DSP Applications Using C and the TMS320C6x DSK. Rulph Chassaing

ISBNs: 0-471-20754-3 (Hardback); 0-471-22112-0 (Electronic)

4

Finite Impulse Response Filters

- Introduction to the z-transform
- Design and implementation of finite impulse response (FIR) filters
- Programming examples using C and TMS320C6x code

The z-transform is introduced in conjunction with discrete-time signals. Mapping from the s-plane, associated with the Laplace transform, to the z-plane, associated with the z-transform, is illustrated. FIR filters are designed with the Fourier series method and implemented by programming a discrete convolution equation. Effects of window functions on the characteristics of FIR filters are covered.

4.1 INTRODUCTION TO THE *Z*-TRANSFORM

The z-transform is utilized for the analysis of discrete-time signals, similar to the Laplace transform for continuous-time signals. We can use the Laplace transform to solve a differential equation that represents an analog filter, or the z-transform to solve a difference equation that represents a digital filter. Consider an analog signal $x(t)$ ideally sampled

$$x_s(t) = \sum_{k=0}^{\infty} x(t)\delta(t - kT) \tag{4.1}$$

where $\delta(t - kT)$ is the impulse (delta) function delayed by kT and $T = 1/F_s$ is the sampling period. The function $x_s(t)$ is zero everywhere except at $t = kT$. The Laplace transform of $x_s(t)$ is

$$X_s(s) = \int_0^\infty x_s(t)e^{-st}dt$$
$$= \int_0^\infty \{x(t)\delta(t) + x(t)\delta(t-T) + \cdots\}e^{-st}dt \tag{4.2}$$

From the property of the impulse function

$$\int_0^\infty f(t)\delta(t-kT)dt = f(kT)$$

$X_s(s)$ in (4.2) becomes

$$X_s(s) = x(0) + x(T)e^{-sT} + x(2T)e^{-2sT} + \cdots = \sum_{n=0}^{\infty} x(nT)e^{-nsT} \tag{4.3}$$

Let $z = e^{sT}$ in (4.3), which becomes

$$X(z) = \sum_{n=0}^{\infty} x(nT)z^{-n} \tag{4.4}$$

Let the sampling period T be implied; then $x(nT)$ can be written as $x(n)$, and (4.4) becomes

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} = ZT\{x(n)\} \tag{4.5}$$

which represents the z-transform (ZT) of $x(n)$. There is a one-to-one correspondence between $x(n)$ and $X(z)$, making the z-transform a unique transformation.

Exercise 4.1: ZT of Exponential Function x(n) = e^{nk}

The ZT of $x(n) = e^{nk}$, $n \geqslant 0$ and k a constant, is

$$X(z) = \sum_{n=0}^{\infty} e^{nk}z^{-n} = \sum_{n=0}^{\infty} (e^k z^{-1})^n \tag{4.6}$$

Using the geometric series, obtained from a Taylor series approximation

$$\sum_{n=0}^{\infty} u^n = \frac{1}{1-u} \qquad |u| < 1$$

(4.6) becomes

$$X(z) = \frac{1}{1 - e^k z^{-1}} = \frac{z}{z - e^k} \tag{4.7}$$

for $|e^k z^{-1}| < 1$ or $|z| > |e^k|$. If $k = 0$, the ZT of $x(n) = 1$ is $X(z) = z/(z-1)$.

Exercise 4.2: ZT of Sinusoid x(n) = sin nωT

A sinusoidal function can be written in terms of complex exponentials. From Euler's formula $e^{ju} = \cos u + j\sin u$,

$$\sin n\omega T = \frac{e^{jn\omega T} - e^{-jn\omega T}}{2j}$$

Then

$$X(z) = \frac{1}{2j}\sum_{n=0}^{\infty}\left(e^{jn\omega T} z^{-n} - e^{-jn\omega T} z^{-n}\right) \tag{4.8}$$

Using the geometric series as in Exercise 4.1, one can solve for $X(z)$; or the results in (4.7) can be used with $k = j\omega T$ in the first summation of (4.8) and $k = -j\omega T$ in the second, to yield

$$\begin{aligned} X(z) &= \frac{1}{2j}\left(\frac{z}{z - e^{j\omega T}} - \frac{z}{z - e^{-j\omega T}}\right) \\ &= \frac{1}{2j}\frac{z^2 - ze^{-j\omega T} - z^2 + ze^{j\omega T}}{z^2 - z(e^{-j\omega T} + e^{j\omega T}) + 1} \\ &= \frac{z\sin\omega T}{z^2 - 2z\cos\omega T + 1} \end{aligned} \tag{4.9}$$

$$= \frac{Cz}{z^2 - Az - B} \qquad |z| > 1 \tag{4.10}$$

where $A = 2\cos\omega T$, $B = -1$, and $C = \sin\omega T$. In Chapter 5 we generate a sinusoid based on this result. We can readily generate sinusoidal waveforms of different frequencies by changing the value of ω in (4.9).

Similarly, using Euler's formula for $\cos n\omega T$ as a sum of two complex exponentials, one can find the ZT of $x(n) = \cos n\omega T = (e^{jn\omega T} + e^{-jn\omega T})/2$, as

$$X(z) = \frac{z^2 - z\cos\omega T}{z^2 - 2z\cos\omega T + 1} \qquad |z| > 1 \tag{4.11}$$

4.1.1 Mapping from s-Plane to z-Plane

The Laplace transform can be used to determine the stability of a system. If the poles of a system are on the left side of the $j\omega$ axis on the s-plane, a time-decaying system response will result, yielding a stable system. If the poles are on the right side of the $j\omega$ axis, the response will grow in time, making such a system unstable. Poles located on the $j\omega$ axis, or purely imaginary poles, will yield a sinusoidal response. The sinusoidal frequency is represented by the $j\omega$ axis, and $\omega = 0$ represents dc (direct current).

In a similar fashion, we can determine the stability of a system based on the location of its poles on the z-plane associated with the z-transform, since we can find corresponding regions between the s-plane and the z-plane. Since $z = e^{sT}$ and $s = \sigma + j\omega$,

$$z = e^{\sigma T} e^{j\omega T} \tag{4.12}$$

Hence, the magnitude of z is $|z| = e^{\sigma T}$ with a phase of $\theta = \omega T = 2\pi f/F_s$, where F_s is the sampling frequency. To illustrate the mapping from the s-plane to the z-plane, consider the following regions from Figure 4.1.

σ < 0

Poles on the left side of the $j\omega$ axis (region 2) in the s-plane represent a stable system, and (4.12) yields a magnitude of $|z| < 1$, because $e^{\sigma T} < 1$. As σ varies from $-\infty$ to 0^-, $|z|$ will vary from 0 to 1^-. Hence, poles *inside* the unit circle within region 2 in the z-plane will yield a stable system. The response of such system will be a decaying exponential if the poles are real, or a decaying sinusoid if the poles are complex.

σ > 0

Poles on the right side of the $j\omega$ axis (region 3) in the s-plane represent an unstable system, and (4.12) yields a magnitude of $|z| > 1$, because $e^{\sigma T} > 1$. As σ varies from 0^+

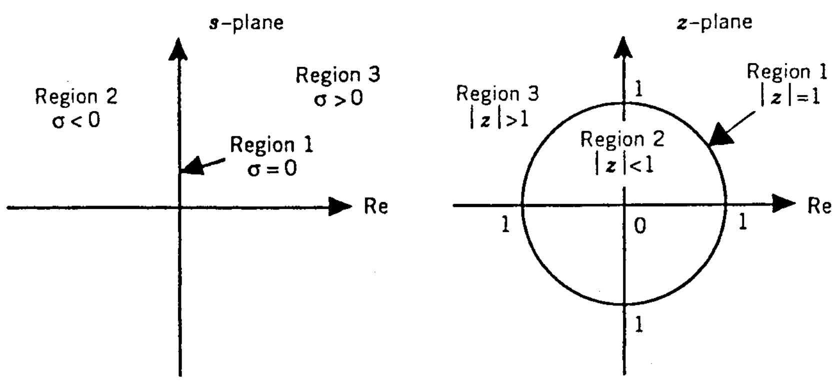

FIGURE 4.1. Mapping from s-plane to z-plane.

to ∞, $|z|$ will vary from 1^+ to ∞. Hence, poles *outside* the unit circle within region 3 in the z-plane will yield an unstable system. The response of such system will be an increasing exponential if the poles are real, or a growing sinusoid if the poles are complex.

σ = 0

Poles on the $j\omega$ axis (region 1) in the s-plane represent a marginally stable system, and (4.12) yields a magnitude of $|z| = 1$, which corresponds to region 1. Hence, poles *on* the unit circle in region 1 in the z-plane will yield a sinusoid. In Chapter 5 we implement a sinusoidal signal by programming a difference equation with its poles *on* the unit circle. Note that from Exercise 4.2 the poles of $X(s) = \sin n\omega T$ in (4.9) or $X(s) = \cos n\omega T$ in (4.11) are the roots of $z^2 - 2z\cos\omega T + 1$, or

$$\begin{aligned} p_{1,2} &= \frac{2\cos\omega T \pm \sqrt{4\cos^2\omega T - 4}}{2} \\ &= \cos\omega T \pm \sqrt{-\sin^2\omega T} = \cos\omega T \pm j\sin\omega T \end{aligned} \tag{4.13}$$

The magnitude of each pole is

$$|p_1| = |p_2| = \sqrt{\cos^2\omega T + \sin^2\omega T} = 1 \tag{4.14}$$

The phase of z is $\theta = \omega T = 2\pi f/F_s$. As the frequency f varies from zero to $\pm F_s/2$, the phase θ will vary from 0 to π.

4.1.2 Difference Equations

A digital filter is represented by a difference equation in a similar fashion as an analog filter is represented by a differential equation. To solve a difference equation, we need to find the z-transform of expressions such as $x(n - k)$, which corresponds to the kth derivative $d^k x(t)/dt^k$ of an analog signal $x(t)$. The order of the difference equation is determined by the largest value of k. For example, $k = 2$ represents a second-order derivative. From (4.5)

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} = x(0) + x(1)z^{-1} + x(2)z^{-2} + \cdots \tag{4.15}$$

Then the z-transform of $x(n - 1)$, which corresponds to a first-order derivative dx/dt, is

$$\begin{aligned} ZT[x(n-1)] &= \sum_{n=0}^{\infty} x(n-1)z^{-n} \\ &= x(-1) + x(0)z^{-1} + x(1)z^{-2} + x(2)z^{-3} + \cdots \\ &= x(-1) + z^{-1}[x(0) + x(1)z^{-1} + x(2)z^{-2} + \cdots] \\ &= x(-1) + z^{-1}X(z) \end{aligned} \tag{4.16}$$

where we used (4.15), and $x(-1)$ represents the initial condition associated with a first-order difference equation. Similarly, the ZT of $x(n-2)$, equivalent to a second derivative $d^2x(t)/dt^2$ is

$$\begin{aligned} ZT[x(n-2)] &= \sum_{n=0}^{\infty} x(n-2)z^{-n} \\ &= x(-2) + x(-1)z^{-1} + x(0)z^{-2} + x(1)z^{-3} + \cdots \\ &= x(-2) + x(-1)z^{-1} + z^{-2}[x(0) + x(1)z^{-1} + \cdots] \\ &= x(-2) + x(-1)z^{-1} + z^{-2}X(z) \end{aligned} \tag{4.17}$$

where $x(-2)$ and $x(-1)$ represent the two initial conditions required to solve a second-order difference equation. In general,

$$ZT[x(n-k)] = z^{-k}\sum_{m=1}^{k} x(-m)z^{m} + z^{k}X(z) \tag{4.18}$$

If the initial conditions are all zero, then $x(-m) = 0$ for $m = 1, 2, \ldots, k$, and (4.18) reduces to

$$ZT[x(n-k)] = z^{-k}X(z) \tag{4.19}$$

4.2 DISCRETE SIGNALS

A discrete signal $x(n)$ can be expressed as

$$x(n) = \sum_{m=-\infty}^{\infty} x(m)\delta(n-m) \tag{4.20}$$

where $\delta(n-m)$ is the impulse sequence $\delta(n)$ delayed by m, which is equal to 1 for $n = m$ and is zero otherwise. It consists of a sequence of values $x(1), x(2), \ldots$, where n is the time, and each sample value of the sequence is taken one sample time apart, determined by the sampling interval or sampling period $T = 1/F_s$.

The signals and systems that we deal with in this book are linear and time-invariant, where both superposition and shift invariance apply. Let an input signal $x(n)$ yield an output response $y(n)$, or $x(n) \rightarrow y(n)$. If $a_1x_1(n) \rightarrow a_1y_1(n)$ and $a_2x_2(n) \rightarrow a_2y_2(n)$, then $a_1x_1(n) + a_2x_2(n) \rightarrow a_1y_1(n) + a_2y_2(n)$, where a_1 and a_2 are constants. This is the superposition property, where an overall output response is the sum of the individual responses to each input. Shift-invariance implies that if the input is delayed by m samples, the output response will also be delayed by m samples, or $x(n-m) \rightarrow y(n-m)$. If the input is a unit impulse $\delta(n)$, the resulting output response is $h(n)$, or $\delta(n) \rightarrow h(n)$, and $h(n)$ is designated as the impulse response. A delayed impulse $\delta(n-m)$ yields the output response $h(n-m)$ by the shift-invariance property.

Furthermore, if this impulse is multiplied by $x(m)$, then $x(m)\delta(n - m) \rightarrow x(m)h(n - m)$. Using (4.20), the response becomes

$$y(n) = \sum_{m=-\infty}^{\infty} x(m)h(n-m) \tag{4.21}$$

which represents a convolution equation. For a causal system, (4.21) becomes

$$y(n) = \sum_{m=-\infty}^{\infty} x(m)h(n-m) \tag{4.22}$$

Letting $k = n - m$ in (4.22) yields

$$y(n) = \sum_{k=0}^{\infty} h(k)x(n-k) \tag{4.23}$$

4.3 FINITE IMPULSE RESPONSE FILTERS

Filtering is one of the most useful signal processing operations [1–47]. Digital signal processors are now available to implement digital filters in real time. The TMS320C6x instruction set and architecture makes it well suited for such filtering operations. An analog filter operates on continuous signals and is typically realized with discrete components such as operational amplifiers, resistors, and capacitors. However, a digital filter, such as a finite impulse response (FIR) filter, operates on discrete-time signals and can be implemented with a digital signal processor such as the TMS320C6x. This involves use of an ADC to capture an external input signal, processing the input samples, and sending the resulting output through a DAC.

Within the last few years, the cost of digital signal processors has been reduced significantly, which adds to the numerous advantages that digital filters have over their analog counterparts. These include higher reliability, accuracy, and less sensitivity to temperature and aging. Stringent magnitude and phase characteristics can be realized with a digital filter. Filter characteristics such as center frequency, bandwidth, and filter type can readily be modified. A number of tools are available to design and implement within a few minutes an FIR filter in real time using the TMS320C6x-based DSK. The filter design consists of the approximation of a transfer function with a resulting set of coefficients.

Different techniques are available for the design of FIR filters, such as a commonly used technique that utilizes the Fourier series, as discussed in the Section 4.4. Computer-aided design techniques such as that of Parks and McClellan are also used for the design of FIR filters [5,6].

The convolution equation (4.23) is very useful for the design of FIR filters, since we can approximate it with a finite number of terms, or

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \tag{4.24}$$

If the input is a unit impulse $x(n) = \delta(0)$, the output impulse response will be $y(n) = h(n)$. We will see in Section 4.4 how to design an FIR filter with N coefficients $h(0), h(1), \ldots, h(N-1)$, and N input samples $x(n), x(n-1), \ldots, x(n-(N-1))$. The input sample at time n is $x(n)$, and the delayed input samples are $x(n-1), \ldots, x(n-(N-1))$. Equation (4.24) shows that an FIR filter can be implemented with knowledge of the input $x(n)$ at time n and of the delayed inputs $x(n-k)$. It is nonrecursive and no feedback or past outputs are required. Filters with feedback (recursive) that require past outputs are discussed in Chapter 5. Other names used for FIR filters are transversal and tapped-delay filters.

