CHAPTER 4

REAL-TIME FILTERING

Filtering is the most commonly used signal processing technique. Filters are usually used to remove or attenuate an undesired portion of a signal's spectrum while enhancing the desired portions of the signal. Often the undesired portion of a signal is random noise with a different frequency content than the desired portion of the signal. Thus, by designing a filter to remove some of the random noise, the signal-to-noise ratio can be improved in some measurable way.

Filtering can be performed using analog circuits with continuous-time analog inputs or using digital circuits with discrete-time digital inputs. In systems where the input signal is digital samples (in music synthesis or digital transmission systems, for example) a digital filter can be used directly. If the input signal is from a sensor which produces an analog voltage or current, then an analog-to-digital converter (A/D converter) is required to create the digital samples. In either case, a digital filter can be used to alter the spectrum of the sampled signal, x_i, in order to produce an enhanced output, y_i. Digital filtering can be performed in either the time domain (see section 4.1) or the frequency domain (see section 4.4), with general-purpose computers using previously stored digital samples or in real-time with dedicated hardware.

4.1 REAL-TIME FIR AND IIR FILTERS

Figure 4.1 shows a typical digital filter structure containing N memory elements used to store the input samples and N memory elements (or delay elements) used to store the output sequence. As a new sample comes in, the contents of each of the input memory elements are copied to the memory elements to the right. As each output sample is formed

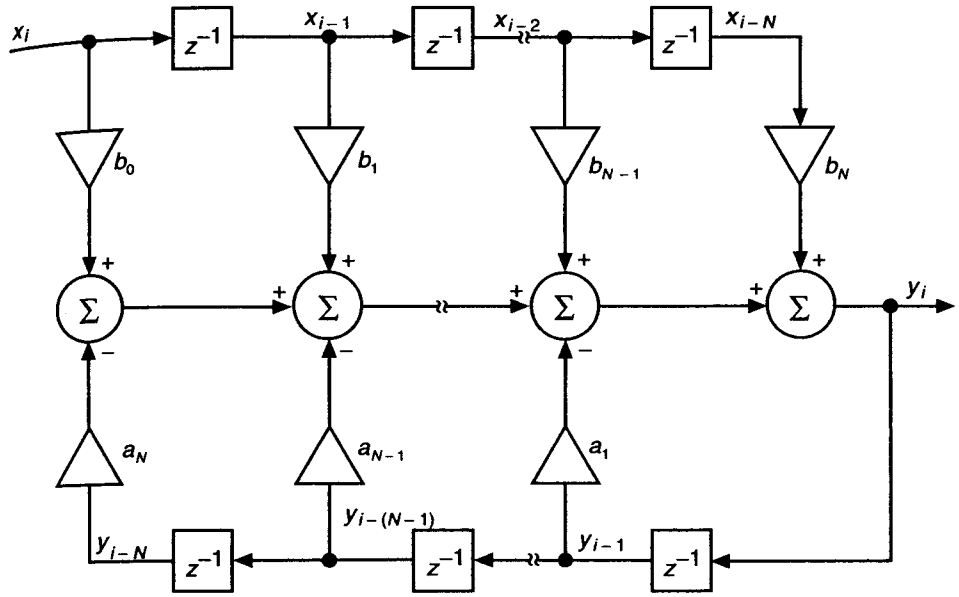

FIGURE 4.1 Filter structure of Nth order filter. The previous N input and output samples stored in the delay elements are used to form the output sum.

by accumulating the products of the coefficients and the stored values, the output memory elements are copied to the left. The series of memory elements forms a digital delay line. The delayed values used to form the filter output are called *taps* because each output makes an intermediate connection along the delay line to provide a particular delay. This filter structure implements the following difference equation:

$$y(n) = \sum_{q=0}^{Q-1} b_q x(n-q) - \sum_{p=1}^{P-1} a_p y(n-p). \tag{4.1}$$

As discussed in Chapter 1, filters can be classified based on the duration of their impulse response. Filters where the a_n terms are zero are called *finite impulse response* (FIR) *filters,* because the response of the filter to an impulse (or any other input signal) cannot change N samples past the last excitation. Filters where one or more of the a_n terms are nonzero are *infinite impulse response* (IIR) *filters.* Because the output of an IIR filter depends on a sum of the N input samples as well as a sum of the past N output samples, the output response is essentially dependent on all past inputs. Thus, the filter output response to any finite length input is infinite in length, giving the IIR filter infinite memory.

