz48000, line_in, intr, I2S_HANDLER);

  while(1)
  {
    // Update slider parameters
    update_slider_parameters(&FM4_GUI);
  }
}

int32_t rand_int32(void)
{
  static int32_t a_start = 100001;

  a_start = (a_start*125) % 2796203;
  return a_start;
}
```

## Testing the float and fixed-Point Versions

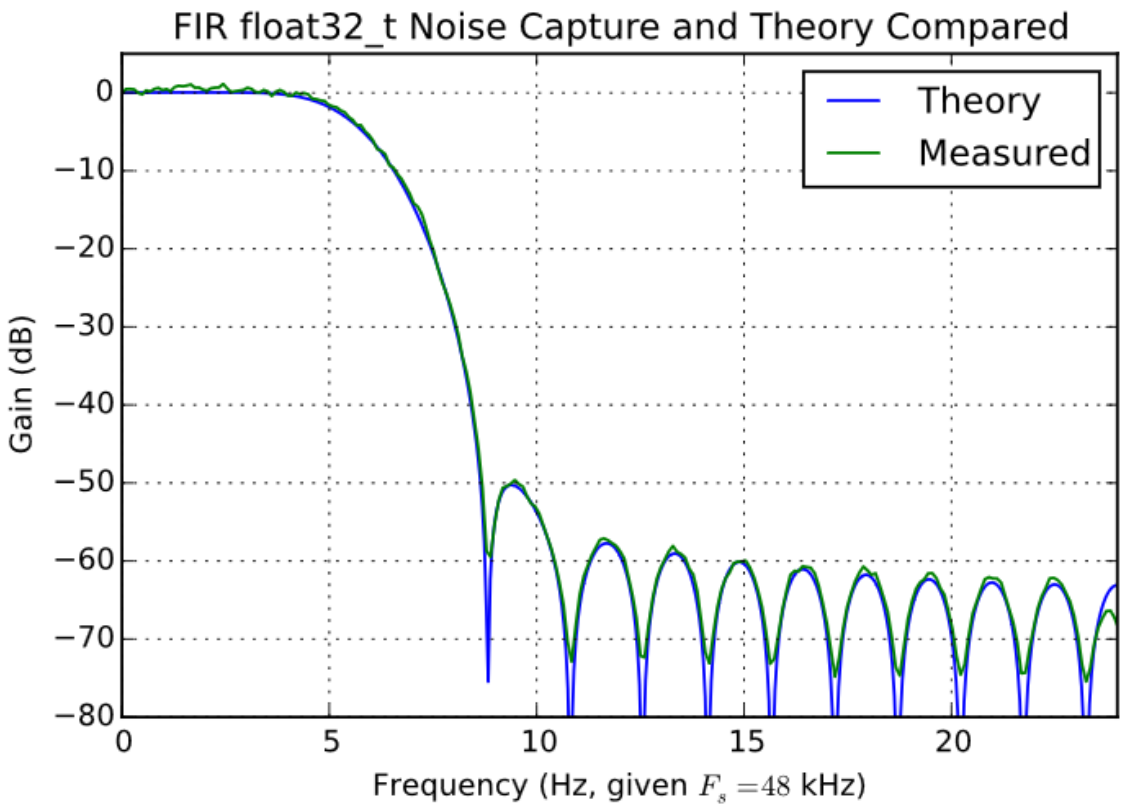
- The Analog Discovery was used to perform testing, in particular testing with a noise input and FFT expo. avg. and comparing with the expected results obtained from Python




- **Float version**:

```
fs_48,P31_float = loadtxt('spec_noise_b31_fs48.csv',delimiter=',',
                                skiprows=1,usecols=(0,1),unpack=True)
```

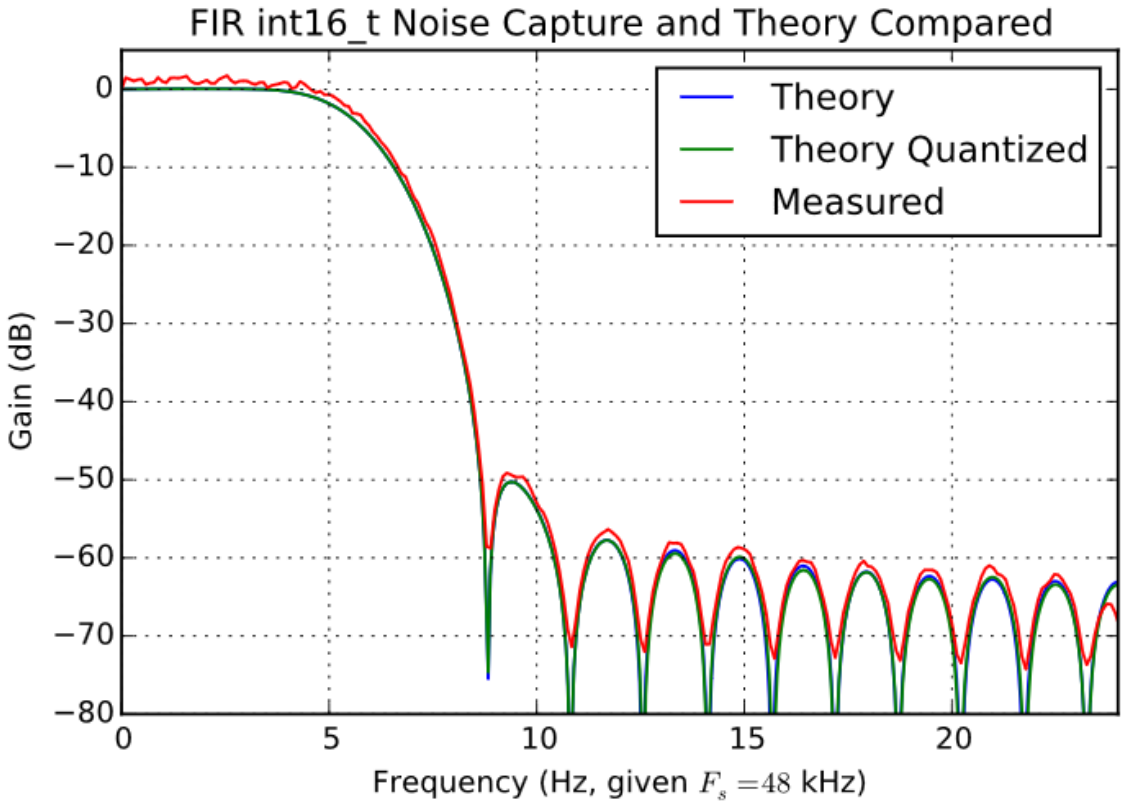
```
f = arange(0,1.0,.001)
w,B = signal.freqz(b,1,2*pi*f)
plot(f*48,20*log10(abs(B)));
plot(fs_48[:340]/1000,P31_float[:340]-P31_float[0])
ylim([-80,5])
xlim([0,48/2])
title(r'FIR float32_t Noise Capture and Theory Compared')
ylabel(r'Gain (dB)')
xlabel(r'Frequency (Hz, given $F_s = 48$ kHz)')
legend((r'Theory',r'Measured'),loc='best')
grid();
```




- ISR execution time on FM4 for FIR alone is, 5.04 $\mu s$
- **Fixed-point version**:

```
fs_48fix,P31_fix = loadtxt('spec_noise_b31_fs48fix.csv',delimiter=',',
                           skiprows=1,usecols=(0,1),unpack=True)
```

```
f = arange(0,1.0,.001)
w,B = signal.freqz(b,1,2*pi*f)
plot(f*48,20*log10(abs(B)));
w,Bq = signal.freqz(rint(b*2**15)/2**15,1,2*pi*f)
plot(f*48,20*log10(abs(Bq)));
plot(fs_48[:340]/1000,P31_fix[:340]-P31_fix[0])
ylim([-80,5])
xlim([0,48/2])
title(r'FIR int16_t Noise Capture and Theory Compared')
ylabel(r'Gain (dB)')
xlabel(r'Frequency (Hz, given $F_s = 48$ kHz)')
legend((r'Theory',r'Theory Quantized',r'Measured'),loc='best')
grid();
```




- ISR execution time, all operations for one channel, 5.22 $\mu s$

### Equal Ripple Design using remez() in Python

-

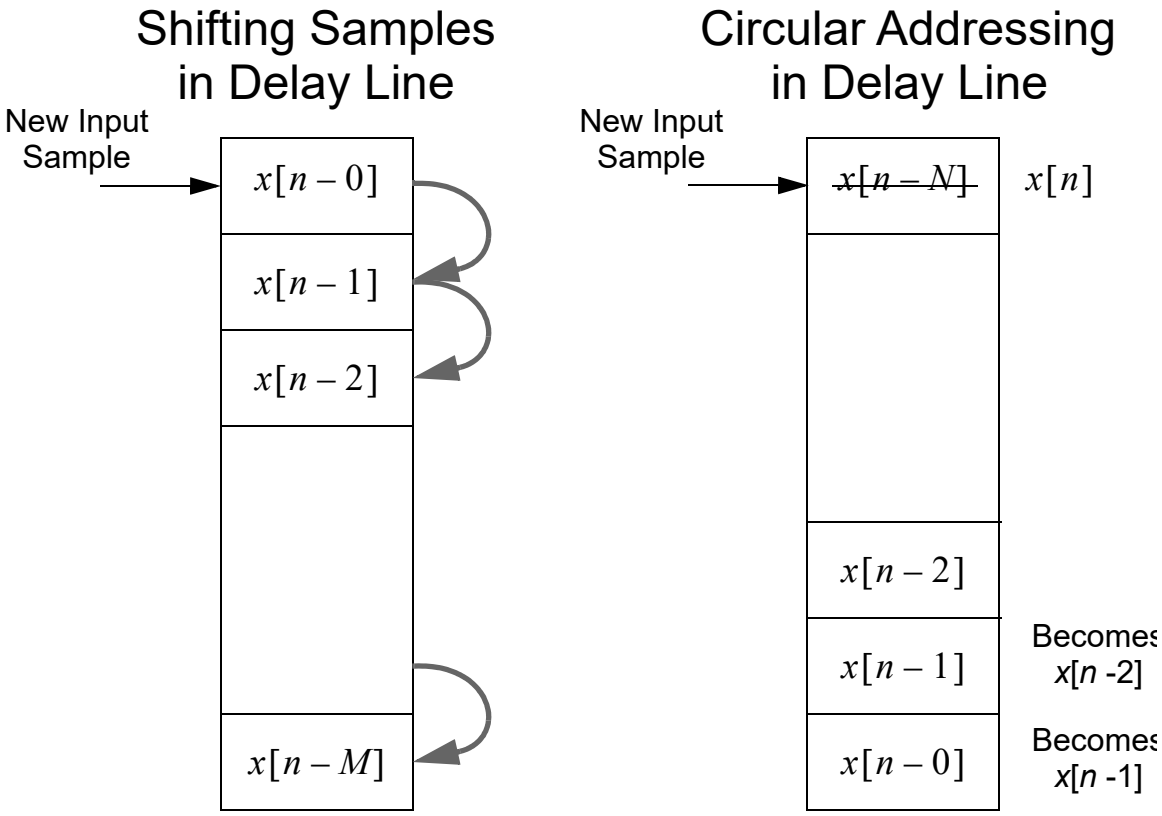
# Circular Addressing

### Circular Buffer in C

- Up to this point the filter memory or history was managed using a linear buffer array in the fixed and float example routines given earlier
- A more efficient implementation of this buffer (tapped delay

line) is to simply overwrite the oldest value of the array with the newest values

- Proper access to the data is accomplished with a pointer that wraps modulo the buffer size




## C Circular Buffer[1]

- A circular buffer can be directly implemented in C
- A good discussion can be found in Yiu in Chapter 21.5.3, which is a section dedicated to FIR filters optimization for the Cortex-M4

1. p.709 J. Yiu, The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition, Newnes, 2014.

- 

**int16_t** x_buffer[N_FIR]; //buffer for delay samples

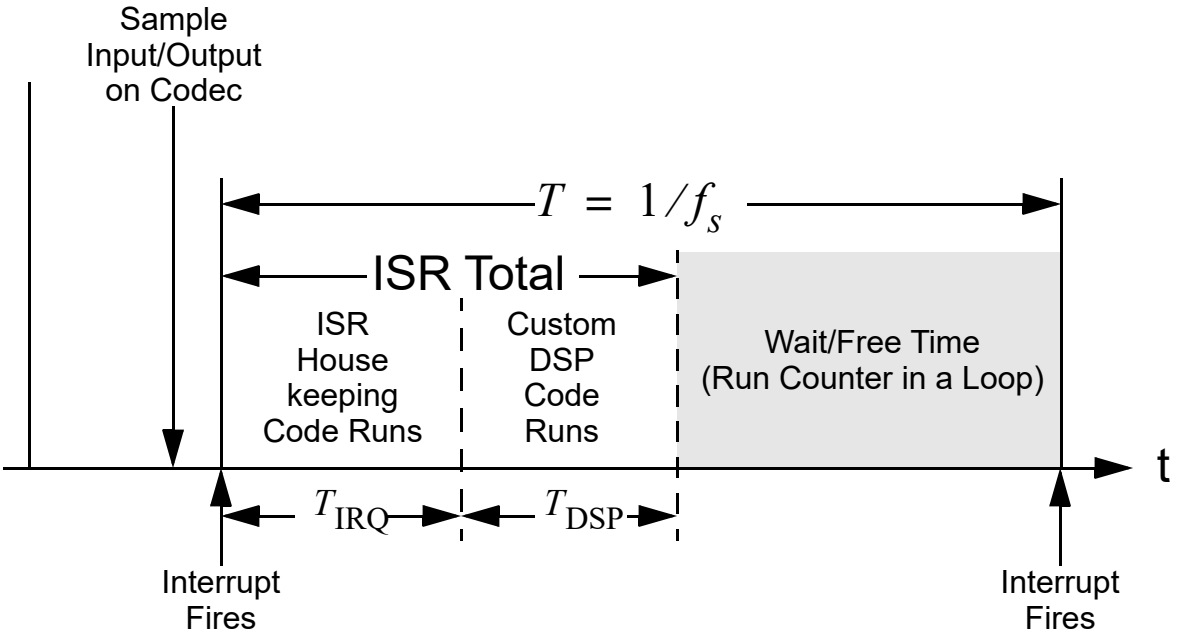
**int16_t** newest, x_index; //circular buffer index control

- The process continues with newest sample location moving around the buffer in a circular fashion
- Code example from Yiu p.711

```
// Block-based FIR filter.
// N equals the length of the filter (number of taps)
// blockSize equals the number of samples to process
// state[] is the state variable buffer and contains the previous N
// samples of the input
// stateIndex points to the oldest sample in the state buffer. It
// will be overwritten with the most recent input sample.
// coeffs[] holds the N coefficients
// inPtr and outPtr point to the input and output buffers, respectively
for(sample=0;sample<blockSize;sample++)
{
  // Copy the new sample into the state buffer and then
  circularly wrap stateIndex
  state[stateIndex++] = inPtr[sample]
  if (stateIndex >= N)
    stateIndex = 0;
  sum = 0.0f;
  for(i=0;i<N;i++)
  {
    sum += state[stateIndex++] * coeffs[N-i];
    if (stateIndex >= N)
      stateIndex = 0;
  }
  outPtr[sample] = sum;
}
```

# Performance of Interrupt Driven Programs

- Pictorially we are doing the following:




The Wait Loop Counter Scheme

# Useful Resources

- TBD