The z-transform of (4.24) with zero initial conditions yields

$$Y(z) = h(0)X(z) + h(1)z^{-1}X(z) + h(2)z^{-2}X(z) + \cdots + h(N-1)z^{-(N-1)}X(z) \tag{4.25}$$

Equation (4.24) represents a convolution in time between the coefficients and the input samples, which is equivalent to a multiplication in the frequency domain, or

$$Y(z) = H(z)X(z) \tag{4.26}$$

where $H(z) = ZT[h(k)]$ is the transfer function, or

$$\begin{aligned} H(z) &= \sum_{k=0}^{N-1} h(k)z^{-k} = h(0) + h(1)z^{-1} + h(2)z^{-2} + \cdots + h(N-1)z^{-(N-1)} \\ &= \frac{h(0)z^{(N-1)} + h(1)z^{N-2} + h(2)z^{N-3} + \cdots + h(N-1)}{z^{N-1}} \end{aligned} \tag{4.27}$$

which shows that there are $N-1$ poles, all of which are located at the origin. Hence, this FIR filter is inherently stable, with its poles located only inside the unit circle. We usually describe an FIR filter as a filter with "no poles." Figure 4.2 shows an FIR filter structure representing (4.24) and (4.25).

A very useful feature of an FIR filter is that it can guarantee *linear phase*. The linear phase feature can be very useful in applications such as speech analysis, where phase distortion can be very critical. For example, with linear phase, all input sinusoidal components are delayed by the same amount. Otherwise, harmonic distortion can occur.

The Fourier transform of a delayed input sample $x(n-k)$ is $e^{-j\omega kT}X(j\omega)$, yielding a phase of $\theta = -\omega kT$, which is a linear function in terms of ω. Note that the group delay function, defined as the derivative of the phase, is a constant, or $d\theta/d\omega = -kT$.

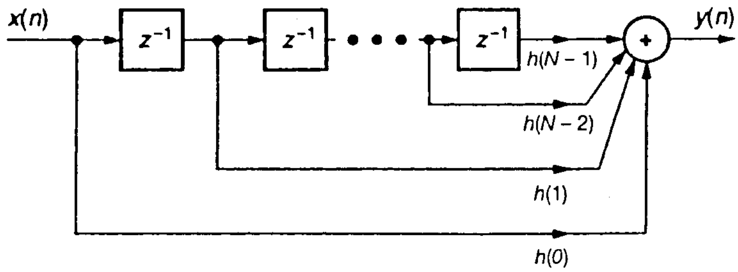

FIGURE 4.2. FIR filter structure showing delays.

4.4 FIR IMPLEMENTATION USING FOURIER SERIES

The design of an FIR filter using a Fourier series method is such that the magnitude response of its transfer function $H(z)$ approximates a desired magnitude response. The transfer function desired is

$$H_d(\omega) = \sum_{n=-\infty}^{\infty} C_n e^{jn\omega T} \qquad |n| < \infty \tag{4.28}$$

where C_n are the Fourier series coefficients. Using a normalized frequency variable ν such that $\nu = f/F_N$, where F_N is the Nyquist frequency, or $F_N = F_s/2$, the desired transfer function in (4.28) can be written as

$$H_d(\nu) = \sum_{n=-\infty}^{\infty} C_n e^{jn\pi\nu} \tag{4.29}$$

where $\omega T = 2\pi f/F_s = \pi\nu$ and $|\nu| < 1$. The coefficients C_n are defined as

$$\begin{aligned} C_n &= \frac{1}{2}\int_{-1}^{1} H_d(\nu) e^{-jn\pi\nu} d\nu \\ &= \frac{1}{2}\int_{-1}^{1} H_d(\nu)(\cos n\pi\nu - j\sin n\pi\nu) d\nu \end{aligned} \tag{4.30}$$

Assume that $H_d(\nu)$ is an even function (frequency selective filter); then (4.30) reduces to

$$C_n = \int_{0}^{1} H_d(\nu)\cos n\pi\nu d\nu \qquad n \geqq 0 \tag{4.31}$$

since $H_d(\nu)$ $\sin n\pi\nu$ is an odd function and

$$\int_{-1}^{1} H_d(\nu)\sin n\pi\nu d\nu = 0$$

with $C_n = C_{-n}$. The desired transfer function $H_d(\nu)$ in (4.29) is expressed in terms of an infinite number of coefficients, and to obtain a realizable filter, we must truncate (4.29), which yields the approximated transfer function

$$H_a(\nu) = \sum_{n=-Q}^{Q} C_n e^{jn\pi\nu} \tag{4.32}$$

where Q is positive and finite and determines the order of the filter. The larger the value of Q, the higher the order of the FIR filter and the better the approximation in (4.32) of the desired transfer function. The truncation of the infinite series with a finite number of terms results in ignoring the contribution of the terms outside a rectangular window function between $-Q$ and $+Q$. In Section 4.5 we see how the characteristics of a filter can be improved by using window functions other than rectangular.

Let $z = e^{j\pi\nu}$; then (4.32) becomes

$$H_a(z) = \sum_{n=-Q}^{Q} C_n z^n \tag{4.33}$$

with the impulse response coefficients $C_{-Q}, C_{-Q+1}, \ldots, C_{-1}, C_0, C_1, \ldots, C_{Q-1}, C_Q$. The approximated transfer function in (4.33), with positive powers of z, implies a noncausal or not realizable filter that would produce an output before an input is applied. To remedy this situation, we introduce a delay of Q samples in (4.33) to yield

$$H(z) = z^{-Q} H_a(z) = \sum_{n=-Q}^{Q} C_n z^{n-Q} \tag{4.34}$$

Let $n - Q = -i$; then $H(z)$ in (4.34) becomes

$$H(z) = \sum_{i=0}^{2Q} C_{Q-i} z^{-i} \tag{4.35}$$

Let $h_i = C_{Q-i}$ and $N - 1 = 2Q$; then $H(z)$ becomes

$$H(z) = \sum_{i=0}^{N-1} h_i z^{-i} \tag{4.36}$$

where $H(z)$ is expressed in terms of the impulse response coefficients h_i, and $h_0 = C_Q, h_1 = C_{Q-1}, \ldots, h_Q = C_0, h_{Q+1} = C_{-1} = C_1, \ldots, h_{2Q} = C_{-Q}$. The impulse response coefficients are symmetric about h_Q, with $C_n = C_{-n}$.

The order of the filter is $N = 2Q + 1$. For example, if $Q = 5$, the filter will have 11 coefficients $h_0, h_1, \ldots, h_{10}$, or

$$h_0 = h_{10} = C_5$$
$$h_1 = h_9 = C_4$$
$$h_2 = h_8 = C_3$$
$$h_3 = h_7 = C_2$$
$$h_4 = h_6 = C_1$$
$$h_5 = C_0$$

Figure 4.3 shows the desired transfer functions $H_d(\nu)$ ideally represented for the frequency-selective filters: lowpass, highpass, bandpass, and bandstop for which the coefficients $C_n = C_{-n}$ can be found.

1. *Lowpass:* $C_0 = \nu_1$

$$C_n = \int_0^{\nu_1} H_d(\nu)\cos n\pi\nu\, d\nu = \frac{\sin n\pi\nu_1}{n\pi} \tag{4.37}$$

2. *Highpass:* $C_0 = 1 - \nu_1$

$$C_n = \sum_{\nu_1}^{1} H_d(\nu)\cos n\pi\nu\, d\nu = -\frac{\sin n\pi\nu_1}{n\pi} \tag{4.38}$$

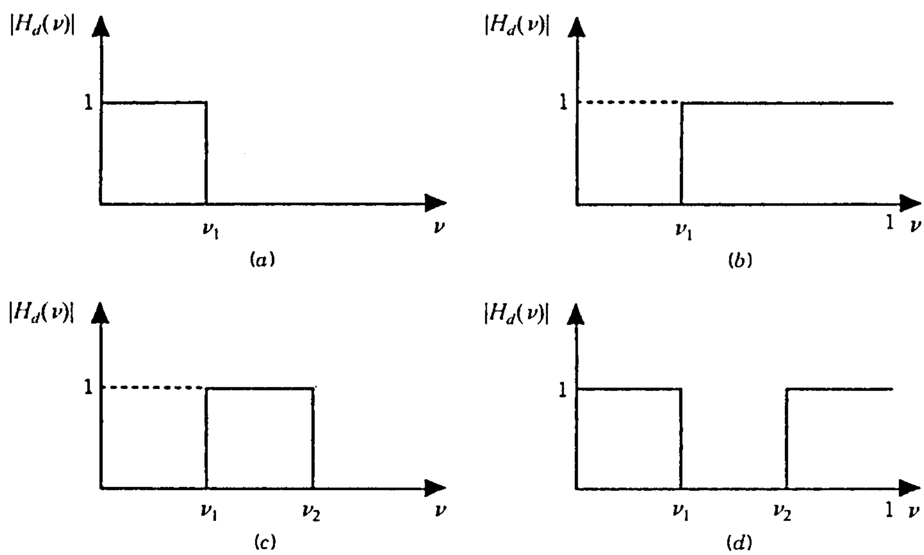

FIGURE 4.3. Desired transfer function: (*a*) lowpass; (*b*) highpass; (*c*) bandpass; (*d*) bandstop.

3. *Bandpass:* $C_0 = \nu_2 - \nu_1$

$$C_n = \int_{\nu_1}^{\nu_2} H_d(\nu)\cos n\pi\nu \, d\nu = \frac{\sin n\pi\nu_2 - \sin n\pi\nu_1}{n\pi} \tag{4.39}$$

4. *Bandstop:* $C_0 = 1 - (\nu_2 - \nu_1)$

$$C_n = \int_0^{\nu_1} H_d(\nu)\cos n\pi\nu \, d\nu + \int_{\nu_2}^{1} H_d(\nu)\cos n\pi\nu \, d\nu = \frac{\sin n\pi\nu_1 - \sin n\pi\nu_2}{n\pi} \tag{4.40}$$

where ν_1 and ν_2 are the normalized cutoff frequencies shown in Figure 4.3. Several filter-design packages are currently available for the design of FIR filters, as discussed later. When we implement an FIR filter, we develop a generic program such that the specific coefficients will determine the filter type (e.g., whether lowpass or bandpass).

Exercise 4.3: Lowpass FIR Filter

We will find the impulse response coefficients of an FIR filter with $N = 11$, a sampling frequency of 10kHz, and a cutoff frequency $f_c = 1$kHz. From (4.37),

$$C_0 = \nu_1 = \frac{f_c}{F_N} = 0.2$$

where $F_N = F_s/2$ is the Nyquist frequency and

$$C_n = \frac{\sin 0.2n\pi}{n\pi} \qquad n = \pm 1, \pm 2, \ldots, \pm 5 \tag{4.41}$$

Since the impulse response coefficients $h_i = C_{Q-i}$, $C_n = C_{-n}$, and $Q = 5$, the impulse response coefficients are

$$\begin{aligned} h_0 &= h_{10} = 0 & h_3 &= h_7 = 0.1514 \\ h_1 &= h_9 = 0.0468 & h_4 &= h_6 = 0.1872 \\ h_2 &= h_8 = 0.1009 & h_5 &= 0.2 \end{aligned} \tag{4.42}$$

These coefficients can be calculated with a utility program (on the accompanying disk) and inserted within a generic filter program, as described later. Note the symmetry of these coefficients about $Q = 5$. While $N = 11$ for an FIR filter is low for a practical design, doubling this number can yield an FIR filter with much better characteristics, such as selectivity. For an FIR filter to have linear phase, the coefficients must be symmetric, as in (4.42).

4.5 WINDOW FUNCTIONS

We truncated the infinite series in the transfer function equation (4.29) to arrive at (4.32). We essentially put a rectangular window function with an amplitude of 1 between $-Q$ and $+Q$ and ignored the coefficients outside that window. The wider this rectangular window, the larger Q is and the more terms we use in (4.32) to get a better approximation of (4.29). The rectangular window function can therefore be defined as

$$w_R(n) = \begin{cases} 1 & \text{for } |n| \le Q \\ 0 & \text{otherwise} \end{cases} \tag{4.43}$$

The transform of the rectangular window function $\omega_R(n)$ yields a sinc function in the frequency domain. It can be shown that

$$W_R(\nu) = \sum_{n=-Q}^{Q} e^{jn\pi\nu} = e^{-jQ\pi\nu}\left(\sum_{n=0}^{2Q} e^{jn\pi\nu}\right) = \frac{\sin\left[\left(\frac{2Q+1}{2}\right)\pi\nu\right]}{\sin(\pi\nu/2)} \tag{4.44}$$

which is a sinc function that exhibits high sidelobes or oscillations caused by the abrupt truncation, specifically, near discontinuities.

A number of window functions are currently available to reduce these high-amplitude oscillations; they provide a more gradual truncation to the infinite series expansion. However, while these alternative window functions reduce the amplitude of the sidelobes, they also have a wider mainlobe, which results in a filter with lower selectivity. A measure of a filter's performance is a ripple factor that compares the peak of the first sidelobe to the peak of the main lobe (their ratio). A compromise or trade-off is to select a window function that can reduce the sidelobes while approaching the selectivity that can be achieved with the rectangular window function. The width of the mainlobe can be reduced by increasing the width of the window (order of the filter). We will later plot the magnitude response of an FIR filter that shows the undesirable sidelobes.

In general, the Fourier series coefficients can be written as

$$C'_n = C_n w(n) \tag{4.45}$$

where $w(n)$ is the window function. In the case of the rectangular window function, $C'_n = C_n$. The transfer function in (4.36) can then be written as

$$H'(z) = \sum_{i=0}^{N-1} h'_i z^{-i} \tag{4.46}$$

where

$$h_i' = C_{Q-i}' \qquad 0 \le i \le 2Q \tag{4.47}$$

The rectangular window has its highest sidelobe level, down by only −13 dB from the peak of its mainlobe, resulting in oscillations with an amplitude of considerable size. On the other hand, it has the narrowest mainlobe that can provide high selectivity. The following window functions are commonly used in the design of FIR filters [12].

4.5.1 Hamming Window

The Hamming window function [12,25] is

$$w_H(n) = \begin{cases} 0.54 + 0.46\cos(n\pi/Q) & \text{for } |n| \le Q \\ 0 & \text{otherwise} \end{cases} \tag{4.48}$$

which has the highest or first sidelobe level at approximately −43 dB from the peak of the main lobe.

4.5.2 Hanning Window

The Hanning or raised cosine window function is

$$w_{HA}(n) = \begin{cases} 0.5 + 0.5\cos(n\pi/Q) & \text{for } |n| \le Q \\ 0 & \text{otherwise} \end{cases} \tag{4.49}$$

which has the highest or first sidelobe level at approximately −31 dB from the peak of the mainlobe.

4.5.3 Blackman Window

The Blackman window function is

$$w_B(n) = \begin{cases} 0.42 + 0.5\cos(n\pi/Q) + 0.08\cos(2n\pi/Q) & |n| \le Q \\ 0 & \text{otherwise} \end{cases} \tag{4.50}$$

which has the highest sidelobe level down to approximately −58 dB from the peak of the mainlobe. While the Blackman window produces the largest reduction in the sidelobe compared with the previous window functions, it has the widest mainlobe. As with the previous windows, the width of the mainlobe can be decreased by increasing the width of the window.

4.5.4 Kaiser Window

The design of FIR filters with the Kaiser window has become very popular in recent years. It has a variable parameter to control the size of the sidelobe with respect to the mainlobe. The Kaiser window function is

$$w_K(n) = \begin{cases} I_0(b)/I_0(a) & |n| \le Q \\ 0 & \text{otherwise} \end{cases} \tag{4.51}$$

where a is an empirically determined variable, and $b = a[1 - (n/Q)^2]^{1/2}$. $I_0(x)$ is the modified Bessel function of the first kind defined by

$$I_0(x) = 1 + \frac{0.25x^2}{(1!)^2} + \frac{(0.25x^2)^2}{(2!)^2} + \cdots = 1 + \sum_{n=1}^{\infty}\left[\frac{(x/2)^n}{n!}\right]^2 \tag{4.52}$$

which converges rapidly. A trade-off between the size of the sidelobe and the width of the mainlobe can be achieved by changing the length of the window and the parameter a.

4.5.5 Computer-Aided Approximation

An efficient technique is the computer-aided iterative design based on the Remez exchange algorithm, which produces equiripple approximation of FIR filters [5,6]. The order of the filter and the edges of both passbands and stopbands are fixed, and the coefficients are varied to provide this equiripple approximation. This minimizes the ripple in both the passbands and the stopbands. The transition regions are left unconstrained and are considered as "don't care" regions, where the solution may fail. Several commercial filter design packages include the Parks–McClellan algorithm for the design of an FIR filter.

4.6 PROGRAMMING EXAMPLES USING C AND ASM CODE

Within minutes, an FIR filter can be designed and implemented in real time. Several filter design packages are available for the design of FIR filters. They are described in Appendix D using MATLAB [48] and in Appendix E using DigiFilter and a homemade package (on the accompanying disk).

Several examples illustrate the implementation of FIR filters. Most of the programs are in C. A few examples using mixed C and ASM code illustrate the use of a circular buffer as a more efficient way to update delay samples, with the circular buffer in internal or external memory. The convolution equation (4.24) is used to program and implement these filters, or

$$y(n) = \sum_{i=0}^{N-1} h(i)x(n-i)$$

We can arrange the impulse response coefficients within a buffer (array) so that the first coefficient, $h(0)$, is at the beginning (first location) of the buffer (lower-memory address). The last coefficient, $h(N-1)$, can reside at the end (last location) of the coefficients buffer (higher-memory address). The delay samples are organized in memory so that the newest sample, $x(n)$, is at the beginning of the samples buffer, while the oldest sample, $x(n-(N-1))$, is at the end of the buffer. The coefficients and the samples can be arranged in memory as shown in Table 4.1. Initially, all the samples are set to zero.

Time n

The newest sample, $x(n)$, at time n is acquired from an ADC and stored at the beginning of the sample buffer. The filter's output at time n is computed from the convolution equation (4.24), or

$$y(n) = h(0)x(n) + h(1)x(n-1) + \cdots + h(N-2)x(n-(N-2)) \\ + h(N-1)x(n-(N-1))$$

The delay samples are then updated, so that $x(n-k) = x(n+1-k)$ can be used to calculate $y(n+1)$, the output for the next unit of time, or sample period T_s. All the samples are updated except the newest sample. For example, $x(n-1) = x(n)$, and $x(n-(N-1)) = x(n-(N-2))$. This updating process has the effect of "moving the data" (down) in memory (see Table 4.2, associated with time $n+1$).

Time n + 1

At time $n+1$, a new input sample $x(n+1)$ is acquired and stored at the top of the sample buffer, as shown in Table 4.2. The output $y(n+1)$ can now be calculated as

TABLE 4.1 Memory Organization for Coefficients and Samples (Initially)

i	Coefficients	Samples
0	h(0)	x(n)
1	h(1)	x(n - 1)
2	h(2)	x(n - 2)
.	.	.
.	.	.
.	.	.
N - 1	h(N - 1)	x(n - (N - 1))

TABLE 4.2 Memory Organization to Illustrate Update of Samples

	Samples		
Coefficients	Time n	Time n + 1	Time n + 2
h(0)	x(n)	x(n + 1)	x(n + 2)
h(1)	x(n - 1)	x(n)	x(n + 1)
h(2)	x(n - 2)	x(n - 1)	x(n)
.	.	.	.
.	.	.	.
.	.	.	.
h(N - 3)	x(n - (N - 3))	x(n - (N - 4))	x(n - (N - 5))
h(N - 2)	x(n - (N - 2))	x(n - (N - 3))	x(n - (N - 4))
h(N - 1)	x(n - (N - 1))	x(n - (N - 2))	x(n - (N - 3))

$$y(n+1) = h(0)x(n+1) + h(1)x(n) + \cdots + h(N-2)x(n-(N-3)) \\ + h(N-1)x(n-(N-2))$$

The samples are then updated for the next unit of time.

Time n + 2

At time $n + 2$, a new input sample, $x(n + 2)$, is acquired. The output becomes

$$y(n+2) = h(0)x(n+2) + h(1)x(n+1) + \cdots + h(N-1)x(n-(N-3))$$

This process continues to calculate the filter's output and updating the delay samples at each unit of time (sample period).

Example 4.8 illustrates four different ways of arranging the coefficients and samples in memory and of calculating the convolution equation (e.g., the newest sample at the end of the buffer and the oldest sample at the beginning).

Example 4.1: FIR Filter Implementation: Bandstop and Bandpass (`FIR`)

Figure 4.4 shows a listing of the C source program *`FIR.c`*, which implements an FIR filter. It is a generic FIR program, since the coefficient file included, *`bs2700.cof`* (Figure 4.5), specifies the filter's characteristics. This coefficient file, which contains 89 coefficients, represents an FIR bandstop (notch) filter centered at 2700 Hz. The number of coefficients N is defined in the coefficient file. This filter was designed using MATLAB's graphical user interface (GUI) filter designer SPTOOL, described in Appendix D. Figure 4.6 shows the filter's characteristics (MATLAB's order of 88 corresponds to 89 coefficients).

A buffer `dly[N]` is created for the delay samples. The newest input sample, *x(n)*, is acquired through `dly[0]` and stored at the beginning of the buffer. The

```
//Fir.c FIR filter. Include coefficient file with length N

#include "bs2700.cof"              //coefficient file BS @ 2700Hz
int yn = 0;                        //initialize filter's output
short dly[N];                      //delay samples

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

      dly[0] = input_sample();     //newest input @ top of buffer
      yn = 0;                      //initialize filter's output
      for (i = 0; i< N; i++)
         yn += (h[i] * dly[i]);    //y(n) += h(i)* x(n-i)
      for (i = N-1; i > 0; i--)    //starting @ bottom of buffer
         dly[i] = dly[i-1];        //update delays with data move

      output_sample(yn >> 15);     //output filter
      return;
}

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

FIGURE 4.4. Generic FIR program (FIR.c).

```
//BS2700.cof FIR bandstop coefficients designed with MATLAB

#define N 89                      //number of coefficients

short h[N]={-14,23,-9,-6,0,8,16,-58,50,44,-147,119,67,-245,
             200,72,-312,257,53,-299,239,20,-165,88,0,105,
            -236,33,490,-740,158,932,-1380,392,1348,-2070,
             724,1650,-2690,1104,1776,-3122,1458,1704,29491,
             1704,1458,-3122,1776,1104,-2690,1650,724,-2070,
             1348,392,-1380,932,158,-740,490,33,-236,105,0,
             88,-165,20,239,-299,53,257,-312,72,200,-245,67,
             119,-147,44,50,-58,16,8,0,-6,-9,23,-14};
```

FIGURE 4.5. Coefficients for a FIR bandstop filter (bs2700.cof).

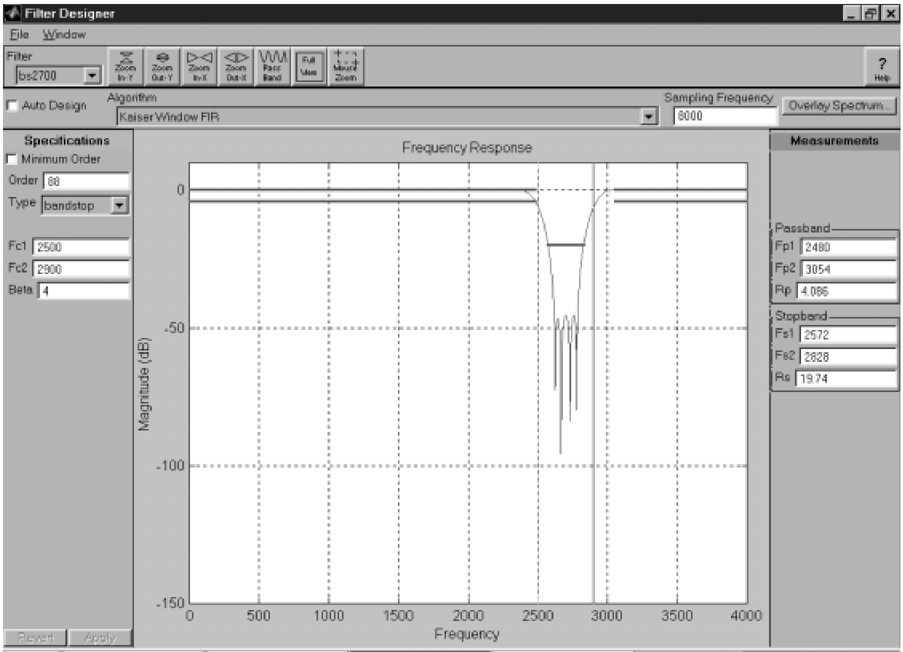

FIGURE 4.6. MATLAB's filter designer SPTOOL, showing the characteristics of a FIR bandstop filter centered at 2700 Hz.

coefficients are stored in another buffer, $h[N]$, with $h[0]$ at the beginning of the coefficients' buffer. The samples and coefficients are then arranged in their respective buffer, as shown in Table 4.1.

Two "for" loops are used within the interrupt service routine (we will also implement an FIR filter using one loop). The first loop implements the convolution equation with N coefficients and N delay samples, for a specific time n. At time n the output is

$$y(n) = h(0)x(n) + h(1)x(n-1) + \cdots + h(N-1)x(n-(N-1))$$

The delay samples are then updated within the second loop to be used for calculating $y(n)$ at time $n+1$, or $y(n+1)$. The newly acquired input sample always resides at the beginning of the samples buffer (in this example). The memory location that contained the sample $x(n)$ now contains the newly acquired sample $x(n+1)$. The output $y(n+1)$ at time $n+1$ is then calculated. This scheme uses a data move to update the delay samples.

Example 4.8 illustrates how various memory organizations can be used for both the delay samples and the filter coefficients, as well as updating the delay samples within the same loop as the convolution equation. We also illustrate the use of a circular buffer with a pointer to update the delay samples, in lieu of moving the data in memory. The output is scaled (right-shifted by 15) before it is sent to the codec's DAC. This allows for a fixed-point implementation as well.

Bandstop, Centered at 2700 Hz (`bs2700.cof`)

Build and run this project as **`FIR`**. Input a sinusoidal signal and vary the input frequency slightly below and above 2700 Hz. Verify that the output is a minimum at 2700 Hz.

Figure 4.7 shows a plot of CCS project windows. It shows the FFT magnitude of the filter's coefficients *h* (see Example 1.3, using a starting address of `h`) using a 128-point FFT. The characteristics of the FIR bandstop filter, centered at 2700 Hz, are displayed. It also shows a CCS time-domain plot, or the impulse response of the filter.

With noise as input, the output frequency response of the bandpass filter can also be verified. The pseudorandom noise sequence developed in Chapter 2, or another noise source (see Appendix D), can be used as input to the FIR filter, as illustrated later. Figure 4.8 shows a plot of the frequency response of the filter with a notch at 2700 Hz implemented in real time. This plot is obtained using an HP 3561A dynamic signal analyzer with an input noise source from the analyzer. The roll-off at approximately 3500 Hz is due to the antialiasing lowpass filter on the codec.

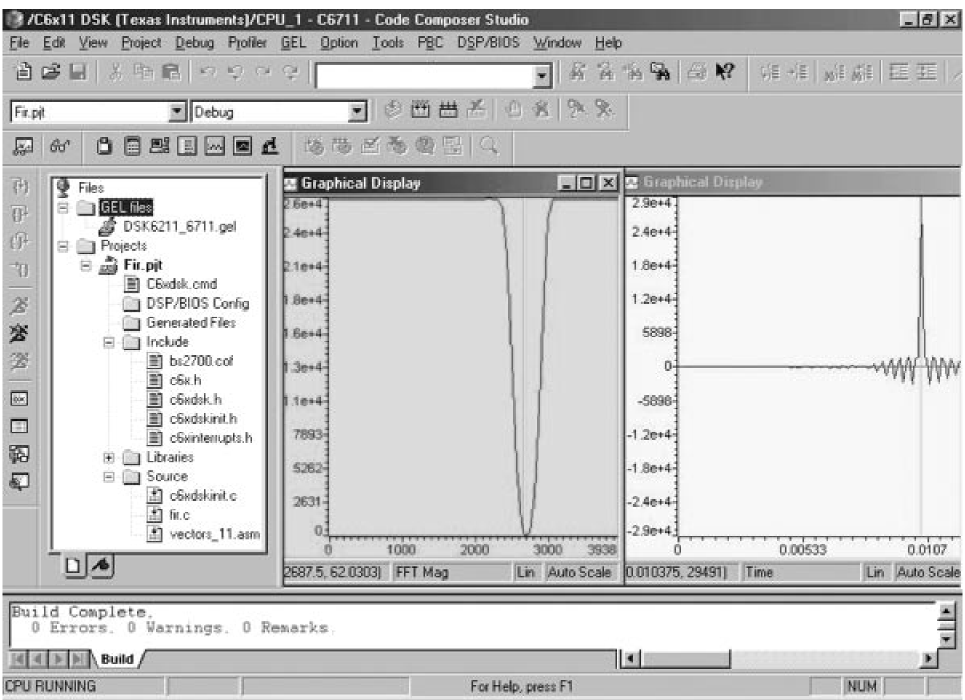

FIGURE 4.7. CCS plots displaying the FFT magnitude of the bandstop filter's coefficients and its impulse response.

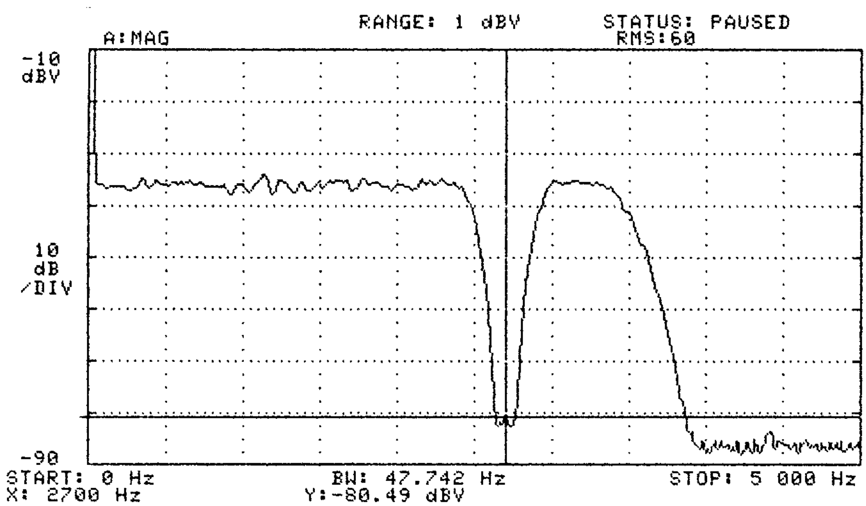

FIGURE 4.8. Output frequency response of FIR bandstop filter centered at 2700 Hz, obtained with a signal analyzer.

Bandpass, Centered at 1750 Hz (`bp1750.cof`)

Within CCS, edit the program `FIR.c` to include the coefficient file *`bp1750.cof`* in lieu of *`bs2700.cof`*. The file *`bp1750.cof`* represents an FIR bandpass filter (81 coefficients) centered at 1750 Hz, as shown in Figure 4.9. This filter was designed

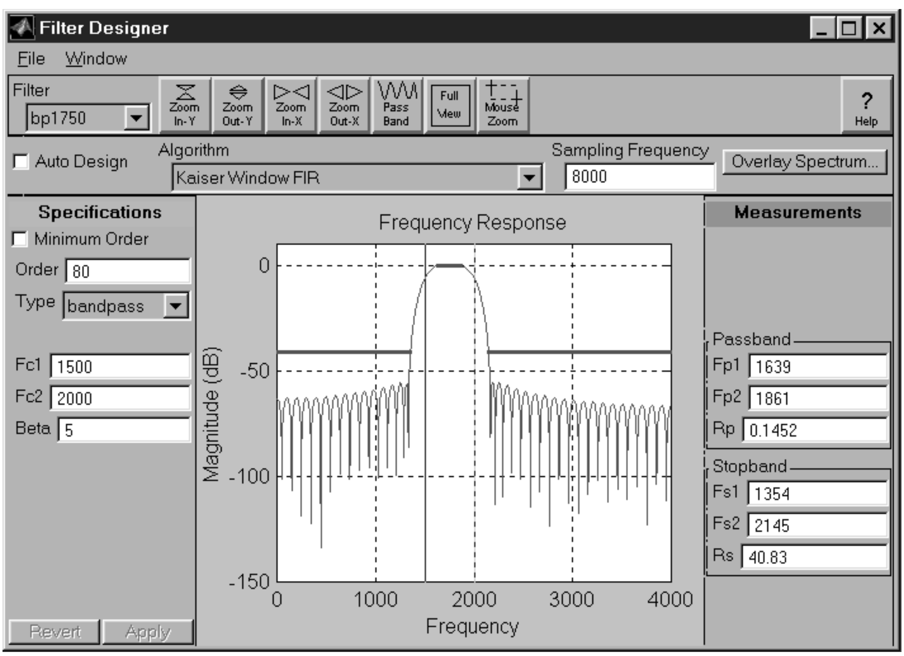

FIGURE 4.9. MATLAB's filter designer SPTOOL, showing characteristics of a FIR bandpass filter centered at 1750 Hz.

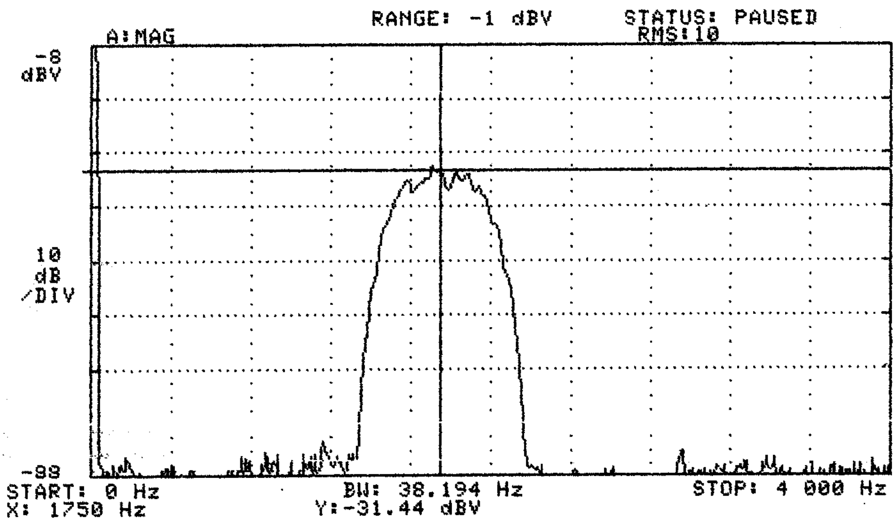

FIGURE 4.10. Output frequency response of a FIR bandpass filter centered at 1750 Hz, obtained with a signal analyzer.

with MATLAB's SPTOOL (Appendix D). Select the incremental Build and the new coefficient file *bp1750.cof* will automatically be included in the project. Run again and verify an FIR bandpass filter centered at 1750 Hz. Figure 4.10 shows a real-time plot of the output frequency response obtained with the HP signal analyzer.

Example 4.2: Effects on Voice Using Three FIR Lowpass Filters (FIR3LP)

Figure 4.11 shows a listing of the program *FIR3lp.c*, which implements three FIR lowpass filters with cutoff frequencies at 600, 1500, and 3000 Hz, respectively. The three lowpass filters were designed with MATLAB's SPTOOL to yield the corresponding three sets of coefficients. This example expands on the generic FIR program in Example 4.1.

`LP_number` selects the desired lowpass filter to be implemented. For example, if `LP_number` is set to 1, `h[1][i]` is equal to `hlp600[i]` (within the "for" loop in the function *main*), which is the address of the first set of coefficients. The coefficients file `LP600.cof` represents an 81-coefficient FIR lowpass filter with a 600-Hz cutoff frequency, using the Kaiser window function. Figure 4.12 shows a listing of this coefficient file (the other two sets are on the disk). That filter is then implemented. `LP_number` can be changed to 2 or 3 to implement the 1500- or 3000-Hz lowpass filter, respectively. With the GEL file `FIR3LP.gel` (Figure 4.13), one can vary `LP_number` from 1 to 3 and slide through the three different filters.

```
//FIR3LP.c FIR using three lowpass coefficients with three different BW

#include "lp600.cof"                    //coeff file LP @ 600 Hz
#include "lp1500.cof"                   //coeff file LP @ 1500 Hz
#include "lp3000.cof"                   //coeff file LP @ 3000 Hz
short LP_number = 1;                    //start with 1st LP filter
int   yn = 0;                           //initialize filter's output
short dly[N];                           //delay samples
short h[3][N];                          //filter characteristics 3xN

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

      dly[0] = input_sample();          //newest input @ top of buffer
      yn = 0;                           //initialize filter output
      for (i = 0; i< N; i++)
         yn +=(h[LP_number][i]*dly[i]); //y(n) += h(LP#,i)*x(n-i)
      for (i = N-1; i > 0; i--)         //starting @ bottom of buffer
         dly[i] = dly[i-1];             //update delays with data move
      output_sample(yn >> 15);          //output filter
      return;                           //return from interrupt
}

void main()
{
      short i;

      for (i=0; i<N; i++)
         {
            dly[i] = 0;                 //init buffer
            h[1][i] = hlp600[i];        //start addr of LP600 coeff
            h[2][i] = hlp1500[i];       //start addr of LP1500 coeff
            h[3][i] = hlp3000[i];       //start addr of LP3000 coeff
         }
      comm_intr();                      //init DSK, codec, McBSP
      while(1);                         //infinite loop
}
```

FIGURE 4.11. FIR program to implement three different FIR lowpass filters using a slider for selection (`FIR3LP.c`).

Build this project as **`FIR3LP`**. Use the `.wav` file *`TheForce.wav`* (on the disk) as input (see Appendix D) and observe the effects of the three lowpass filters on the input voice. With the lower bandwidth of 600 Hz, using the first set of coefficients, the frequency components of the speech signal above 600 Hz are suppressed. Connect the output to a speaker or a spectrum analyzer to verify such results, and observe the different bandwidths of the three FIR lowpass filters.

```
//LP600.cof FIR lowpass filter coefficients using Kaizer window

#define N 81        //length of filter

short hlp600[N] = {0,-6,-14,-22,-26,-24,-13,8,34,61,80,83,63,19,-43,-113,
-171,-201,-185,-117,0,146,292,398,428,355,174,-99,-416,-712,-905,-921,
-700,-218,511,1424,2425,3391,4196,4729,4915,4729,4196,3391,2425,1424,
511,-218,-700,-921,-905,-712,-416,-99,174,355,428,398,292,146,0,-117,
-185,-201,-171,-113,-43,19,63,83,80,61,34,8,-13,-24,-26,-22,-14,-6,0};
```

FIGURE 4.12. Coefficient file for a FIR lowpass filter with 600-Hz cutoff frequency (LP600.cof).

```
/*FIR3LP.gel Gel file to step through 3 different LP filters*/

menuitem "Filter Characteristics"

slider Filter(1,3,1,1,filterparameter)   /*from 1 to 3,incr by 1*/
{
      LP_number = filterparameter;       /*for 3 LP filters*/
}
```

FIGURE 4.13. GEL file for selecting one of three FIR lowpass filter coefficients (FIR3LP.gel).

Example 4.3: Implementation of Four Different Filters: Lowpass, Highpass, Bandpass, and Bandstop (FIR4types)

This example is similar to Example 4.2 and illustrates the gel (slider) file to step through four different types of FIR filters. Each filter has 81 coefficients, designed with MATLAB's SPTOOL. The four coefficient files (on the accompanying disk) are:

1. *lp1500.cof*: lowpass with bandwidth of 1500 Hz
2. *hp2200.cof*: highpass with bandwidth of 2200 Hz
3. *bp1750.cof*: bandpass with center frequency at 1750 Hz
4. *bs790.cof*: bandstop with center frequency at 790 Hz

The program *FIR4types.c* (on disk) implements this project. The program FIR3LP.c (Example 4.2) is modified slightly to incorporate a fourth filter.

Build and run this project as **FIR4types**. Load the GEL file *FIR4types.gel* (on the disk) and verify the implementation of the four different FIR filters. This example can readily be expanded to implement more FIR filters.

Figure 4.9 shows the characteristics of the FIR bandpass filter centered at 1750 Hz obtained with MATLAB's filter designer; and Figure 4.10 shows its frequency response obtained with an HP signal analyzer.

```
//FIRPRN.c FIR with internally generated input noise sequence

#include "bp55.cof"                  //BP @ Fs/4 coeff file in float
#include "noise_gen.h"               //header file for noise sequence
int dly[N];                          //delay samples
short fb;                            //feedback variable
shift_reg sreg;

short prn(void)                      //pseudorandom noise generation
{
 short prnseq;                       //for pseudorandom sequence

 if(sreg.bt.b0)                      //sequence {1,-1}
      prnseq = -16000;               //scaled negative noise level
 else
      prnseq = 16000;                //scaled positive noise level
 fb =(sreg.bt.b0)^(sreg.bt.b1);      //XOR bits 0,1
 fb ^=(sreg.bt.b11)^(sreg.bt.b13);   //with bits 11,13 ->fb
 sreg.regval<<=1;                    //shift register 1 bit to left
 sreg.bt.b0 = fb;                    //close feedback path

 return prnseq;                      //return sequence
}

interrupt void c_int11()             //ISR
{
      int i;
      int yn = 0;                    //initialize filter's output

      dly[0] = prn();                //input noise sequence
      for (i = 0; i< N; i++)
         yn +=(h[i]*dly[i]);         //y(n)+= h(i)*x(n-i)
      for (i = N-1; i > 0; i--)      //start @ bottom of buffer
         dly[i] = dly[i-1];          //data move to update delays

      output_sample(yn);             //output filter
      return;                        //return from interrupt
}

void main()
{
      short i;

      sreg.regval = 0xFFFF;          //shift register to nominal values
      fb = 1;                        //initial feedback value
      for (i = 0; i<N; i++)
         dly[i] = 0;                 //init buffer
      comm_intr();                   //init DSK, codec, McBSP
      while(1);                      //infinite loop
}
```

FIGURE 4.14. FIR program with pseudorandom noise sequence as input (FIRPRN.c).

Example 4.4: FIR Implementation with Pseudorandom Noise Sequence as Input to Filter (FIRPRN)

The program *FIRPRN.c* (Figure 4.14) implements an FIR filter using an internally generated pseudorandom noise as input to the filter. This input is the pseudorandom noise sequence generated in Example 2.16. The coefficient file *BP55.cof* uses a float data format and is shown in Figure 4.15. [A filter development package (on disk) that generates filter coefficients in float or hexadecimal format is described in Appendix E.] It represents a 55-coefficient FIR bandpass filter with a center frequency at $F_s/4$.

Build this project as **FIRPRN**. Run this project and verify that the output is an FIR bandpass filter centered at 2kHz. To verify the output as the noise sequence, output `dly[0]` in lieu of `yn` when calling the function `output_sample`.

Testing Different FIR Filters

Halt the program. Edit the C source program to include and test different coefficient files (on the disk) that represent different FIR filters, all using float format values. Each coefficient file contains 55 coefficients (except `comb14.cof`).

1. `BS55.cof`: bandstop with center frequency $F_s/4$
2. `BP55.cof`: bandpass with center frequency $F_s/4$
3. `LP55.cof`: lowpass with cutoff frequency $F_s/4$
4. `HP55.cof`: highpass with bandwidth $F_s/4$
5. `Pass2b.cof`: with two passbands
6. `Pass3b.cof`: with three passbands

```
//bp55.cof Coefficients for bandpass FIR filter centered @ Fs/4

#define N 55                    //number of coefficients

float h[N]=
{1.7619E-017, 7.0567E-003, 2.2150E-018,-1.0962E-002, 4.0310E-017,
 1.3946E-002, 7.1787E-018,-1.4588E-002, 3.9928E-017, 1.1474E-002,
 5.9881E-018,-3.5159E-003,-6.6174E-018,-9.7476E-003,-1.7919E-017,
 2.7932E-002,-9.4329E-017,-4.9740E-002, 3.3834E-017, 7.3066E-002,
-3.6228E-017,-9.5284E-002, 3.2194E-017, 1.1365E-001,-2.2165E-017,
-1.2576E-001, 7.8980E-018, 1.3000E-001, 7.8980E-018,-1.2576E-001,
-2.2165E-017, 1.1365E-001, 3.2194E-017,-9.5284E-002,-3.6228E-017,
 7.3066E-002, 3.3834E-017,-4.9740E-002,-9.4329E-017, 2.7932E-002,
-1.7919E-017,-9.7476E-003,-6.6174E-018,-3.5159E-003, 5.9881E-018,
 1.1474E-002, 3.9928E-017,-1.4588E-002, 7.1787E-018, 1.3946E-002,
 4.0310E-017,-1.0962E-002, 2.2150E-018, 7.0567E-003, 1.7619E-017};
```

FIGURE 4.15. Coefficient file in float format for a FIR bandpass filter centered at $F_s/4$ (`BP55.cof`).

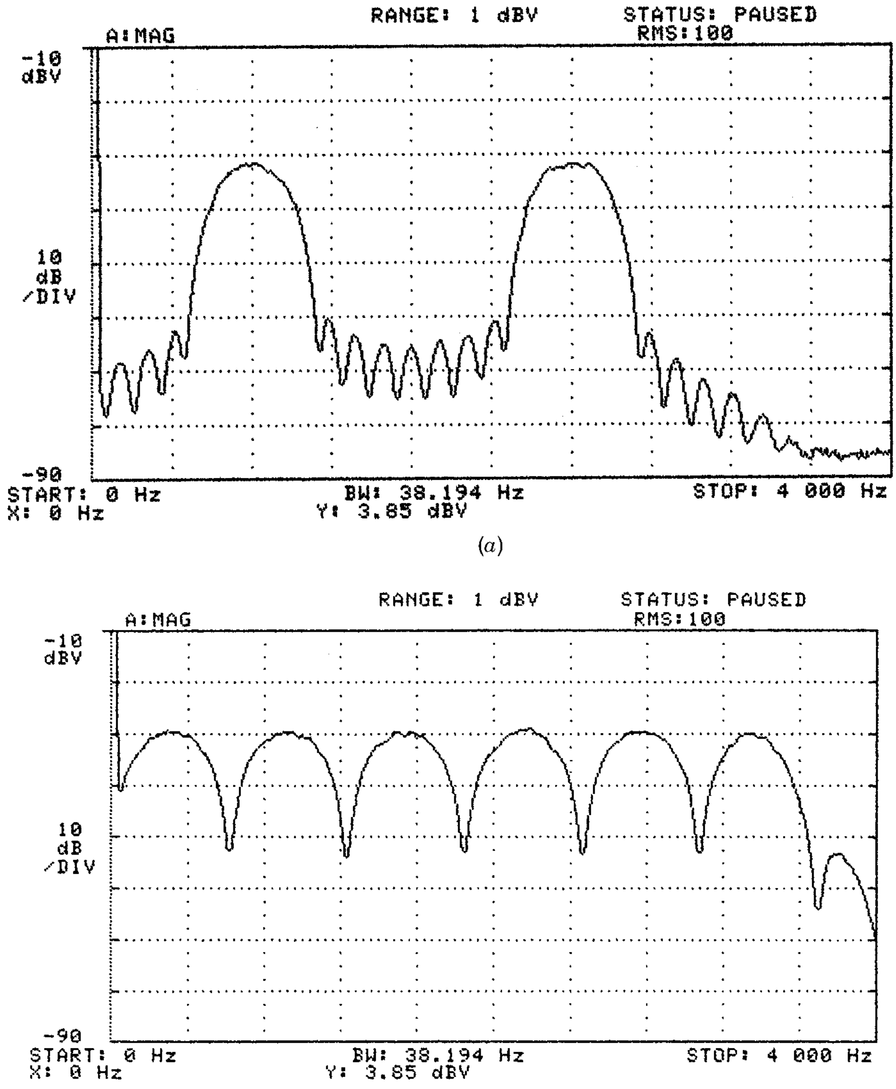

FIGURE 4.16. Output frequency responses obtained with HP analyzer: (*a*) FIR filter with two passbands; (*b*) FIR comb filter.

7. `Pass4b.cof`: with four passbands
8. `Stop3b.cof`: with three stopbands
9. `Comb14.cof`: with multiple notches (comb filter)

Figure 4.16*a* shows the real-time output frequency response of an FIR filter with two passbands, using the coefficient file *`pass2b.cof`*. This filter was designed with MATLAB. Figure 4.16*b* shows the frequency response of the comb filter, using the coefficients file *`comb14.cof`*. These plots were obtained with the HP 3561A signal analyzer.

Example 4.5: FIR Filter with Frequency Response Plot Using CCS (`FIRbuf`)

Figure 4.17 shows a listing of the program `FIRbuf.c`, which implements an FIR filter and stores the filter's output into a buffer. The FFT magnitude of the filter's output frequency response can then be plotted using CCS. Example 4.1 illustrated the implementation of an FIR filter using a generic program that includes the coefficient file representing the characteristics of a desired filter. Example 1.2 shows how one can store the output into a buffer so that it can be plotted within CCS. The program `FIRbuf.c` is based on these two earlier examples. The coefficient file `bp41.cof` represents a 41-coefficient FIR bandpass filter centered at 1 kHz. The output buffer has a size of 1024.

Build this project as **`FIRbuf`**. Verify that the output is a bandpass filter, centered at 1 kHz. Halt the processor.

With noise as input to the filter, the output frequency response can be plotted using CCS. The shareware utility Goldwave generates different signals, including noise, using a sound card (see Appendix E). The output from the sound card with the noise generated by Goldwave can be used as the input to the DSK.

Select View → Graph → Time/Frequency. Select/set for:

1. *Display type:* FFT magnitude
2. *Start address:* `yn_buffer`
3. *Acquisition buffer size:* 1024
4. *FFT frame size:* 1024
5. *FFT order:* 10
6. *DSP data type:* 16-bit signed integer
7. *Sampling rate:* 8000 Hz

Use default for the other fields. The FFT order is M, where 2^M = FFT frame size. Run the program and verify the output frequency response of the filter plotted in Figure 4.18.

Example 4.6: FIR Filter with Internally Generated Pseudorandom Noise as Input to Filter and Output Stored in Memory (`FIRPRNbuf`)

This example builds on Example 2.16, `noise_gen`, which generates a pseudorandom noise sequence, and Example 4.5, `FIRbuf`, which implements an FIR filter with the filter's output also stored in a memory buffer. Figure 4.19 shows a listing of the program `FIRPRNbuf.c`, which implements this project example.

The input to the filter is a software-generated noise sequence using `dly[0]` as the newest noise sequence. The coefficient file `BP41.cof`, which represents a 41-coefficient FIR bandpass filter, is the same as that used in Example 4.5.

```
//Firbuf.c FIR filter with output in buffer plotted with CCS

#include "bp41.cof"                   //BP @ 1 kHz coefficient file

int yn = 0;                           //initialize filter's output
short dly[N];                         //delay samples
short buffercount = 0;                //init buffer count
const short bufferlength = 1024;      //buffer size
short yn_buffer[1024];                //output buffer

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

      dly[0] = input_sample();        //newest input @ top of buffer
      yn = 0;                         //initialize filter's output
      for (i = 0; i< N; i++)
         yn +=(h[i]*dly[i]) >> 15;    //y(n)+=h(i)*x(n-i)
      for (i = N-1; i > 0; i--)       //start @ bottom of buffer
         dly[i] = dly[i-1];           //data move to update delays

      output_sample(yn);              //output filter

      yn_buffer[buffercount] = yn;    //filter's output into buffer
      buffercount++;                  //increment buffer count
      if(buffercount==bufferlength)   //if buffer count = size
         buffercount = 0;             //reinitialize buffer count
      return;                         //return from interrupt
}

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

FIGURE 4.17. FIR program with the filter output stored in memory (FIRbuf.c).

Build and run this project as **FIRPRNbuf**. Verify the output frequency response of a 1-kHz FIR bandpass filter. Goldwave can also be used as a crude spectrum analyzer to obtain the frequency response of the filter (with the output of the DSK connected to the input of the sound card).

Using CCS, verify the FFT magnitude plot as shown in Figure 4.20, using 1024 points. The address of the output buffer is *yn_buffer*. Figure 4.21 shows the frequency response of the FIR bandpass filter, centered at $F_s/8$, displayed using an HP analyzer.

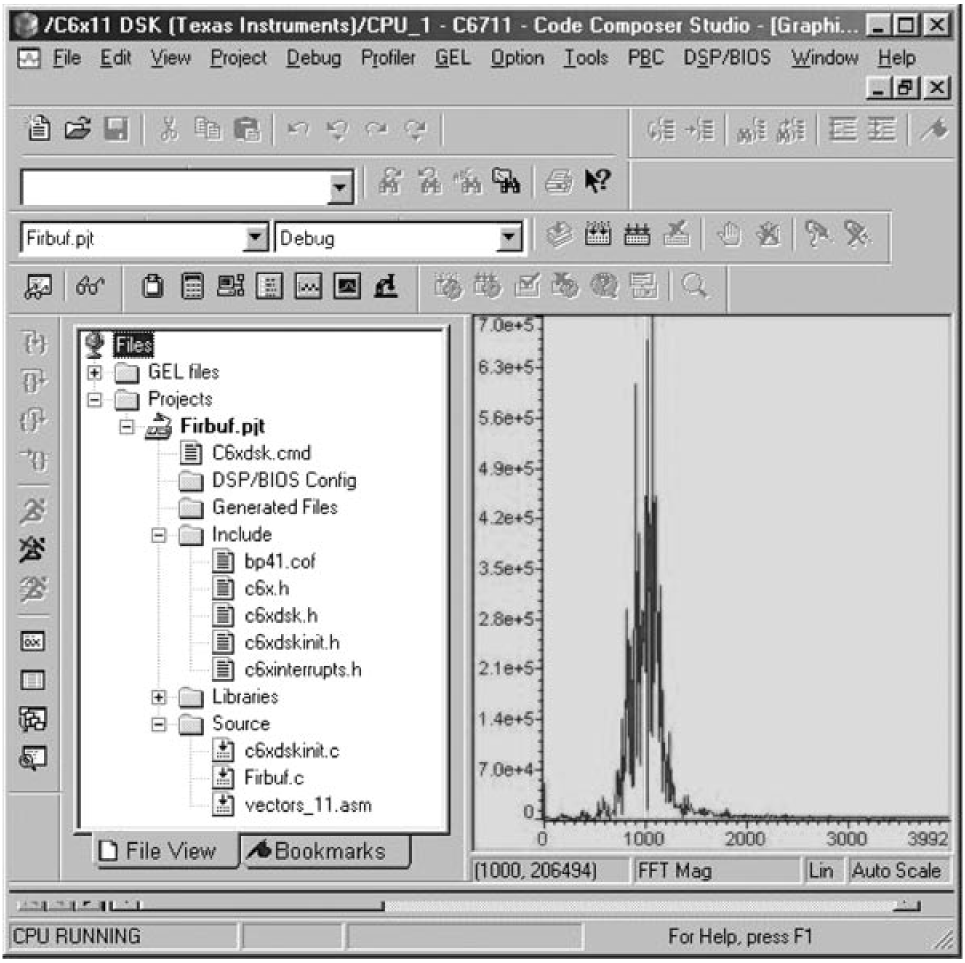

FIGURE 4.18. Output frequency response of a 1-kHz FIR bandpass filter plotted with CCS using external noise as input for project `FIRbuf`.

Change the output buffer so that the noise sequence is stored in memory using

```
yn_buffer[i] = dly[0];
```

Run the program again and plot the FFT magnitude of the noise sequence. It is not quite flat since the resulting plot is not averaged.

You can also output the noise sequence using

```
output_sample(dly[0]);
```

in the program. With the output to a spectrum analyzer with averaging capability, verify that the noise spectrum is quite flat until about 3500 Hz, the bandwidth of the antialiasing filter on the codec (looks like a lowpass filter with a bandwidth of

```
//FIRPRNbuf.c FIR filter with input noise sequence & output in buffer

#include "bp41.cof"                  //BP @ 1 kHz coefficient file
#include "noise_gen.h"               //header file for noise sequence
int yn = 0;                          //initialize filter's output
short dly[N];                        //delay samples
short buffercount = 0;               //init buffer count
const short bufferlength = 1024;     //buffer size
short yn_buffer[1024];               //output buffer
short fb;                            //feedback variable
shift_reg sreg;

short prn(void)                      //pseudorandom noise generation
{
 short prnseq;                       //for pseudorandom sequence

 if(sreg.bt.b0)                      //sequence {1,-1}
      prnseq = -8000;                //scaled negative noise level
 else
      prnseq = 8000;                 //scaled positive noise level
 fb =(sreg.bt.b0)^(sreg.bt.b1);      //XOR bits 0,1
 fb ^=(sreg.bt.b11)^(sreg.bt.b13);   //with bits 11,13 ->fb
 sreg.regval<<=1;                    //shift register 1 bit to left
 sreg.bt.b0 = fb;                    //close feedback path

  return prnseq;
}

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

      dly[0] = prn();                //input noise sequence
      yn = 0;                        //initialize filter's output
      for (i = 0; i< N; i++)
         yn +=(h[i]*dly[i]) >>15;    //y(n)+=h(i)*x(n-i)
      for (i = N-1; i > 0; i--)      //start @ bottom of buffer
         dly[i] = dly[i-1];          //data move to update delays

      output_sample(yn);             //output filter

      yn_buffer[buffercount] = yn;   //filter's output into buffer
      buffercount++;                 //increment buffer count
      if(buffercount==bufferlength)  //if buffer count = size
         buffercount = 0;            //reinitialize buffer count
      return;                        //return from interrupt
}

void main()
{
      sreg.regval = 0xFFFF;          //shift register to nominal values
      fb = 1;                        //initial feedback value
      comm_intr();                   //init DSK, codec, McBSP
      while(1);                      //infinite loop
}
```

FIGURE 4.19. FIR program with an input pseudorandom noise sequence and output stored in the memory buffer (FIRPRNbuf.c).

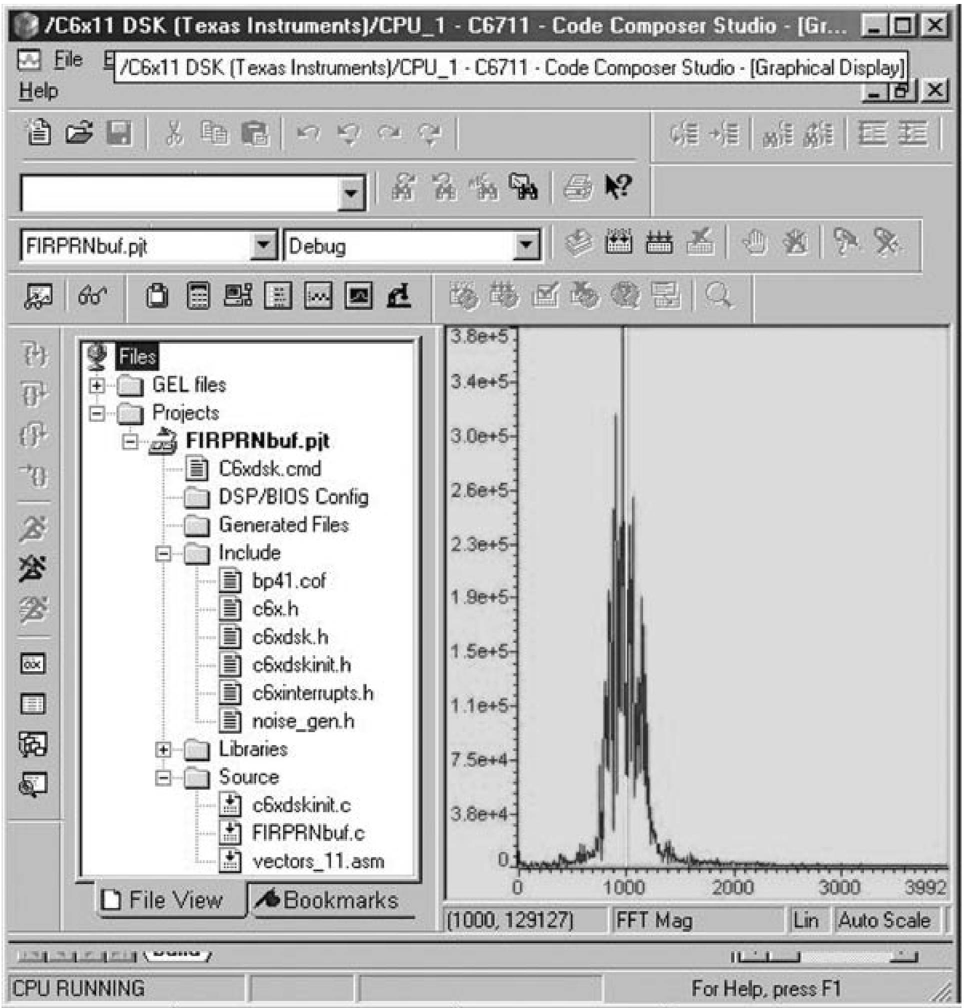

FIGURE 4.20. CCS output frequency response of a 1-kHz FIR bandpass filter using an internally generated noise sequence as input to the filter for project `FIRPRNbuf`.

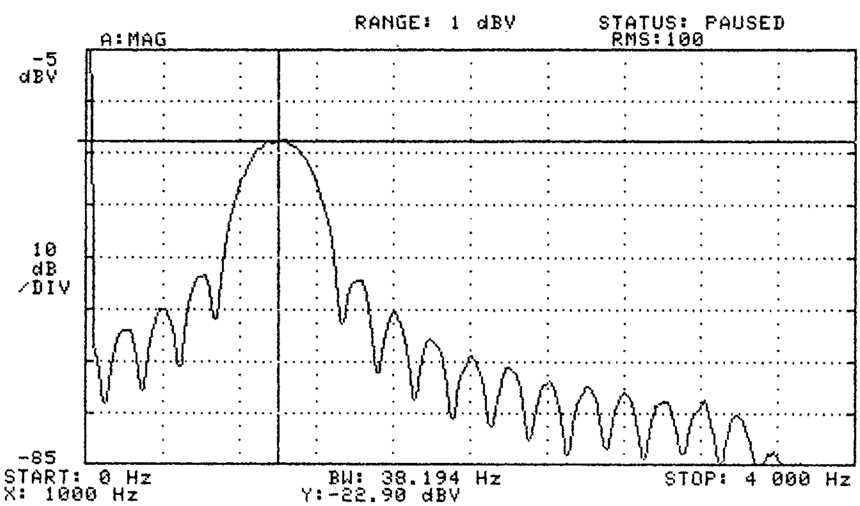

FIGURE 4.21. Frequency response of a 1-kHz FIR bandpass filter using an HP analyzer.

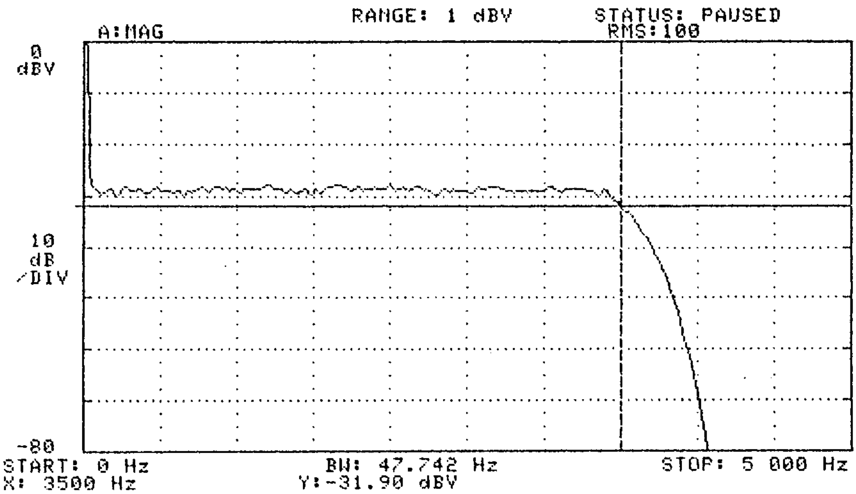

FIGURE 4.22. Spectrum of an internally generated pseudorandom noise sequence using an HP analyzer.

3500 Hz). Figure 4.22 shows the spectrum of this noise sequence using the HP analyzer (averaged with the analyzer). Use a GEL file to develop a slider so that the DSK output is either the noise sequence generated internally, `dly[0]`, or the filter's output `y(n)`.

Example 4.7: Two Notch Filters to Recover Corrupted Input Voice (NOTCH2)

This example illustrates the implementation of two notch (bandstop) FIR filters to remove two undesired sinusoidal signals corrupting an input voice signal. The voice signal (*TheForce.wav*, on the disk) was ADDed (using Goldwave) with the two undesired sinusoidal signals at frequencies of 900 Hz and 2700 Hz, to produce the corrupted input signal *corruptvoice.wav* (on the disk).

Figure 4.23 shows a listing of the program *NOTCH2.c*, which implements the two notch filters in cascade (series). Two coefficient files, `BS900.cof` and `BS2700.cof` (on the disk), each containing 89 coefficients and designed with MATLAB, are included in the filter program *NOTCH2.c*. They represent two FIR notch filters, centered at 900 Hz and 2700 Hz, respectively. A buffer is used for the delay samples of each filter. The output of the first notch filter, centered at 900 Hz, becomes the input to the second notch filter, centered at 2700 Hz.

Build this project as **NOTCH2**. Input (play) the corrupted voice signal *corruptvoice.wav*. Verify that the slider in position 1 (as set initially) outputs the corrupted voice signal, as shown in Figure 4.24. This plot is obtained with Goldwave using the DSK output as the input to a sound card (see Appendix E). The plot is shown on only one side (left channel) since a mono signal is used. Observe the two spikes (representing the two sinusoidal signals) at 900 Hz and 2700 Hz, respectively.

```
//Notch2.C Two FIR notch filters to remove two sinusoidal noise signals

#include "BS900.cof"                 //BS @ 900 Hz coefficient file
#include "BS2700.cof"                //BS @ 2700 Hz coefficient file
short dly1[N]={0};                   //delay samples for 1st filter
short dly2[N]={0};                   //delay samples for 2nd filter
int y1out = 0, y2out = 0;            //init output of each filter
short out_type = 1;                  //slider for output type

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

      dly1[0] = input_sample();      //newest input @ top of buffer
      y1out = 0;                     //init output of 1st filter
      y2out = 0;                     //init output of 2nd filter
      for (i = 0; i< N; i++)
         y1out += h900[i]*dly1[i];   //y1(n)+=h900(i)*x(n-i)

      dly2[0]=(y1out >>15);          //out of 1st filter->in 2nd filter
      for (i = 0; i< N; i++)
         y2out += h2700[i]*dly2[i];  //y2(n)+=h2700(i)*x(n-i)

      for (i = N-1; i > 0; i--)      //from bottom of buffer
        {
         dly1[i] = dly1[i-1];        //update samples of 1st buffer
         dly2[i] = dly2[i-1];        //update samples of 2nd buffer
        }

      if (out_type==1)               //if slider is in position 1
      output_sample(dly1[0]);        //corrupted input(voice+sines)
      if (out_type==2)
      output_sample(y2out>>15);      //output of 2nd filter (voice)
      return;                        //return from ISR
}

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

FIGURE 4.23. Program with two FIR notch filters in cascade to remove two undesired sinusoidal signals (NOTCH2.c).

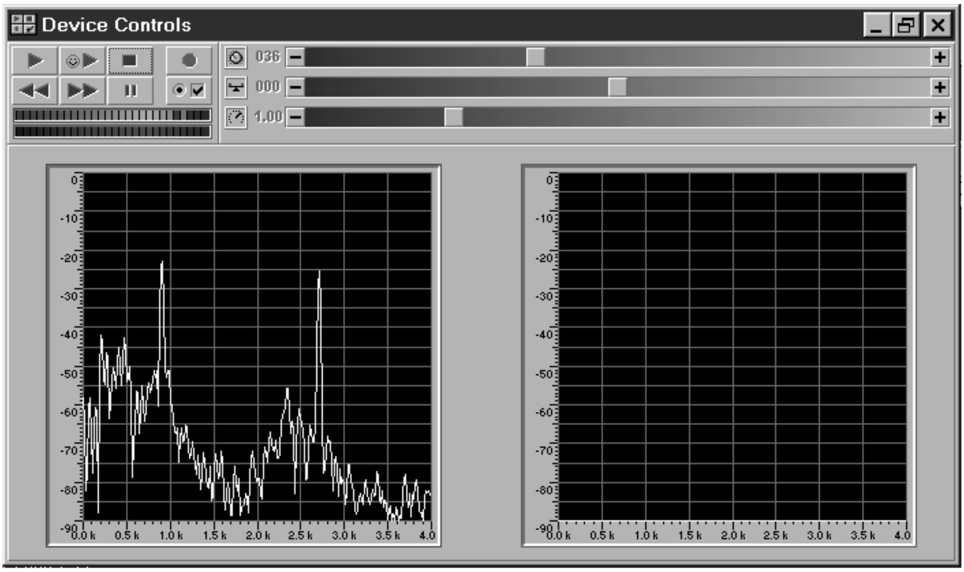

FIGURE 4.24. Spectrum of voice signal corrupted by two sinusoidal signals at 900 and 2700 Hz (obtained with Goldwave).

Change the slider to position 2 and verify that the two undesirable sinusoidal signals are removed.

Also output `y1out` through the function `output_sample` (rebuild) and verify that only the 2700-Hz corrupts the input voice signal.

Example 4.8: FIR Implementation Using Four Different Methods (`FIR4ways`)

Figure 4.25 shows a listing of the program `FIR4ways.c`, which implements an FIR filter using four alternative methods for convolving/updating the delay samples. This example extends Example 4.1, where the first method (method A) is used. In this first method with two "for" loops, the delay samples are arranged in memory with the newest sample at the beginning of the buffer and the oldest sample at the end of the buffer. The convolution starts with the newest sample and the first coefficient using

$$y(n) = h(0)x(n) + h(1)x(n-1) + \cdots + h(N-1)x(n-(N-1))$$

Each data value is "moved down" in memory to update the delay samples, with the newest sample being the newly acquired input sample. The size of the array for the delay samples is now set at $N+1$ and not N, to illustrate the third method (method C). The other three methods use a buffer size of N for the delay samples. The bottom (end) of the buffer in this example refers to memory location N, not $N+1$. Note

```
//FIR4ways.c FIR with alternative ways of storing/updating samples

#include "bp41.cof"                  //BP @ 1 kHz coefficient file
#define METHOD 'D'                   //change to B or C or D
int yn = 0;                          //initialize filter's output
short dly[N+1];                      //delay samples array(one extra)

interrupt void c_int11()             //ISR
{
      short i;
      yn = 0;                        //initialize filter's output

#if METHOD == 'A'                    //if 1st method
      dly[0] = input_sample();       //newest sample @ top of buffer
      for (i = 0; i< N; i++)
         yn += (h[i] * dly[i]);      //y(n)=h[0]*x[n]+..+h[N-1]x[n-(N-1)]
      for (i = N-1; i > 0; i--)      //from bottom of buffer
         dly[i] = dly[i-1];          //update sample data move "down"

#elif METHOD == 'B'                  //if 2nd method
      dly[0] = input_sample();       //newest sample @ top of buffer
      for (i = N-1; i >= 0; i--)     //start @ bottom to convolve
        {
         yn += (h[i] * dly[i]);      //y=h[N-1]x[n-(N-1)]+...+h[0]x[n]
         dly[i] = dly[i-1];          //update sample data move "down"
        }

#elif METHOD == 'C'                  //use xtra memory location
      dly[0] = input_sample();       //newest sample @ top of buffer
      for (i = N-1; i>=0; i--)       //start @ bottom of buffer
        {
         yn += (h[i] * dly[i]);      //y=h[N-1]x[n-(N-1)]+...+h[0]x[n]
         dly[i+1] = dly[i];          //update sample data move "down"
        }

#elif METHOD == 'D'                  //1st convolve before loop
      dly[N-1] = input_sample();     //newest sample @ bottom of buffer
      yn = h[N-1] * dly[0];          //y=h[N-1]x[n-(N-1)] (only one)
      for (i = 1; i<N; i++)          //convolve the rest
        {
         yn +=(h[N-(i+1)]*dly[i]);   //h[N-2]x[n-(N-2)]+...+h[0]x[n]
         dly[i-1] = dly[i];          //update sample data move "up"
        }
#endif
output_sample(yn >> 15);             //output filter
return;                              //return from ISR
}

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

FIGURE 4.25. FIR program using four alternative methods for convolution and updating of delay samples (FIR4ways.c).

that in this case the unused data $x(n-N)$ in memory location $(N+1)$ is not updated, by using the index $i < N$.

The second method (method B) performs the convolution and updates the delay samples using one loop. The convolution starts with the oldest coefficient and the oldest sample, "moving up" through the buffers using

$$y(n) = h(N-1)x(n-(N-1)) + h(N-2)x(n-(N-2)) + \cdots + h(0)x(n)$$

The updating scheme is similar to the first method. In method B, when $i = 0$, the newest sample is updated by an invalid data value residing at the memory location preceding the start of the sample buffer. But this invalid data item is then replaced by a newly acquired input sample with `dly[0]` before calculating $y(n)$ for the next unit of time. Or, one could use an "if" statement to update the delay samples for all values of i except for $i = 0$.

The third method uses $N + 1$ memory locations to update the delay samples. The unused data at memory location $N + 1$ is also updated. The extra memory location is used so that a valid data item in that location is not overwritten.

The fourth method performs the first convolution expression "outside" the loop. The delay samples in the previous methods were arranged in memory so that the newest sample, $x(n)$, is at the beginning of the buffer and the oldest sample, $x(n-(N-1))$, is at the end. However, in this method, the newest input sample is acquired through `dly[N - 1]` so that the newest sample is now at the end of the buffer and the updating process moves the data up.

Build and run this project as **`FIR4ways`**. Verify that the output is an FIR bandpass filter centered at 1 kHz. Change the method to test (define) the other three methods and verify that the resulting output is the same.

Example 4.9: Voice Scrambler Using Filtering and Modulation (`Scram16k`)

This example illustrates a voice scrambling/descrambling scheme. The approach makes use of basic algorithms for filtering and modulation. Modulation was introduced in Example 2.14. With voice as input, the resulting output is scrambled voice. The original unscrambled voice is recovered when the output of the DSK is used as the input to a second DSK running the same program.

An up-sampling scheme is used to process at a sampling rate of 16 kHz in lieu of the 8-kHz rate set with the AD535 codec. This results in a better performance, allowing for a wider input signal bandwidth.

The scrambling method used is commonly referred to as *frequency inversion*. It takes an audio range, represented by the band 0.3 to 3 kHz, and "folds" it about a carrier signal. The frequency inversion is achieved by multiplying (modulating) the audio input by a carrier signal, causing a shift in the frequency spectrum with upper and lower sidebands. On the lower sideband that represents the audible speech range, the low tones are high tones, and vice versa.

Figure 4.26 is a block diagram of the scrambling scheme. At point A we have a

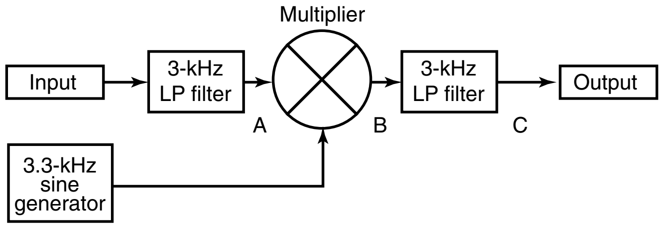

FIGURE 4.26. Block diagram of scrambler/descrambler scheme.

bandlimited signal 0 to 3 kHz. At point B we have a double-sideband signal with suppressed carrier. At point C the upper sideband is filtered out. Its attractiveness comes from its simplicity, since only simple DSP algorithms are utilized: filtering, sine generation/modulation, and up-sampling (due to low sampling frequency with the AD535 codec).

Figure 4.27 shows a listing of the program `Scram16k.c`, which implements this project. The input signal is first lowpass filtered and the resulting output (at point A) is multiplied (modulated) by a 3.3-kHz sine function with data values in a buffer (lookup table). The modulated signal (at point B) is filtered again, and the overall output is a scrambled signal (at point C).

There are three functions in Figure 4.27 in addition to the function *`main`*. One of the functions, *filtmodfilt*, calls a filter function to implement the first lowpass filter as an antialiasing filter. The resulting output (filtered input) becomes the input to a multiplier/modulator. The function *`sinemod`* modulates (multiplies) the filtered input with the 3.3-kHz sine data values. This produces higher and lower sideband components. The modulated output is again filtered, so that only the lower sideband components are kept.

The up-sampling scheme to obtain a 16-kHz sampling rate is achieved by "processing" the data twice and retaining only the second result. This allows for a wider input signal bandwidth to be scrambled.

A buffer is used to store the 114 coefficients that represent the lowpass filter. The coefficient file *`lp114.cof`* is on disk. Two other buffers are used for the delay samples, one for each filter. The samples are arranged in memory as

$$x(n-(N-1)), x(n-(N-2)), \ldots, x(n-1), x(n)$$

with the oldest sample at the beginning of the buffer and the newest sample at the end (bottom) of the buffer. The file *`sine160.h`* with 160 data values over 33 cycles is on disk. The frequency generated is $f = F_s$ (number of cycles)/(number of points) = 16,000(33)/160 = 3.3 kHz.

Using the resulting output as the input to a second DSK running the same algorithm, the original unscrambled input is recovered as the output of the second DSK. Note that the program can still run on the first DSK when it is disconnected from the parallel port cable (DB25 cable).

Build and run this project as **`Scram16k`**. First test this project using a 2-kHz input sine wave. The resulting output is a lower sideband signal of 1.3 kHz, obtained as

```
//Scram16k.c Voice scrambler/de-scrambler program

#include "sine160.h"                  //sine data values
#include "LP114.cof"                  //filter coefficient file
short filtmodfilt(short data);
short filter(short inp,short *dly);
short sinemod(short input);
static short filter1[N],filter2[N];
short input, output;

void main()
{
 short i;

 comm_poll();                      //init DSK using polling
 for (i=0; i< N; i++)
   {
   filter1[i] = 0;                 //init 1st filter buffer
   filter2[i] = 0;                 //init 2nd filter buffer
   }
 while(1)
   {
   input=input_sample();           //input new sample data
   filtmodfilt(input);             //process sample twice(upsample)
   output=filtmodfilt(input);      //and throw away 1st result
   output_sample(output);          //then output
   }
}

short filtmodfilt(short data)      //filtering & modulating
{
 data = filter(data,filter1);      //newest in ->1st filter
 data = sinemod(data);             //modulate with 1st filter out
 data = filter(data,filter2);      //2nd LP filter
 return data;
}

short filter(short inp,short *dly)    //implements FIR
{
 short i;
 int yn;

 dly[N-1] = inp;                   //newest sample @bottom buffer
 yn = dly[0] * h[N-1];             //y(0)=x(n-(N-1))*h(N-1)
 for (i = 1; i < N; i++)           //loop for the rest
   {
   yn += dly[i] * h[N-(i+1)];      //y(n)=x[n-(N-1-i)]*h[N-1-i]
   dly[i-1] = dly[i];              //data up to update delays
   }
 yn = ((yn) >>15);                 //filter's output
 return yn;                        //return y(n) at time n
}

short sinemod(short input)         //sine generation/modulation
{
 static short i=0;
 input=(input*sine160[i++])>>11;   //(input)*(sine data)
 if(i>= NSINE) i = 0;              //if end of sine table
 return input;                     //return modulated signal
}
```

FIGURE 4.27. Voice scrambler program (`Scram16k.c`).

(3.3 kHz − 2 kHz). The upper sideband signal of (3.3 kHz + 2 kHz) is filtered out by the second lowpass filter.

A second DSK is used to recover/unscramble the original signal (simulating the receiving end). Use the output of the first DSK as the input to the second DSK. Run the same program on the second DSK. This produces the reverse procedure, yielding the original unscrambled signal. If the same 2-kHz original input is considered, the 1.3 kHz as the scrambled signal becomes the input to the second DSK. The resulting output is the original signal of 2 kHz (3.3 kHz − 1.3 kHz), the lower sideband signal.

With a sweeping input sinusoidal signal increasing in frequency, the resulting output is the sweeping signal "decreasing" in frequency. Use as input the `.wav` file *`TheForce.wav`* and verify the scrambling/descrambling scheme.

Interception of the speech signal can be made more difficult by changing the modulation frequency dynamically and including (or omitting) the carrier frequency according to a predefined sequence: for example, a code for no modulation, another for modulating at frequency f_{c1}, and a third code for modulating at frequency f_{c2}.

This project was first implemented using the TMS320C25 [49] and also on the TMS320C31 DSK without the need for up-sampling.

Example 4.10: Illustration of Aliasing Effects with Down-Sampling (`aliasing`)

Figure 4.28 shows a listing of the program `aliasing.c`, which implements this project. To illustrate the effects of aliasing, the processing rate is down-sampled by a factor of 2, to an equivalent 4-kHz rate. Note that the antialiasing and reconstruction filters on the AD535 codec are fixed and connot be bypassed or altered. Up-sampling and lowpass filtering are needed to output the 4-kHz rate samples to the AD535 codec sampling at 8 kHz.

Build this project as **`aliasing`**. Load the slider file `aliasing.gel` (on the disk). With antialiasing initially set to zero in the program, aliasing will occur.

1. Input a sinusoidal signal and verify that for an input signal frequency up to 2 kHz, the output is essentially a loop program (delayed input). Increase the input signal frequency to 2.5 kHz and verify that the output is an aliased 1.5-kHz signal. Similarly, a 3- and a 3.5-kHz input signal yield an aliased output signal of 1 and 0.5 kHz, respectively. Input signals with frequencies beyond 3.5 kHz are supressed due to the AD535 codec's antialiasing filter.
2. Change the slider position to 1, so that antialiasing at the down-sampled rate of 4 kHz is desired. For an input signal frequency up to about 1.8 kHz, the output is a delayed version of the input. Increase the input signal frequency beyond 1.8 kHz and verify that the output reduces to zero. This is due to the 1.8-kHz antialiasing lowpass filter, implemented using the coefficient file `lp33.cof` (on the disk).

```
//Aliasing.c illustration of downsampling, aliasing, upsampling

#include "lp33.cof"                  //lowpass at 1.8 kHz
short flag = 0;                      //toggles for 2x down-sampling
float indly[N],outdly[N];            //antialias and reconst delay lines
short i;                             //index
float yn;                            //filter output
short antialiasing = 0;              //init for no antialiasing filter

interrupt void c_int11()             //ISR
{

 indly[0]=(float)(input_sample());   //new sample to antialias filter
 yn = 0.0;                           //initialize downsampled value
 if (flag == 0)                      //discard input sample value
     flag = 1;                       //don't discard at next sampling
 else
  {
   if (antialiasing == 1)            //if antialiasing filter desired
    {                                //compute downsampled value
     for (i = 0 ; i < N ; i++)       //using LP @ 1.8 kHz filter coeffs
        yn += (h[i]*indly[i]);       //filter is implemented using float
    }
   else                              //if filter is bypassed
        yn = indly[0];               //downsampled value is input value
   flag = 0;                         //next input value will be discarded
  }
  for (i = N-1; i > 0; i--)
        indly[i] = indly[i-1];       //update input buffer

  outdly[0] = (yn);                  //input to reconst filter
  yn = 0.0;                          //4 kHz sample values and zeros
  for (i = 0 ; i < N ; i++)          //are filtered at 8 kHz rate
        yn += (h[i]*outdly[i]);      //by reconstruction lowpass filter

  for (i = N-1; i > 0; i--)
        outdly[i] = outdly[i-1];     //update delays

  output_sample((short)(yn));        //8 kHz rate sample
  return;                            //return from interrupt
}

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

FIGURE 4.28. Program to illustrate aliasing and antialiasing down-sampling to a rate of 4 kHz (aliasing.c).

In lieu of using a sinusoidal signal as input, you can play sweep.wav from Goldwave (see Appendix E).

Example 4.11: Implementation of an Inverse FIR Filter (FIRinverse)

Figure 4.29 shows a listing of the program *FIRinverse.c*, which implements an inverse FIR filter. An original input sequence to an FIR filter can be recovered using an inverse FIR filter. The transfer function of an FIR filter of order *N* is

```
//FIRinverse.c Implementation of inverse FIR Filter

#include "bp41.cof"          //coefficient file BP @ Fs/8
int yn;                      //filter's output
short dly[N];                //delay samples
int out_type = 1;            //output type for slider

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

 dly[0] = input_sample();    //newest input sample data
 yn = 0;                     //initialize filter's output

 for (i = 0; i<N; i++)
    yn += (h[i]*dly[i]);     //y(n)+=h(i)*x(n-i)
 if(out_type==1)             //if slider in position 1
     output_sample(dly[0]);  //original input
 if(out_type==2)
     output_sample(yn>>15);  //output of FIR filter
 if(out_type==3)             //calculate inverse FIR
   {
    for (i = N-1; i>1; i--)
       yn -= (h[i]*dly[i]);  //calculate inverse FIR filter
    yn = yn/h[0];            //scale output of inverse filter
    output_sample(yn>>8);    //send output of inverse filter
   }
 for (i = N-1; i>0; i--)     //from bottom of buffer
    dly[i] = dly[i-1];       //update delay samples
 return;                     //return from ISR
}

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

FIGURE 4.29. Program to implement an inverse FIR filter (FIRinverse.c).

$$H(z) = \sum_{i=0}^{N-1} h_i z^{-i}$$

where h_i represents the impulse response coefficients. The output sequence of the FIR filter is

$$y(n) = \sum_{i=0}^{N-1} h_i x(n-i) = h_0 x(n) + h_1 x(n-1) + \cdots + h_{N-1} x(n-(N-1))$$

where $x(n - i)$ represents the input sequence. The original input sequence, x, can then be recovered, using $\hat{x}(n)$ as an estimate of $x(n)$, or

$$\hat{x}(n) = \frac{y(n) - \sum_{i=1}^{N-1} h_i \hat{x}(n-i)}{h_0}$$

Build this project as **FIRinverse**. Use noise as input (from Goldwave or from a noise generator, or modify the program to use the pseudorandom noise sequence, etc.). Verify that the output is the input noise sequence, with the slider in position 1 (default). Change the slider to position 2 and verify the output as an FIR bandpass filter centered at 1 kHz. With the slider in position 3, the inverse of the FIR filter is calculated, so that the output is the original input noise sequence.

Example 4.12: FIR Implementation Using C Calling ASM Function (FIRcasm)

The C program *FIRcasm.c* (Figure 4.30) calls the ASM function *FIRcasm-func.asm* (Figure 4.31), which implements an FIR filter.

Build and run this project as **FIRcasm**. Verify that the output is a 1-kHz FIR bandpass filter. Two buffers are created: *dly* for the data samples and *h* for the filter's coefficients. On each interrupt, a new data sample is acquired and stored at the end (higher-memory address) of the buffer *dly*. The delay samples and the filter coefficients are arranged in memory as shown in Table 4.3. The delay samples are stored in memory starting with the oldest sample with the newest sample at the end of the buffer. The coefficients are arranged in memory with h(0) at the beginning of the coefficient buffer and $h(N - 1)$ at the end.

The addresses of the delay sample buffer, the filter coefficient buffer, and the size of each buffer are passed to the ASM function through registers A4, B4, and A6, respectively. The size of each buffer through register A6 is doubled since data in each memory location are stored as byte. The pointers A4 and B4 are incremented or decremented every two bytes (two memory locations). The end address of the coefficients' buffer is in B4, which is at $2N - 1$.

```
//FIRCASM.c FIR C program calling ASM function fircasmfunc.asm

#include "bp41.cof"                //BP @ Fs/8 coefficient file
int yn = 0;                        //initialize filter's output
short dly[N];                      //delay samples

interrupt void c_int11()           //ISR
{
      dly[N-1] = input_sample();   //newest sample @bottom buffer
      yn = fircasmfunc(dly,h,N);   //to ASM func through A4,B4,A6
      output_sample(yn >> 15);     //filter's output
      return;                      //return from ISR
}

void main()
{
      short i;

      for (i = 0; i<N; i++)
      dly[i] = 0;                  //init buffer for delays
      comm_intr();                 //init DSK, codec, McBSP
      while(1);                    //infinite loop
}
```

FIGURE 4.30. C program calling an ASM function for FIR implementation (`FIRcasm.c`).

TABLE 4.3 Memory Organization of Coefficients and Samples for FIRcasm

Coefficients	Samples: Time n		Samples: Time n + 1
h(0)	A4 → x(n - (N - 1))		A4 → x(n - (N - 2))
h(1)	x(n - (N - 2))		x(n - (N - 3))
h(2)	x(n - (N - 3))		x(n - (N - 4))
.	.		.
.	.		.
.	.		.
h(N - 2)	x(n - 1)		x(n)
B4 → h(N - 1)	x(n)	← newest →	x(n + 1)

The two `LDH` instructions load the content in memory pointed at (whose address is specified by) A4 and the content in memory at the address specified in B4. This loads the oldest samples, $x(n - (N - 1))$ and $h(N - 1)$, respectively. A4 is then postincremented to point at $x(n - (N - 2))$, and B4 is postdecremented to point at $h(N - 2)$. After the first accumulation, the oldest sample is updated. The content in memory at the address specified by A4 is loaded into A7, then stored at the preceding memory location. This is because A4 is postdecremented *without* modifica-

```
;FIRCASMfunc.asm ASM function called from C to implement FIR
;A4 = Samples address, B4 = coeff address, A6 = filter order
;Delays organized as:x(n-(N-1))...x(n);coeff as h[0]...h[N-1]

            .def    _fircasmfunc
_fircasmfunc:                       ;ASM function called from C
            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 byte
            SUB     B4,1,B4         ;B4=bottom coeff array h[N-1]
loop:                               ;start of FIR loop
            LDH     *A4++,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
            NOP     4
            MPY     A2,B2,A6        ;A6=x[n-(N-1)+i]*h[N-1-i]
            NOP
            ADD     A6,A8,A8        ;accumlate in A8

            LDH     *A4,A7          ;A7=x[(n-(N-1)+i+1]update delays
            NOP     4               ;using data move "up"
            STH     A7,*-A4[1]      ;-->x[(n-(N-1)+i] update sample
            SUB     A1,1,A1         ;decrement loop count
     [A1]   B       loop            ;branch to loop if count # 0
            NOP     5

            MV      A8,A4           ;result returned in A4
            B       B3              ;return addr to calling routine
            NOP     5
```

FIGURE 4.31. FIR ASM function called from C (FIRcasmfunc.asm).

tion to point at the memory location containing the oldest sample. As a result, the oldest sample, $x(n-(N-1))$, is replaced (updated) by $x(n-(N-2))$. The updating of the delay samples is for the next unit of time. As the output at time n is being calculated, the samples are updated or "primed" for time $(n + 1)$. At time n the filter's output is

$$y(n) = h(N-1)x(n-(N-1)) + h(N-2)x(n-(N-2)) + \cdots + h(1)x(n-1) + h(0)x(n)$$

The loop is processed 41 times. For each time n, $n + 1$, and $n + 2$ an output value is calculated, with each sample updated for the next unit of time. The newest sample is also updated in this process, with an invalid data value residing at the memory location beyond the end of the buffer. But this is remedied since for each unit of time, the newest sample, acquired through the ADC of the codec, overwrites it.

Accumulation is in A8 and the result, for each unit of time, is moved to A4 to be returned to the calling function. The address of the calling function is in B3.

Viewing Update of Samples in Memory

1. *Select → View → Memory using a 16-bit hex format and a starting address of* `dly`. The delay samples are within 82 (not 41) memory locations, each location specified with a byte. The coefficients also occupy 82 memory locations, in the buffer `h`. You can verify the content in the coefficient buffer stored as a 16-bit or half-word value. Right-click on the memory window and deselect "Float in Main Window" for a better display with both source program and memory.
2. *Select → View → Mixed C/ASM.* Place a breakpoint within the function `FIRcasmfunc.asm` at the move instruction

   ```
   MV    A8,A4
   ```

   (you can either double-click on that line of code, or right-mouse-click to Toggle Breakpoint).
3. *Select → Debug → Animate (introduced in Chapter 1).* Execution halts at the set breakpoint for each unit of time. Observe the bottom memory location of the delay samples. Verify that the newest sample data value is placed at the end of the buffer. This value is then moved up the buffer. Observe after a while that the samples are being updated, with each value in the buffer moving up in memory. You can also observe the register (pointer) A4 incrementing by 2 (two bytes) and B4 decrementing by 2.

Example 4.13: FIR Implementation Using C Calling Faster ASM Function (`FIRcasmfast`)

The same C calling program, `FIRcasm.c`, is used in this example as in Example 4.12. It calls the ASM function *`Fircasmfunc.asm`* (Figure 4.32) within the file *`FIRcasmfuncfast`* (not *`FIRcasmfunc`*).

This function executes faster than the function in Example 4.12 by having parallel instructions and rearranging the sequence of instructions. There are two parallel instructions: `LDH/LDH` and `SUB/LDH`.

1. The number of `NOP`s is reduced from 19 to 11.
2. The `SUB` instruction to decrement the loop count is moved up the program.
3. The sequence of some instructions changed to "fill" some of the `NOP` slots.

For example, the conditional branch instruction executes *after* the `ADD` instruction to accumulate in A8, since branching has five delay slots. Additional changes

```
;FIRCASMfuncfast.asm C-called faster function to implement FIR
            .def   _fircasmfunc
_fircasmfunc:                     ;ASM function called from C
            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 byte
            SUB    B4,1,B4        ;B4=bottom coeff array h[N-1]
loop:                             ;start of FIR loop
            LDH    *A4++,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 loop count
   ||       LDH    *A4,A7         ;A7=x[(n-(N-1)+i+1]update delays
            NOP    4
            STH    A7,*-A4[1]     ;-->x[(n-(N-1)+i] update sample
   [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       ;accumlate in A8

            B      B3             ;return addr to calling routine
            MV     A8,A4          ;result returned in A4
            NOP    4
```

FIGURE 4.32. ASM function called from C for faster execution (`FIRcasmfuncfast.asm`).

to make it faster would also make it less comprehensible, due to further resequencing of the instructions.

Build this project as **`FIRcasmfast`**, so that the linker option names the output executable file `FIRcasmfast.out`. The resulting output is the 1-kHz bandpass filter in Example 4.12.

Example 4.14: FIR Implementation with C Program Calling ASM Function Using Circular Buffer (`FIRcirc`)

The C program *`FIRcirc.c`* (Figure 4.33) calls the ASM function *`FIRcircfunc.asm`* (Figure 4.34), which implements an FIR filter using a circular buffer. This example expands Example 4.13. The coefficients within the file *`bp1750.cof`* were designed with MATLAB using the Kaiser window and represent a 128-coefficient FIR bandpass filter with a center frequency of 1750 Hz. Figure 4.35 displays the characteristics of this filter, obtained from MATLAB's filter designer SPTOOL (described in Appendix D).

```
//FIRcirc.c C program calling ASM function using circular buffer

#include "bp1750.cof"               //BP at 1750 Hz coeff file
int yn = 0;                         //init filter's output

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

 sample_data = input_sample();      //newest input sample data
 yn = fircircfunc(sample_data,h,N); //ASM func passing to A4,B4,A6
 output_sample(yn >> 15);           //filter's output
 return;                            //return to calling function
}

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

FIGURE 4.33. C program calling an ASM function using a circular buffer (`FIRcirc.c`).

In lieu of moving the data to update the delay samples, a pointer is used. The 16 LSBs of the address mode register (AMR) are set with a value of

```
0x0040 = 0000 0000 0100 0000
```

This selects A7 mode as the circular buffer pointer register. The 16 MSBs of AMR are set with `N = 0x0007` to select the block BK0 as a circular buffer. The buffer size is $2^{N+1} = 256$. A circular buffer is used in this example only for the delay samples. It is also possible to use a second circular buffer for the coefficients. For example, using

```
0x0140 = 0000 0001 0100 0000
```

would select two pointers, B4 and A7.

Within a C program, an inline assembly code can be used with the `asm` statement. For example,

```
asm(" MVK 0x0040,B6")
```

Note the blank space after the first quote so that the instruction does not start in column 1. The circular mode of addressing eliminates the data move to update the delay samples, since the pointer can be moved to achieve the same result faster.

```
;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 register AMR

          MVK    last_addr,A9   ;A9=last circ addr(lower 16 bits)
          MVKH   last_addr,A9   ;last circ addr (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
```

FIGURE 4.34. C-called ASM function using a circular buffer to update samples (`FIRcircfunc.asm`).

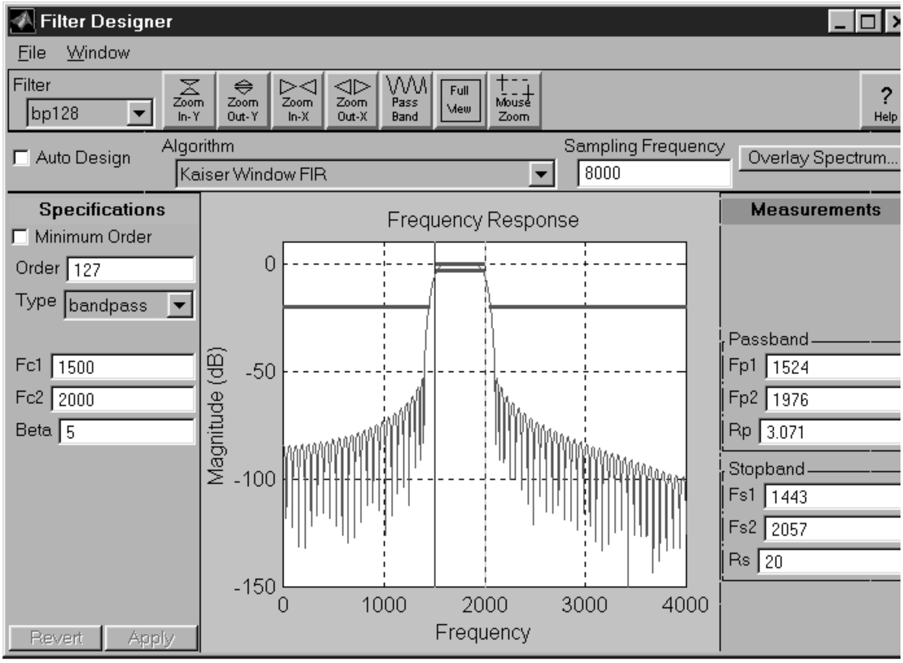

FIGURE 4.35. Frequency characteristics of a 128-coefficient FIR bandpass filter centered at 1750 Hz using MATLAB's filter designer SPTOOL.

Initially, the register pointer A7 points to the last address in the sample buffer. Consider for now the sample buffer only, since it is circular.

1. *Time n*. At time n, A7 points to the end of the buffer, where the newest sample is stored. It is then postincremented to point to the beginning of the buffer, as shown in Table 4.4. Then the section of code within the loop starts, and calculates

$$y(n) = h(N-1)x(n-(N-1)) + h(N-2)x(n-(N-2)) + \cdots \\ + h(1)x(n-1) + h(0)x(n)$$

   After the last multiplication, $h(0)x(n)$, A7 is postincremented to point to the beginning address of the buffer. The resulting filter's output at time n is then returned to the calling function. Before the loop starts for each unit of time, A7 always contains the address where the newest sample is to be stored. While the newly acquired sample is passed to the ASM function through A4 at each unit of time $n, n+1, n+2, \ldots$, A4 is stored in A7, which always contains the last address.

2. *Time n + 1*. At time $(n+1)$, the newest sample, $x(n+1)$, is passed to the ASM function through A4. The STH instruction stores that sample into memory

whose address is in A7, which is at the beginning of the buffer. It is then post-incremented to point at the address containing $x(n-(N-2))$, as shown in Table 4.4. The output is now

$$y(n+1) = h(N-1)x(n-(N-2)) + h(N-2)x(n-(N-3)) + \cdots$$
$$+ h(1)x(n) + h(0)x(n+1)$$

The last multiplication always involves `h(0)` and the newest sample.

3. *Time* $n+2$. At time $(n+2)$, the filter's output is

$$y(n+2) = h(N-1)x(n-(N-3)) + h(N-2)x(n-(N-4)) + \cdots$$
$$+ h(1)x(n+1) + h(0)x(n+2)$$

Note that for each unit of time, the newly acquired sample overwrites the oldest sample at the previous unit of time. At each time $n, n+1, \ldots$, the filter's output is calculated within the ASM function, and the result is sent to the calling C function, where a new sample is acquired at each sample period.

The conditional branch instruction was moved up as in Example 4.13. Branching to loop takes effect (due to five delay slots) after the `ADD` instruction to accumulate in A8. One can save the content of AMR at the end of processing one buffer and restore it before using it again with a pair of `MVC` instructions: `MVC AMR,Bx` and `MVC Bx,AMR` using a B register.

Build and run this project as **`FIRcirc`**. Verify an FIR bandpass filter centered at 1750 Hz. Halt, reset, and reload the program.

Place a breakpoint within the ASM function `FIRcircfunc.asm` at the branch instruction to return to the calling C function. View memory at the address `delays` and verify that this buffer of size 256 is initialized to zero. Right-click on the memory

TABLE 4.4 Memory Organization of Coefficients and Samples Using Circular Buffer

	Samples		
Coefficients	Time `n`	Time `n + 1`	Time `n + 2`
`h(0)`	A7 → `x(n - (N - 1))`	newest → `x(n + 1)`	`x(n + 1)`
`h(1)`	`x(n - (N - 2))`	A7 → `x(n - (N - 2))`	newest → `x(n + 2)`
`h(2)`	`x(n - (N - 3))`	`x(n - (N - 3))`	A7 → `x(n - (N - 3))`
.	.	.	.
.	.	.	.
.	.	.	.
`h(N - 2)`	`x(n - 1)`	`x(n - 1)`	`x(n - 1)`
`h(N - 1)`	newest → `x(n)`	`x(n)`	`x(n)`

window to toggle "Float in Main Window" (for a better display). Run the program. Execution stops at the breakpoint. Verify that the newest sample (16 bits) is stored at the end (higher address) of the buffer (at 0x3FE and 0x3FF). Memory location 0x400 contains the last address 0x301 where the subsequent sample is to be stored. This address is the beginning of the buffer. View the core registers and verify that A7 contains that address.

Run again and observe the new sample stored at the beginning of the buffer (you can animate now). Note that A7 is incremented to 0x303, 0x305, ... The circular method of updating the delays is more efficient. It is important that the buffer is aligned on a boundary of a power of 2.

Example 4.15: FIR Implementation with C Program Calling ASM Function Using Circular Buffer in External Memory (FIRcirc_ext)

This example implements an FIR filter using a circular buffer in external memory. It expands slightly on Example 4.14. The C program FIRcirc.c in Example 4.14 is modified to obtain *FIRcirc_ext.c* (Figure 4.36) so that it calls the ASM function FIRcircfunc_ext.asm (in lieu of FIRcircfunc.asm).

The linker command file *FIRcirc_ext.cmd* used in this example is listed in

```
//FIRcirc_ext.c C program calling ASM function using circular buffer

#include "bp1750.cof"                     //BP at 1750 Hz coeff file
int yn = 0;                               //init filter's output

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

 sample_data = input_sample();            //newest input sample data
 yn = fircircfunc_ext(sample_data,h,N);  //ASM funcn passing to A4,B4,A6
 output_sample(yn >> 15);                 //filter's output
 return;                                  //return to calling function
}

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

FIGURE 4.36. C program calling an ASM function with a circular buffer in external memory (FIRcirc_ext.c).

```
;FIRcircfunc_ext.asm Function using circular buffer in external memory
;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_ext
           .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
           .text                   ;code section
_fircircfunc_ext:                   ;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 register AMR

           MVKL   last_addr,A9  ;A9=bottom circ addr in external mem
           MVKH   last_addr,A9  ;(higher 16 bits)in external circ
           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
```

FIGURE 4.37. C-called ASM function with a circular buffer in external memory (FIRcircfunc_ext.asm).

```
/*FIRcirc_ext.cmd Linker file for circular buffer in external memory*/

MEMORY
{
  VECS:       org =         0h, len =      0x220
  IRAM:       org = 0x00000220, len = 0x0000FDC0
  buffer_ext: org = 0x80000000, len = 0x00000110
  SDRAM:      org = 0x80000110, len = 0x01000000
  FLASH:      org = 0x90000000, len = 0x00020000
}
SECTIONS
{
  circdata :> buffer_ext
  vectors  :> VECS
  .text    :> IRAM
  .bss     :> IRAM
  .cinit   :> IRAM
  .stack   :> IRAM
  .sysmem  :> SDRAM
  .const   :> IRAM
  .switch  :> IRAM
  .far     :> SDRAM
  .cio     :> SDRAM
}
```

FIGURE 4.38. Linker command file for a circular buffer in external memory (`FIRcirc_ext.cmd`).

Figure 4.38. The section *`circdata`* designates the memory section `buffer_ext`, which starts in external memory at `0x80000000`.

Build this project as **`FIRcirc_ext`**. View the memory at the address `delays`. This should display the external memory section. Verify the circular buffer in external memory. Place a breakpoint as in Example 4.14, animate, and verify that the newest sample is stored at the end of the circular buffer and that the subsequent acquired sample is stored at the beginning of the buffer. Halt, remove the breakpoint, and verify that the output is an FIR bandpass filter centered at 1750 Hz.

REFERENCES

1. W. J. Gomes III and R. Chassaing, Filter design and implementation using the TMS320C6x interfaced with MATLAB, *Proceedings of the 2000 ASEE Annual Conference*, 2000.
2. A. V. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1989.

3. B. Gold and C. M. Rader, *Digital Signal Processing of Signals*, McGraw-Hill, New York, 1969.
4. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1975.
5. T. W. Parks and J. H. McClellan, Chebychev approximation for nonrecursive digital filter with linear phase, *IEEE Transactions on Circuit Theory*, Vol. CT-19, 1972, pp. 189–194.
6. J. H. McClellan and T. W. Parks, A unified approach to the design of optimum linear phase digital filters, *IEEE Transactions on Circuit Theory*, Vol. CT-20, 1973, pp. 697–701.
7. J. F. Kaiser, Nonrecursive digital filter design using the I0-sinh window function, *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1974.
8. J. F. Kaiser, Some practical considerations in the realization of linear digital filters, *Proceedings of the 3rd Allerton Conference on Circuit System Theory*, Oct. 1965, pp. 621–633.
9. L. B. Jackson, *Digital Filters and Signal Processing*, Kluwer Academic, Norwell, MA, 1996.
10. J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, Prentice Hall, Upper Saddle River, NJ, 1996.
11. R. G. Lyons, *Understanding Digital Signal Processing*, Addison-Wesley, Reading, MA, 1997.
12. F. J. Harris, On the use of windows for harmonic analysis with the discrete Fourier transform, *Proceedings of the IEEE*, Vol. 66, 1978, pp. 51–83.
13. I. F. Progri, W. R. Michalson, and R. Chassaing, Fast and efficient filter design and implementation on the TMS320C6711 digital signal processor, *2001 International Conference on Acoustics, Speech, and Signal Processing Student Forum*, May 2001.
14. B. Porat, *A Course in Digital Signal Processing*, Wiley, New York, 1997.
15. T. W. Parks and C. S. Burrus, *Digital Filter Design*, Wiley, New York, 1987.
16. S. D. Stearns and R. A. David, *Signal Processing in Fortran and C*, Prentice Hall, Upper Saddle River, NJ, 1993.
17. N. Ahmed and T. Natarajan, *Discrete-Time Signals and Systems*, Reston Publishing, Reston, VA, 1983.
18. S. J. Orfanidis, *Introduction to Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1996.
19. A. Antoniou, *Digital Filters: Analysis, Design, and Applications*, McGraw-Hill, New York, 1993.
20. E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing: A Practical Approach*, Addison-Wesley, Reading, MA, 1993.
21. P. A. Lynn and W. Fuerst, *Introductory Digital Signal Processing with Computer Applications*, Wiley, New York, 1994.
22. R. D. Strum and D. E. Kirk, *First Principles of Discrete Systems and Digital Signal Processing*, Addison-Wesley, Reading, MA, 1988.

23. D. J. DeFatta, J. G. Lucas, and W. S. Hodgkiss, *Digital Signal Processing: A System Approach*, Wiley, New York, 1988.

24. C. S. Williams, *Designing Digital Filters*, Prentice Hall, Upper Saddle River, NJ, 1986.

25. R. W. Hamming, *Digital Filters*, Prentice Hall, Upper Saddle River, NJ, 1983.

26. S. K. Mitra and J. F. Kaiser, eds., *Handbook for Digital Signal Processing*, Wiley, New York, 1993.

27. S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, 1998.

28. R. Chassaing, B. Bitler, and D. W. Horning, Real-time digital filters in C, *Proceedings of the 1991 ASEE Annual Conference*, June 1991.

29. R. Chassaing and P. Martin, Digital filtering with the floating-point TMS320C30 digital signal processor, *Proceedings of the 21st Annual Pittsburgh Conference on Modeling and Simulation*, May 1990.

30. S. D. Stearns and R. A. David, *Signal Processing in Fortran and C*, Prentice Hall, Upper Saddle River, NJ, 1993.

31. R. A. Roberts and C. T. Mullis, *Digital Signal Processing*, Addison-Wesley, Reading, MA, 1987.

32. E. P. Cunningham, *Digital Filtering: An Introduction*, Houghton Mifflin, Boston, 1992.

33. N. J. Loy, *An Engineer's Guide to FIR Digital Filters*, Prentice Hall, Upper Saddle River, NJ, 1988.

34. H. Nuttall, Some windows with very good sidelobe behavior, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 1, Feb. 1981.

35. L. C. Ludemen, *Fundamentals of Digital Signal Processing*, Harper & Row, New York, 1986.

36. M. Bellanger, *Digital Processing of Signals: Theory and Practice*, Wiley, New York, 1989.

37. M. G. Bellanger, *Digital Filters and Signal Analysis*, Prentice Hall, Upper Saddle River, NJ, 1986.

38. F. J. Taylor, *Principles of Signals and Systems*, McGraw-Hill, New York, 1994.

39. F. J. Taylor, *Digital Filter Design Handbook*, Marcel Dekker, New York, 1983.

40. W. D. Stanley, G. R. Dougherty, and R. Dougherty, *Digital Signal Processing*, Reston Publishing, Reston, VA, 1984.

41. R. Kuc, *Introduction to Digital Signal Processing*, McGraw-Hill, New York, 1988.

42. H. Baher, *Analog and Digital Signal Processing*, Wiley, New York, 1990.

43. J. R. Johnson, *Introduction to Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1989.

44. S. Haykin, *Modern Filters*, Macmillan, New York, 1989.

45. T. Young, *Linear Systems and Digital Signal Processing*, Prentice Hall, Upper Saddle River, NJ, 1985.

46. A. Ambardar, *Analog and Digital Signal Processing*, PWS, MA, 1995.

47. A. W. M. van den Enden and N. A. M. Verhoeckx, *Discrete-Time Signal Processing*, Prentice-Hall International, Hemel Hempstead, Hertfordshire, England, 1989.

48. MATLAB, MathWorks, Natick, MA.

49. R. Chassaing and D. W. Horning, *Digital Signal Processing with the TMS320C25*, Wiley, New York, 1990.