Finite impulse response (FIR) *filters* have several properties that make them useful for a wide range of applications. A perfect linear phase response can easily be con-

structed with an FIR filter allowing a signal to be passed without phase distortion. FIR filters are inherently stable, so stability concerns do not arise in the design or implementation phase of development. Even though FIR filters typically require a large number of multiplies and adds per input sample, they can be implemented using fast convolution with FFT algorithms (see section 4.4.1). Also, FIR structures are simpler and easier to implement with standard fixed-point digital circuits at very high speeds. The only possible disadvantage of FIR filters is that they require more multiplies for a given frequency response when compared to IIR filters and, therefore, often exhibit a longer processing delay for the input to reach the output.

During the past 20 years, many techniques have been developed for the design and implementation of FIR filters. *Windowing* is perhaps the simplest and oldest FIR design technique (see section 4.1.2), but is quite limited in practice. The *window design method* has no independent control over the passband and stopband ripple. Also, filters with unconventional responses, such as *multiple passband filters,* cannot be designed. On the other hand, window design can be done with a pocket calculator and can come close to optimal in some cases.

This section discusses FIR filter design with different equiripple error in the passbands and stopbands. This class of FIR filters is widely used primarily because of the well-known *Remez exchange algorithm* developed for FIR filters by Parks and McClellan. The general Parks-McClellan program can be used to design filters with several passbands and stopbands, digital differentiators, and Hilbert transformers. The FIR coefficients obtained program can be used directly with the structure shown in Figure 4.1 (with the a_n terms equal to zero). The floating-point coefficients obtained can be directly used with floating-point arithmetic (see section 4.1.1).

The Parks-McClellan program is available on the IEEE digital signal processing tape or as part of many of the filter design packages available for personal computers. The program is also printed in several DSP texts (see Elliot, 1987, or Rabiner and Gold, 1975). The program REMEZ.C is a C language implementation of the Parks-McClellan program and is included on the enclosed disk. An example of a filter designed using the REMEZ program is shown at the end of section 4.1.2. A simple method to obtain FIR filter coefficients based on the Kaiser window is also described in section 4.1.2. Although this method is not as flexible as the Remez exchange algorithm it does provide optimal designs without convergence problems or filter length restrictions.

4.1.1 FIR Filter Function

Figure 4.2 shows a block diagram of the FIR real-time filter implemented by the function **`fir_filter`** (shown in Listing 4.1 and contained in the file FILTER.C). The **`fir_filter`** function implements the FIR filter using a history pointer and coefficients passed to the function. The history array is allocated by the calling program and is used to store the previous $N-1$ input samples (where N is the number of FIR filter coefficients). The last few lines of code in **`fir_filter`** implements the multiplies and accumulates required for the FIR filter of length N. As the history pointer is advanced by using a post-increment, the coefficient pointer is post-decremented. This effectively time reverses the

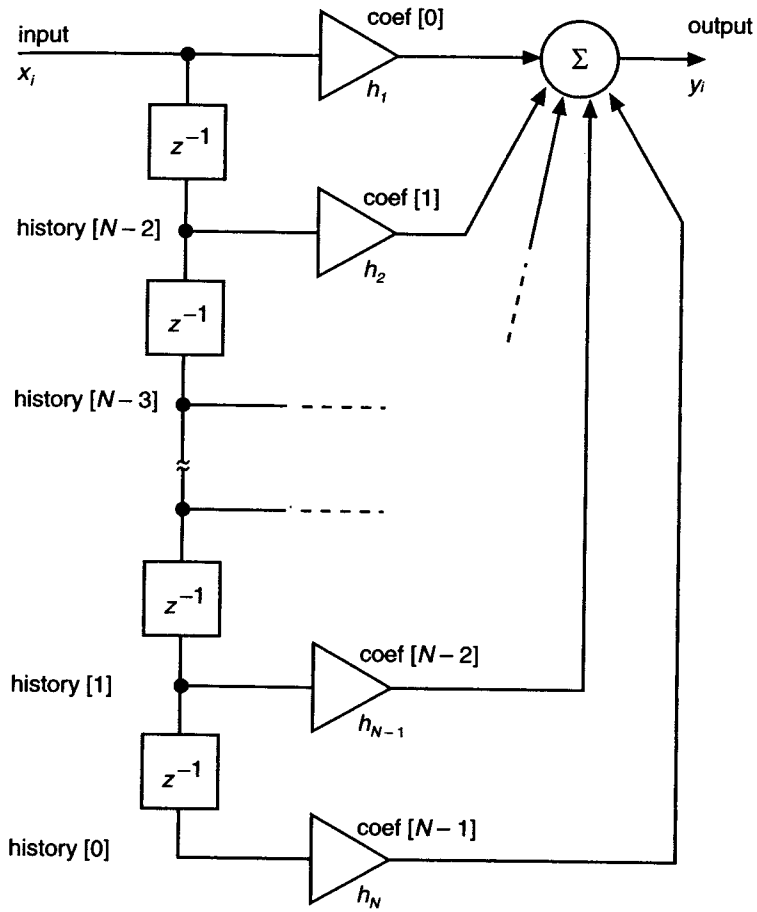

FIGURE 4.2 Block diagram of real-time N tap FIR filter structure as implemented by function **`fir_filter`**.

coefficients so that a true convolution is implemented. On some microprocessors, post-decrement is not implemented efficiently so this code becomes less efficient. Improved efficiency can be obtained in this case by storing the filter coefficients in time-reversed order. Note that if the coefficients are symmetrical, as for simple linear phase lowpass filters, then the time-reversed order and normal order are identical. After the **`for`** loop and $N-1$ multiplies have been completed, the history array values are shifted one sample toward **`history[0]`**, so that the new input sample can be stored in **`history[N-1]`**. The **`fir_filter`** implementation uses pointers extensively for maximum efficiency.

```
/*************************************************************************

fir_filter - Perform fir filtering sample by sample on floats

Requires array of filter coefficients and pointer to history.
Returns one output sample for each input sample.

float fir_filter(float input,float *coef,int n,float *history)

    float input         new float input sample
    float *coef         pointer to filter coefficients
    int n               number of coefficients in filter
    float *history      history array pointer

Returns float value giving the current output.

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

float fir_filter(float input,float *coef,int n,float *history)
{
    int i;
    float *hist_ptr,*hist1_ptr,*coef_ptr;
    float output;

    hist_ptr = history;
    hist1_ptr = hist_ptr;           /* use for history update */
    coef_ptr = coef + n - 1;        /* point to last coef */

/* form output accumulation */
    output = *hist_ptr++ * (*coef_ptr--);
    for(i = 2 ; i < n ; i++) {
        *hist1_ptr++ = *hist_ptr;           /* update history array */
        output += (*hist_ptr++) * (*coef_ptr--);
    }
    output += input * (*coef_ptr);          /* input tap */
    *hist1_ptr = input;                     /* last history */

    return(output);
}
```

LISTING 4.1 Function **fir_filter(input,coef,n,history)**.

4.1.2 FIR Filter Coefficient Calculation

Because the stopband attenuation and passband ripple and the filter length are all specified as inputs to filter design programs, it is often difficult to determine the filter length required for a particular filter specification. Guessing the filter length will eventually reach a reasonable solution but can take a long time. For one stopband and one passband

the following approximation for the filter length (N) of an optimal lowpass filter has been developed by Kaiser:

$$N = \frac{A_{stop} - 16}{29\Delta f} + 1 \tag{4.2}$$

where:

$$\Delta f = (f_{stop} - f_{pass})/f_s$$

and A_{stop} is the minimum stopband attenuation (in dB) of the stopband from f_{stop} to $f_s/2$. The approximation for N is accurate within about 10 percent of the actual required filter length. For the Kaiser window design method, the passband error (δ_1) is equal to the stopband error (δ_2) and is related to the passband ripple (A_{max}) and stopband attenuation (in dB) as follows:

$$\delta_1 = 1 - 10^{-A_{max}/40}$$

$$\delta_2 = 10^{-A_{max}/20}$$

$$A_{max} = -40\log_{10}\left(1 - 10^{-A_{stop}/20}\right)$$

As a simple example, consider the following filter specifications, which specify a lowpass filter designed to remove the upper half of the signal spectrum:

Passband (f_{pass}):	$0–0.19 f_s$
Passband ripple (A_{max}):	< 0.2 dB
Stopband (f_{stop}):	$0.25 - 0.5 f_s$
Stopband Attenuation (A_{stop}):	> 40 dB

From these specifications

$$\delta_1 = 0.01145,$$

$$\delta_2 = 0.01,$$

$$\Delta f = 0.06.$$

The result of Equation (4.2) is $N = 37$. Greater stopband attenuation or a smaller transition band can be obtained with a longer filter. The filter coefficients are obtained by multiplying the Kaiser window coefficients by the ideal lowpass filter coefficients. The ideal lowpass coefficients for a very long odd length filter with a cutoff frequency of f_c are given by the following sinc function:

$$c_k = \frac{\sin(2f_c k\pi)}{k\pi}. \tag{4.3}$$

Note that the center coefficient is $k = 0$ and the filter has even symmetry for all coefficients above $k = 0$. Very poor stopband attenuation would result if the above coefficients

were truncated by using the 37 coefficients (effectively multiplying the *sinc function* by a rectangular window, which would have a stopband attenuation of about 13 dB). However, by multiplying these coefficients by the appropriate Kaiser window, the stopband and passband specifications can be realized. The symmetrical Kaiser window, w_k, is given by the following expression:

$$w_k = \frac{I_0\left\{\beta\sqrt{1-\left(1-\frac{2k}{N-1}\right)^2}\right\}}{I_0(\beta)}, \qquad (4.4)$$

where $I_0(\beta)$ is a modified zero order Bessel function of the first kind, β is the Kaiser window parameter which determines the stopband attenuation. The empirical formula for β when A_{stop} is less than 50 dB is $\beta = 0.5842*(A_{stop} - 21)^{0.4} + 0.07886*(A_{stop} - 21)$. Thus, for a stopband attenuation of 40 dB, $\beta = 3.39532$. Listing 4.2 shows program KSRFIR.C, which can be used to calculate the coefficients of a FIR filter using the Kaiser window method. The length of the filter must be odd and bandpass; bandstop or highpass filters can also be designed. Figure 4.3(a) shows the frequency response of the resulting 37-point lowpass filter, and Figure 4.3(b) shows the frequency response of a 35-point lowpass filter designed using the Parks-McClellan program. The following computer dialog shows the results obtained using the REMEZ.C program:

```
   ...     REMEZ EXCHANGE FIR FILTER DESIGN PROGRAM ...

1: EXAMPLE1 — LOWPASS FILTER
2: EXAMPLE2 — BANDPASS FILTER
3: EXAMPLE3 — DIFFERENTIATOR
4: EXAMPLE4 — HILBERT TRANSFORMER
5: KEYBOARD — GET INPUT PARAMETERS FROM KEYBOARD

selection [1 to 5] ? 5

number of coefficients [3 to 128] ? 35

Filter types are: 1=Bandpass, 2=Differentiator, 3=Hilbert

filter type [1 to 3] ? 1

number of bands [1 to 10] ? 2
Now inputting edge (corner) frequencies for 4 band edges

edge frequency for edge (corner) # 1 [0 to 0.5] ? 0

edge frequency for edge (corner) # 2 [0 to 0.5] ? .19

edge frequency for edge (corner) # 3 [0.19 to 0.5] ? .25
```

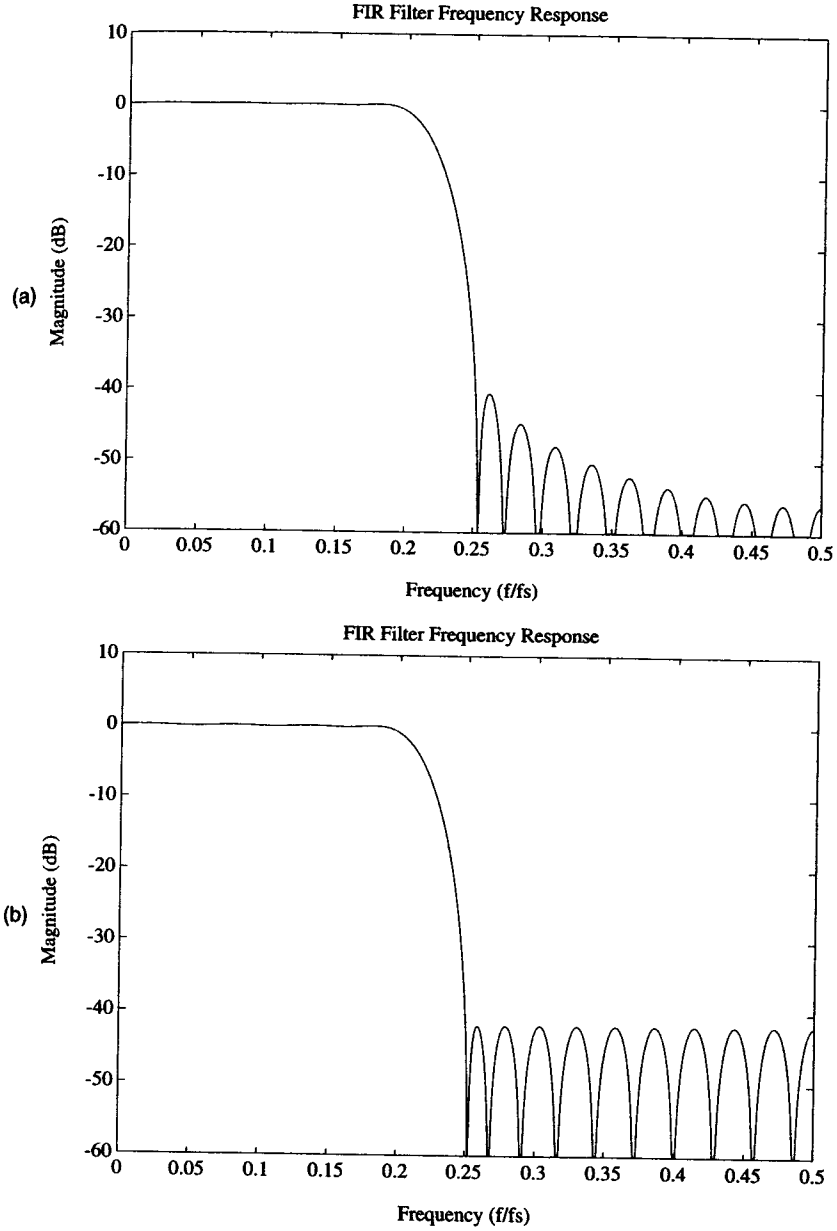

FIGURE 4.3 **(a)** Frequency response of 37 tap FIR filter designed using the Kaiser window method. **(b)** Frequency response of 35 tap FIR filter designed using the Parks-McClellan program.

```
edge frequency for edge (corner) # 4 [0.25 to 0.5] ? .5

gain of band # 1 [0 to 1000] ? 1

weight of band # 1 [0.1 to 100] ? 1

gain of band # 2 [0 to 1000] ? 0

weight of band # 2 [0.1 to 100] ? 1

#coeff = 35
Type = 1
#bands = 2
Grid = 16
E[1] = 0.00
E[2] = 0.19
E[3] = 0.25
E[4] = 0.50
Gain, wt[1] = 1.00 1.00
Gain, wt[2] = 0.00 1.00

Iteration 1 2 3 4 5 6 7
********************************************************************
                    FINITE IMPULSE RESPONSE (FIR)
                    LINEAR PHASE DIGITAL FILTER DESIGN
                    REMEZ EXCHANGE ALGORITHM
                    BANDPASS FILTER
              FILTER LENGTH = 35
              ***** IMPULSE RESPONSE *****
                    H(  1) = -6.360096001e-003 = H(  35)
                    H(  2) = -7.662615827e-005 = H(  34)
                    H(  3) =  7.691285583e-003 = H(  33)
                    H(  4) =  5.056414595e-003 = H(  32)
                    H(  5) = -8.359812578e-003 = H(  31)
                    H(  6) = -1.040090568e-002 = H(  30)
                    H(  7) =  8.696002091e-003 = H(  29)
                    H(  8) =  2.017050147e-002 = H(  28)
                    H(  9) = -2.756078525e-003 = H(  27)
                    H( 10) = -3.003477728e-002 = H(  26)
                    H( 11) = -8.907503106e-003 = H(  25)
                    H( 12) =  4.171576865e-002 = H(  24)
                    H( 13) =  3.410815421e-002 = H(  23)
                    H( 14) = -5.073291821e-002 = H(  22)
                    H( 15) = -8.609754956e-002 = H(  21)
                    H( 16) =  5.791494030e-002 = H(  20)
                    H( 17) =  3.117008479e-001 = H(  19)
                    H( 18) =  4.402931165e-001 = H(  18)
```



```
                         BAND 1          BAND 2
LOWER BAND EDGE       0.00000000      0.25000000
UPPER BAND EDGE       0.19000000      0.50000000
DESIRED VALUE         1.00000000      0.00000000
WEIGHTING             1.00000000      1.00000000
DEVIATION             0.00808741      0.00808741
DEVIATION IN DB     -41.84380886    -41.84380886

EXTREMAL FREQUENCIES
  0.0156250   0.0520833   0.0815972   0.1093750   0.1371528
  0.1614583   0.1822917   0.1900000   0.2500000   0.2586806
  0.2777778   0.3038194   0.3298611   0.3576389   0.3854167
  0.4131944   0.4427083   0.4704861   0.5000000

FIR coefficients written to text file COEF.DAT
```

Note that the Parks-McClellan design achieved the specifications with two fewer coefficients, and the stopband attenuation is 1.8 dB better than the specification. