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 AIC23 codec interface and associated ISR 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

![Diagram of C/D to D/C system with continuous-time and discrete-time inputs and outputs](image)

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)

\[
H_{\text{eff}}(j\Omega) = \begin{cases} 
H(e^{j\Omega T}), & |\Omega| < \pi/T \\
0, & \text{otherwise}
\end{cases} \quad (7.1)
\]

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.,

\[
H(e^{j\omega}) = H_{\text{eff}}\left(j\frac{\omega}{T}\right), \ |\omega| < \pi \quad (7.2)
\]
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 \textit{brickwall} responses using window functions
  – Frequency sampling desired response using transition samples
  – Optimum equiripple approximations (use the \textit{Parks-McClellan algorithm} via the \textit{Remez exchange algorithm}, \texttt{firpm()} in MATLAB)
  – Minimum mean-square error in the frequency domain

• In the FIR approaches the additional constraint of \textit{linear phase} is usually imposed
Basic FIR Filter Topologies

- Recall that a causal FIR filter containing \( M + 1 \) coefficients has impulse response

\[
h[n] = \sum_{k=0}^{M} a_k \delta[n-k]
\]  
(7.3)

- The corresponding \( z \)-transform of the filter is

\[
H(z) = \sum_{k=0}^{M} a_k z^{-k} = a_0 + a_1 z^{-1} + \ldots + a_{M-1} z^{-M}
\]  
(7.4)

- The frequency response of the filter is

\[
H(e^{j\omega}) = \sum_{k=0}^{M} a_k e^{-j\omega k}
\]  
(7.5)

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

\[\text{Direct Form FIR Structure}\]

- Often times we are dealing with linear phase FIR designs which have even or odd symmetry, e.g.,
\[ h[n] = h[M - n] \text{ even} \]
\[ h[n] = -h[M - n] \text{ odd} \] (7.6)

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

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

Modified Direct Form FIR Structure for \( M \) Even

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

- 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 \( k = \text{tf2latc}(\text{num}) \))
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 $-\infty < n < \infty$
where the ideal frequency responses corresponding to the above impulse responses are

\[
\begin{align*}
    h_{d, \text{LP}}[n] &= \frac{\sin(\omega_c n)}{\pi n} \\
    h_{d, \text{HP}}[n] &= \delta(n) - \frac{\sin(\omega_c n)}{\pi n} \\
    h_{d, \text{BP}}[n] &= \frac{\sin(\omega_b n) - \sin(\omega_a n)}{\pi n} \\
    h_{d, \text{BS}}[n] &= \delta(n) - \frac{\sin(\omega_b n) - \sin(\omega_a n)}{\pi n}
\end{align*}
\]

Ideal Brickwall Filters

- Consider obtaining a causal FIR filter that approximates \( h_d[n] \) by letting

\[
h[n] = \begin{cases} 
    h_d[n - M/2], & 0 \leq n \leq M \\
    0, & \text{otherwise}
\end{cases}
\]
• 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

  $$h[n] = h_d[n - M/2]w[n]$$

  (7.9)

  where here $w[n]$ is unity for $0 \leq n \leq M$ 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

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

  (7.10)

\[\begin{array}{c}
\text{(a) Circular convolution (windowing)}
\end{array}\]
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
Consider an $M = 32$ point design with $\omega_c = \pi/2$

The basic response $h_d[n]$ for a lowpass is of the form

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

(7.12)

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:

```matlab
>> 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

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

(7.13)

where $A_e(e^{j\omega})$ is a real and even function of $\omega$
Chapter 7 • Real-Time FIR Digital Filters

M = 32

Sample Index $n$

$h_d[n]$  

$w[n]$  

$w[n]$  

$H = 32$  

Hanning Window
Overview of FIR Filter Design

M = 32

\[ |H_d(e^{j\omega})|_{\text{dB}} \]

Down only 21 dB

\[ |W(e^{j\omega})|_{\text{dB}} \]

Hanning Window

\[ |H(e^{j\omega})|_{\text{dB}} \]
• For an odd symmetric design

\[ H(e^{j\omega}) = jA_e(e^{j\omega})e^{-j\omega M/2} \]  

(7.14)

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

\[ A_e(e^{j\omega}) = \int_{-\infty}^{\infty} H_e(e^{j\theta})W_e(e^{j(\omega - \theta)})d\theta \]  

(7.15)

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

\[
h[n] = \frac{\sin(\omega_c(n - M/2))}{\pi(n - M/2)} w[n]
\]  
(7.16)

- The lowpass amplitude specifications of interest are

\[
|H_d(e^{j\omega})| = 1 \pm \delta
\]

Lowpass Amplitude Response Specifications

- 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 (7.19))

---

\[
\begin{align*}
\text{hn} & = \frac{\sin(\omega_c(n - M/2))}{\pi(n - M/2)} \approx \frac{\sin(\omega_c(n - M/2))}{\pi(n - M/2)} \\
\text{wn} & = \frac{\omega_c(n - M/2)}{\pi(n - M/2)}
\end{align*}
\]
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
   \[ \Delta \omega \leq \omega_s - \omega_p \] (7.17)
3. Choose $\omega_c$ in the truncated impulse response such that
   \[ \omega_c = \frac{\omega_p + \omega_s}{2} \] (7.18)
4. Plot $|H(e^{j\omega})|$ to see if the specifications are satisfied
5. Adjust $\omega_c$ and $M$ if necessary to meet the requirements; If possible reduce $M$

Table 7.1: Window characteristics for FIR design.

<table>
<thead>
<tr>
<th>Type</th>
<th>Transition Bandwidth $\Delta \omega$</th>
<th>Minimum Stopband Attenuation</th>
<th>Equivalent Kaiser $\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle</td>
<td>$1.81\pi/M$</td>
<td>21 dB</td>
<td>0</td>
</tr>
<tr>
<td>Bartlett</td>
<td>$1.80\pi/M$</td>
<td>25 dB</td>
<td>1.33</td>
</tr>
<tr>
<td>Hanning</td>
<td>$5.01\pi/M$</td>
<td>44 dB</td>
<td>3.86</td>
</tr>
<tr>
<td>Hamming</td>
<td>$6.27\pi/M$</td>
<td>53 dB</td>
<td>4.86</td>
</tr>
<tr>
<td>Blackman</td>
<td>$9.19\pi/M$</td>
<td>74 dB</td>
<td>7.04</td>
</tr>
</tbody>
</table>
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.,

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

   (7.19)

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

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

   (7.20)

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

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

   (7.21)

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 \( H(e^{j\omega}) \)

\[
E(\omega) = W(\omega)[H(e^{j\omega}) - H_d(e^{j\omega})]
\]  
(7.22)

• 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

---

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

\[
|H_d(e^{j\omega})| = \begin{cases} 
1 + \delta_1 & \text{for } \omega < \omega_p \\
1 - \delta_1 & \text{for } \omega_p < \omega < \omega_c \\
0 & \text{for } \omega_c < \omega < \omega_s \\
\delta_2 & \text{for } \omega_s < \omega < \pi 
\end{cases}
\]

Lowpass Amplitude Response Specifications

• 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y = filter(b,a,x)</code></td>
<td>Direct form II filter vector x</td>
</tr>
<tr>
<td><code>[H,w] = freqz(b,a)</code></td>
<td>Z-domain frequency response computation</td>
</tr>
<tr>
<td><code>[Gpd,w] = grpdelay(b,a)</code></td>
<td>Group delay computation</td>
</tr>
<tr>
<td><code>h = impz(b,a)</code></td>
<td>Impulse response computation</td>
</tr>
<tr>
<td><code>unwrap</code></td>
<td>Phase unwrapping</td>
</tr>
<tr>
<td><code>zplane(b,a)</code></td>
<td>Plotting of the z-plane pole/zero map</td>
</tr>
</tbody>
</table>

### Linear System Transformations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>residuez</code></td>
<td>Z-domain partial fraction conversion</td>
</tr>
<tr>
<td><code>tf2zp</code></td>
<td>Transfer function to zero-pole conversion</td>
</tr>
<tr>
<td><code>zp2sos</code></td>
<td>Zero-pole to second-order biquadratic sections conv.</td>
</tr>
<tr>
<td><code>tf2latc</code></td>
<td>Transfer function to lattice conversion</td>
</tr>
</tbody>
</table>

### FIR Filter Design

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fir1</code></td>
<td>Window-based FIR filter design with standard response</td>
</tr>
<tr>
<td><code>kaiserord</code></td>
<td>Kaiser window FIR filter design estimation parameters with <code>fir1</code></td>
</tr>
</tbody>
</table>
## Windowed FIR Design From Amplitude Specifications

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

- 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

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

<table>
<thead>
<tr>
<th>MATLAB Basic Filter Design Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIR Filter Design (Continued)</strong></td>
<td><strong>fir2</strong></td>
</tr>
<tr>
<td><strong>firpm</strong></td>
<td>Parks-McClellan optimal FIR filter design; optimal (equiripple) fit between the desired and actual frequency responses</td>
</tr>
<tr>
<td><strong>firpmord</strong></td>
<td>Parks-McClellan optimal FIR filter order estimation</td>
</tr>
</tbody>
</table>
Verification of these designs can be accomplished using the MATLAB filter design function `fir1()`

```matlab
% Hamming window design
% (note fir1 uses hamming by default)
b = fir1(32, 2*0.25, 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 0.5 -70 2])
grid
% Kasier window design
bk = fir1(30, 2*0.25, kaiser(30+1, 4.5335));
[Hk, F] = freqz(bk, 1, 512, 1);
plot(F, 20*log10(abs(Hk)))
axis([0 0.5 -70 2])
grid
```

![Impulse response with Hamming window](image)

**$M = 32$, $\omega_c = 0.25 \times 2\pi$**
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.

\[ M = 32 \text{ pole-zero plot with Hamming window} \]

\[ M = 32 \text{ Hamming Window FIR} \]

\[ \omega_c = 0.25 \times 2\pi \]

\[ \omega_s = 0.6\pi \]

\[ \text{Should be } -50 \text{ dB} \]

\[ M = 32 \text{ frequency response with Hamming window} \]
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 $f_p = 400$ Hz and $f_s = 600$ Hz.

Under Set Quantization Parameters, we set coefficient quantize to a Q15 format.
• Many other filter viewing options are available as well
  – Display phase, Gain & phase, Group delay
  – Impulse response, Step response
  – Pole-zero plot
  – Filter coefficients
• Once you are satisfied with a design, the filter coefficients can be exported to a .h file, or to the MATLAB workspace
/ * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 *
 * Generated by MATLAB(R) 7.0.1 and the Filter Design Toolbox 3.1.
 *
 * Generated on: 15-Mar-2005 14:59:26
 *
 */

/* Discrete-Time FIR Filter (real)
 * -------------------------------
 * Filter Structure : Direct-Form FIR
 * Filter Order : 30
 * Stable : Yes
 * Linear Phase : Yes (Type 1)
 * Arithmetic : fixed
 * Numerator : S16Q15
 * Input : S16Q15
 * Output : S16Q9
 * Product : S32Q30
 * Accumulator : S40Q30
 * Round Mode : convergent
 * Overflow Mode : wrap
 * Cast Before Sum : true
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"

/* Expected path to tmwtypes.h
 * C:\MATLAB701\extern\include\tmwtypes.h
 */
const int N_FIR = 31;
const int16_T h[31] = {
    -39, 0, 123, 0, -275, 0, 529, 0, 
    -943, 0, 1662, 0, -3208, 0, 10338, 16384, 
    10338, 0, -3208, 0, 1662, 0, -943, 0, 
    529, 0, -275, 0, 123, 0, -39
};
Example: Equiripple FIR Design Using float

- As a second example of using FDATool consider an equiripple design using a sampling frequency of 8 kHz to match a AIC3106, \( f_p = 1600 \text{Hz} \), \( f_s = 2000 \text{Hz} \), a passband ripple of \( \pm 0.5 \text{dB} \) (1 dB total), and a stopband attenuation of 60 dB.

- Since the lab is no longer equipped with the fixed-point toolbox, we will export the coefficients using the function

```matlab
function filter_C_header(Hq,mode,filename,N)
% filter_C_header(Hq,mode,filename,N): Used to create a C-style header file containing filter coefficients. This reads Hq filter objects assuming a direct-form FIR design is present
```
%        Hq = quantized filter object containing desired coefficients
%      mode = specify 'fixed', 'float', or 'exponent'
% file_name = string name of file to be created
%         N = number of coefficients per line

Writing float C Coefficient Files

An alternative to using FDA tool to write header files, we can use the function

function filter_C_header(Hq,mode,filename,N)
%        filter_C_header(Hq,mode,filename,N): Used to create a C-style header file
%        containing filter coefficients. This reads Hq filter objects assuming a
%        direct-form FIR design is present
%
%        Hq = quantized filter object containing desired coefficients
%      mode = specify 'fixed', 'float', or 'exponent'
% file_name = string name of file to be created
%         N = number of coefficients per line

>> filter_C_header(Hd,'float','fir_fltcoeff.h',3)
The result is

```c
#define N_FIR 40

/* Filter Coefficients */
float h[N_FIR] = {-0.004314029307, -0.013091321622, -0.016515087727,
                 -0.006430584433, 0.009817876267, 0.010801880238,
                 -0.006567413713, -0.016804829623, 0.000653253913,
                 0.022471280087, 0.010147131468, -0.02567740989,  
                 -0.026558960619, 0.023048392854, 0.050385290390,  
                 -0.009291203588, -0.087918503442, -0.033770330014,  
                 0.18734796517, 0.401505729848, 0.401505729848,  
                 0.18734796517, -0.033770330014, -0.087918503442,  
                 -0.009291203588, 0.050385290390, 0.023048392854,  
                 -0.026558960619, -0.02567740989, 0.010147131468,  
                 0.022471280087, 0.000653253913, -0.016804829623,  
                 -0.006567413713, 0.010801880238, 0.009817876267,  
                 -0.006430584433, -0.016515087727, -0.013091321622,  
                 -0.004314029307};
```

Real-time code based on `ISRs_Plus.c` is the following

```c
#include "DSP_Config.h"
#include "fir_fltcoeff.h"    //coefficients in float format

#include "DSP_Config.h"
#include "fir_fltcoeff.h"    //coefficients in float format

// Function Prototypes
long int rand_int(void);

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word. The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.
#define LEFT  0
```
Using MATLAB’s fdatool

#define RIGHT 1

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

/* add any global variables here */
float x_buffer[N_FIR]; //buffer for delay samples

interrupt void Codec_ISR()

///////////////////////////////////////////////////////////////////////
// Purpose: Codec interface interrupt service routine
//
// Input: None
//
// Returns: Nothing
//
// Calls: CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes: None
///////////////////////////////////////////////////////////////////////
{
    /* add any local variables here */
    WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse
    int i;
    float result = 0; //initialze the accumulator

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

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

    //Work with Left ADC sample
    //x_buffer[0] = 0.25 * CodecDataIn.Channel[ LEFT];
    //Use the next line to noise test the filter
    x_buffer[0] = 0.125*((short) rand_int())); //scale input by 1/8

    //Filtering using a 32-bit accumulator
    for(i=0; i< N_FIR; i++)
    {
        result += x_buffer[i] * h[i];
    }
    //Update filter history
for(i=N_FIR-1; i>0; i--)
{
    x_buffer[i] = x_buffer[i-1];
}

//Return 16-bit sample to DAC
CodecDataOut.Channel[ LEFT] = (short) result;
// Copy Right input directly to Right output with no filtering
CodecDataOut.Channel[RIGHT] = CodecDataIn.Channel[ RIGHT];
/* end your code here */
WriteCodecData(CodecDataOut.UINT);// send output data to port
WriteDigitalOutputs(0); // Write to GPIO J15, pin 6; end ISR timing pulse

//White noise generator for filter noise testing
long int rand_int(void)
{
    static long int a = 100001;
    a = (a*125) % 2796203;
    return a;
}

• In this C routine the filter is implemented using a buffer to hold present and past values of the input in normal order
  float x_buffer[N_FIR];

• In particular x_buffer[0] holds the present input,
  x_buffer[1] holds the past input and so on

• The filter coefficients are held in the short array h[] as \(a_0 = h[0]\ldots a_{39} = h[39]\)

• The filter output is computed by placing
  \(\text{result += x_buffer[i] * h[i]}\);
in a loop running from \(i=0\) to \(i=N\_FIR-1\), here \(N\_FIR = 40\)
• Once the program has been loaded and run, we can verify that the float coefficients contained in `fir_flt coeff.h` are loaded into memory properly using the graph capability of CCS.

![Graph of power spectrum](image1)

A 30 second sound file created at 44.1 ksp, 16-bits, was imported into MATLAB and then the power spectrum was estimated using `psd()`

```matlab
>> [Px,F] = simpleSA(x(:,1),2^11,48000,-90,10,'b');
>> plot(F,10*log10(Px/max(Px)))
```
ISR Timing Results

- We see that the ISR running the 40 tap FIR requires a total of 9 μs
- In the limit this says when sampling at 8 kspgs a maximum of $125 \times 40 / 9 \approx 555$ taps can be implemented
Code Enhancements

- The use of C intrinsics would allow us to utilize parallel multiply accumulate operations
  - This idea applies to word (32-bit) operations for the fixed filter and double word operations on the C67 for the float case
- We have not yet used a circular buffer, which is part of the C6x special hardware capabilities

A Simple Fixed-Point Implementation

- Simple modification to the previous float FIR example yields a version using short integers

```c
#include "DSP_Config.h"
#include "fir_fixcoeff.h" // coefficients in decimal format

#define LEFT 0
#define RIGHT 1
```

// Data is received as 2 16-bit words (left/right) packed into one 32-bit word. The union allows the data to be accessed as a single entity when transferring to and from the serial port, but still be able to manipulate the left and right channels independently.
volatile union {
    Uint32 UINT;
    Int16 Channel[2];
} CodecDataIn, CodecDataOut;

/* add any global variables here */
short x_buffer[N_FIR]; // buffer for delay samples

interrupt void Codec_ISR()

///////////////////////////////////////////////////////////////////////
// Purpose: Codec interface interrupt service routine
// Input: None
// Returns: Nothing
// Calls: CheckForOverrun, ReadCodecData, WriteCodecData
// Notes: None
///////////////////////////////////////////////////////////////////////
{
    /* add any local variables here */
    WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse
int i;
    int result = 0; // initialize the accumulator

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

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

    /* add your code starting here */
    // Work with Left ADC sample
    // x_buffer[0] = 0.25 * CodecDataIn.Channel[ LEFT];
    // Use the next line to noise test the filter
    x_buffer[0] = ((short) rand_int())>>3; // scale input by 1/8

    // Filtering using a 32-bit accumulator
    for(i=0; i< N_FIR; i++)
    {
        result += x_buffer[i] * h[i];
    }
    // Update filter history
    for(=N_FIR-1; i>=0; i--)
}
Using MATLAB’s \texttt{fdatool}

\begin{verbatim}
{
    x_buffer[i] = x_buffer[i-1];
}

//Return 16-bit sample to DAC; recall result is a 32 bit accumulator
CodecDataOut.Channel[ LEFT] = (\texttt{short}) (result >> 15);
// Copy Right input directly to Right output with no filtering
CodecDataOut.Channel[RIGHT] = CodecDataIn.Channel[ RIGHT];
/* end your code here */

WriteCodecData(CodecDataOut.UINT); // send output data to port
WriteDigitalOutputs(0); // Write to GPIO J15, pin 6; end ISR timing pulse

//White noise generator for filter noise testing
long int \texttt{rand_int}(\texttt{void})
{
    static long int a = 100001;

    a = (a*125) \% 2796203;
    return a;
}

\end{verbatim}

**Writing fixed C Coefficient Files**

\texttt{>> filter\_C\_header(Hd,'fixed','fir\_fixcoeff.h',8)}

//define the FIR filter length
\texttt{#define N\_FIR 40}

/*****************************/
\texttt{/* Filter Coefficients */}
\texttt{short h[N\_FIR] = \{ -141, -429, -541, -211, 322, 354, -215, -551,}
\texttt{21, 736, 333, -841, -870, 755, 1651, -304,}
\texttt{-2881,-1107, 6139,13157,13157, 6139,-1107,-2881,}
\texttt{-304, 1651, 755, -870, -841, 333, 736, 21,}
\texttt{-551, -215, 354, 322, -211, -541, -429, -141 \};}

/*****************************/

- The filter was tested with the OMAP-L138 board’s AIC3106 and a white noise input generated in software
  \texttt{x\_buffer[0] = ((\texttt{short}) rand\_int())\gg 3; //scale input by 1/8}
- A 30 second sound file created at 44.1 ksp, 16-bits, was imported into MATLAB and then the power spectrum was
estimated using \texttt{psd}()

\begin{verbatim}
>> [Px,F] = simpleSA(x(:,1),2^11,48000,-90,10,'b');
>> plot(F,10*log10(Px/max(Px)))
\end{verbatim}

\textbf{ISR Timing Results}

- We see that the ISR running the 40 tap FIR requires a total of 8.25 \mu s, which is slightly faster than the \texttt{float} version
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.

- See text Section 3.4.3

![Circular Addressing Diagram](attachment:image.png)
• See also TI Application Report SPRA645–June 2002

**C Circular Buffer**

• A circular buffer can be directly implemented in C
• The code given below is a modification of the program ISRs_fir_short_circ.c:

```c
/* add any global variables here */
short x_buffer[N_FIR]; //buffer for delay samples
short newest, x_index; //circular buffer index control

interrupt void Codec_ISR()

///////////////////////////////////////////////////////////////////////
// Purpose: Codec interface interrupt service routine
//
// Input: None
//
// Returns: Nothing
//
// Calls: CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes: None
///////////////////////////////////////////////////////////////////////
{
    /* add any local variables here */
    WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse
    int i;
    int result = 0; //initialize the accumulator

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

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

    /* add your code starting here */
    //circularly increment newest
    ++newest;
    if(newest == N_FIR) newest = 0;
    //Put new sample in circular buffer from ADC
    //Work with Left ADC sample
    //x_buffer[newest] = 0.25 * CodecDataIn.Channel[ LEFT];
    //Use the next line to noise test the filter
```

---

7–40  
ECE 5655/4655 Real-Time DSP
Circular Addressing

```c
x_buffer[newest] = ((short) rand_int()) >> 3; // scale input by 1/8

// Filtering using a 32-bit accumulator
x_index = newest;
for (i = 0; i < N_FIR; i++)
{
    result += h[i] * x_buffer[x_index];
    // circularly decrement x_index
    --x_index;
    if (x_index == -1) x_index = N_FIR - 1;
}

// Return 16-bit sample to DAC; recall result is a 32-bit accumulator
CodecDataOut.Channel[LEFT] = (short) (result >> 15);
// Copy Right input directly to Right output with no filtering
CodecDataOut.Channel[RIGHT] = CodecDataIn.Channel[RIGHT];
/* end your code here */

WriteCodecData(CodecDataOut.UINT); // send output data to port
WriteDigitalOutputs(0); // Write to GPIO J15, pin 6; end ISR timing pulse
```

- Notice that C’s mod function `x % y` is not used here since it is less efficient than the if statement which does the circular wrapping
- A timing comparison is done using a logic analyzer:

![Logic Analyzer Screenshot]

- We see the interrupt time is now 7.5 μs, which is faster by 0.75 μs, compared with the earlier fixed-point FIR
Circular Buffering via C Calling Assembly

- The C6x has underlying hardware to handle circular buffer setup and manipulation
- In the Chassaing text the programs `FIRcirc.c`, and when using external memory for the filter coefficients, `FIRcirc_ext.c`, take advantage of the C6x circular addressing mode
  - Details on the use of hardware circular addressing can be found in Chassaing Example 4.14 and Appendix B
- Two circular buffers can be defined using `BK0` and `BK1` in the address mode register (`AMR`)

<table>
<thead>
<tr>
<th>Mode Select Fields</th>
<th>Block Size Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>B7 Mode</td>
<td>B6 Mode</td>
</tr>
<tr>
<td>B5 Mode</td>
<td>B4 Mode</td>
</tr>
<tr>
<td>A7 Mode</td>
<td>A6 Mode</td>
</tr>
<tr>
<td>A5 Mode</td>
<td>A4 Mode</td>
</tr>
<tr>
<td>Mode Select Fields</td>
<td>Block Size Fields</td>
</tr>
<tr>
<td>32</td>
<td>26 25</td>
</tr>
<tr>
<td>21 20</td>
<td>16</td>
</tr>
<tr>
<td>Reserved</td>
<td>BK1</td>
</tr>
<tr>
<td>BK0</td>
<td></td>
</tr>
</tbody>
</table>

- The size of the circular buffers, in bytes, is set into the block size fields as $2^{N+1}$, e.g., for 128 bytes (64 short values), $N = 6$ or for `BK0` bits 16–20 would be 00110b
- The `AMR` is also used to configure the eight register pointers `A4–A7` and `B4–B7`
• The two mode select bits are set as shown in the table below

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Linear addressing, the default on reset</td>
</tr>
<tr>
<td>01</td>
<td>Circular addressing using BK0</td>
</tr>
<tr>
<td>10</td>
<td>Circular addressing using BK1</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 7.2: AMR mode register settings.

• When returning from an assembly routine that uses circular addressing, the AMR setting must be returned to the linear value

• We now consider fir_asmcirc.c and the associated assembly file FIRcircfunc.asm

```c
//A circular buffer FIR real-time filtering program
//A modified version of Fir_pcm.c using the AIC23

//project fir_asmcirc.pjt
//fir_short_asmcirc_AIC23.c FIR

int fircircfunc(short x, short *h, int N);

//#include "fir_hcoeff.h"  //N_FIR=40 coeff. in hex
// Assembly assumes 128 taps or some power of 2
#include "bp1750.cof"       //N=128 BP coeff. in decimal
                //f0 = 1750 at fs = 8k

int rand_int(void);
```
#include "DSK6713_AIC23.h"
//set sampling rate \{8,16,24,32,44.1,48,96\}
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;
#define DSK6713_AIC23_INPUT_MIC 0x0015
#define DSK6713_AIC23_INPUT_LINE 0x0011
//select input
Uint16 inputsource=DSK6713_AIC23_INPUT_LINE;

int result = 0;       //initial filter output

interrupt void c_int11() //ISR
{
    short sample_data;

    //Get sample from ADC
    sample_data = input_left_sample();
    //Use the next line to noise test the filter
    //sample_data = ((short) rand_int())>>1;

    //Filtering using a 32-bit accumulator
    result = fircircfunc(sample_data, h, N);

    //Return 16-bit sample to DAC
    output_left_sample((short) (result >> 15));

    return;       //return from ISR
}

void main()
{
    comm_intr();       //init DSK, codec, McBSP
    while(1);          //infinite loop
}
...
Circular Addressing

;FIRcircfunc.asm ASM function called from C using
; circular addressing. A4=newest sample, 
; B4=coefficient address, A6=filter order 
; Delay samples organized: x[n-(N-1)]...x[n] 
; Coeff as h(0)...h[N-1]

.def _fircircfunc
.def last_addr
.def delays
.sect "circdata" ;circular data section
.align 256 ;align delay buffer 
;256-byte boundary
delays .space 256 ;init 256-byte 
;buffer with 0's
last_addr .int last_addr-1 ;point to bottom of 
;delays buffer 
.text ;code section
_fircircfunc: ;FIR function using circ addr
MV A6,A1 ;setup loop count
MPY A6,2,A6 ;since dly buffer data as
;byte
ZERO A8 ;init A8 for accumulation
ADD A6,B4,B4 ;since coeff buffer
;data as bytes
SUB B4,1,B4 ;B4=bottom coeff array h[N-1]

MVKL 0x00070040,B6 ;select A7 as pointer 
;and BK0
MVKH 0x00070040,B6 ;BK0 for 256 bytes 
;(128 shorts)
MVC B6,AMR ;set address mode reg. AMR
MVK last_addr,A9 ;A9=last circ addr 
;(lower 16 bits)
MVKH last_addr,A9 ;last circ addr
Chapter 7 • Real-Time FIR Digital Filters

; (higher 16 bits)
LDW *A9,A7  ;A7=last circ addr
NOP 4
STH A4,*A7++  ;newest sample-->last
            ;address
loop:    ;begin FIR loop
    LDH *A7++,A2  ;A2=x[n-(N-1)+i]
                ;i=0,1,...,N-1
    ||  LDH *B4-- ,B2  ;B2=h[N-1-i]
                ;i=0,1,...,N-1
    SUB A1,1,A1  ;decrement count

[A1]  B  loop  ;branch to loop if count # 0
    NOP 2
    MPY A2,B2,A6  ;A6=x[n-(N-1)+i]*h[N-1+i]
    NOP
    ADD A6,A8,A8  ;accumulate in A8

    STW A7,*A9  ;store last circ addr to
            ;last_addr
    B  B3  ;return addr to calling routine
    MV A8,A4  ;result returned in A4
    NOP 4

- Looking at the assembly module we see that a block of 256
  bytes or 128 short integers is reserved for the circular buffer
  – As currently configured the circular buffer must be a
    power of 2, so the filter coefficient set used in a previous
    example will not work, since here N_FIR = 40
• Once the circular buffer is configured via the AMR settings, all we need to maintain from sample-to-sample is the location of the current newest signal sample; here we use memory location label \texttt{last_addr}.

\textbf{On First Pass Through Algorithm}

\begin{align*}
\text{delays}[n] & & h[n] \\
0 & & 0 \\
1 & & \\
\vdots & & \\
N-1 & & N-2 \\
\text{last_addr} & & N-1 \\
\text{last_addr-1} & & \\
\text{Holds most recent last address into the circular buffer} & & \\
\text{Oldest Sample} & & \\
\text{Newest Sample} & & \\
\text{Multiply and add} & & \\
\end{align*}
The process continues with newest sample location moving around the buffer in a circular fashion.
Performance of Interrupt Driven Programs

• In the development of real-time signal processing systems, there are clearly performance trade-offs to be made

• The character of all the interrupt driven programs we have seen thus far is that:
  – We first load system parameters into memory, e.g., filter coefficients
  – Next registers are initialized on the CPU and serial port is configured
  – The appropriate interrupt is enabled on the CPU
  – The program then enters an infinite (wait) loop
  – While in this infinite loop the CPU is interrupted at the sampling rate of $f_s$ samples per second by the AIC23, which has new analog I/O samples to exchange
  – Before returning to the wait loop again, the program executes code that carries out the desired signal processing tasks (Note: here we assume for simplicity that there are no other background signal processing tasks to be performed)

• In order for the DSP application to maintain true real-time status, all signal processing computations must be completed before the next codec initiated interrupt is fired

• Debugging interrupt driven programs is not a simple matter
• Pictorially we are doing the following:

The Wait Loop Counter Scheme
C5515 Code Example

- The C5515 is a fixed-point processor, so in this FIR filter example we port the earlier OMAP-L138 fixed-point example
- The data types on the VC5505/C5515 are slightly different, so first we consider the differences

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Representation</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, signed char</td>
<td>16 bits</td>
<td>ASCII</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>unsigned char</td>
<td>16 bits</td>
<td>ASCII</td>
<td>0</td>
<td>65 535</td>
</tr>
<tr>
<td>short, signed short</td>
<td>16 bits</td>
<td>2s complement</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16 bits</td>
<td>Binary</td>
<td>0</td>
<td>65 535</td>
</tr>
<tr>
<td>int, signed int</td>
<td>16 bits</td>
<td>2s complement</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16 bits</td>
<td>Binary</td>
<td>0</td>
<td>65 535</td>
</tr>
<tr>
<td>long, signed long</td>
<td>32 bits</td>
<td>2s complement</td>
<td>-2 147 483 648</td>
<td>2 147 483 647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32 bits</td>
<td>Binary</td>
<td>0</td>
<td>4 294 967 295</td>
</tr>
<tr>
<td>long long</td>
<td>40 bits</td>
<td>2s complement</td>
<td>-549 755 813 888</td>
<td>549 755 813 887</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>40 bits</td>
<td>Binary</td>
<td>0</td>
<td>1 099 511 627 775</td>
</tr>
<tr>
<td>enum</td>
<td>16 bits</td>
<td>2s complement</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>IEEE 32-bit</td>
<td>1.175 494e−38</td>
<td>3.40 282 346e+38</td>
</tr>
<tr>
<td>double</td>
<td>32 bits</td>
<td>IEEE 32-bit</td>
<td>1.175 494e−38</td>
<td>3.40 282 346e+38</td>
</tr>
<tr>
<td>long double</td>
<td>32 bits</td>
<td>IEEE 32-bit</td>
<td>1.175 494e−38</td>
<td>3.40 282 346e+38</td>
</tr>
<tr>
<td>pointers (data)</td>
<td>16 bits</td>
<td>Binary</td>
<td>0</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>small memory mode</td>
<td>23 bits</td>
<td>Binary</td>
<td>0</td>
<td>0x7FFFF</td>
</tr>
<tr>
<td>large memory mode</td>
<td>24 bits</td>
<td>Binary</td>
<td>0</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>

- Note in particular that int is only 16 bits and long is 32 bits
- There are 40-bit registers available on the processor
- There are also two MAC units that support $17 \times 17$ bit
Design the filter using `fdatool`

In a polling-based AIC3204 program, we insert a modified version of the OMAP-L138 fixed-point routine presented earlier

```c
//
// main.c in C5515_AIC_3204_polling_FIR
// Mark Wickert March 2012
//
// Codec Library Functions
#include "aic3204.h"
#include "fir_fixcoeff.h"  // coefficients in decimal format

// Function Prototypes
long int rand_int(void);
```
short x_buffer[N_FIR]; //buffer for delay samples

/* Sampling Rates:
 * AIC3204_FS_8KHZ
 * AIC3204_FS_16KHZ
 * AIC3204_FS_24KHZ
 * AIC3204_FS_32KHZ
 * AIC3204_FS_48KHZ
 * AIC3204_FS_96KHZ
 */
short fs = AIC3204_FS_8KHZ;

void main(void)
{
    //Define working variables
    short i = 0;
    short l_chan, r_chan;
    short buffer[128];

    //Define filter variables
    long result; //32 bit accumulator

    // Initialize Polling
    comm_poll();

    while(1)
    {
        // Get sample using inputs
        input_sample(&l_chan, &r_chan);

        // Get random sample
        r_chan = ((short) rand_int())>>3;

        /* Fill buffer */
        buffer[i] = r_chan;
        i += 1;
        if (i == 128) i = 0;

        // FIR Filter
        x_buffer[0] = r_chan;
        result = 0;
        for(i=0; i< N_FIR; i++)
        {
            result += (long) x_buffer[i] * h[i];
        }
        //Update filter history
        for(i=N_FIR-1; i>0; i--)
        {
Chapter 7 • Real-Time FIR Digital Filters

{ 
    x_buffer[i] = x_buffer[i-1]; 
}

/* Write Digital audio input */
output_sample(l_chan, (short)(result>>16));
//output_sample(l_chan, r_chan);
}

// White noise generator for filter noise testing
long int rand_int(void)
{
    static long int a = 100001;
    a = (a*125) % 2796203;
    return a;
}

// fir_fixcoeff.h
// define the FIR filter length
#define N_FIR 40
/**************************************************************************/
/* Filter Coefficients */
short h[N_FIR] = { -141, -429, -541, -211, 322, 354, -215, -551, 
    '21, 736, 333, -841, -870, 755, 1651, -304, 
    -2881,-1107, 6139,13157,13157, 6139,-1107,-2881, 
    -304, 1651, 755, -870, -841, 333, 736, 21, 
    -551, -215, 354, 322, -211, -541, -429, -141 }; 
/**************************************************************************/

• Audio noise capture of a 2 kHz lowpass filter, assuming a sampling rate of 48 kHz

>> [Px,F] = simpleSA(x(:,2),2^11,48000,-90,10,'b');
>> plot(F,10*log10(Px/max(Px)))
>> axis([0 8000 -80 0])
The results look similar to the earlier OMAP-L138 AIC3106 when using the same fixed-point coefficient set.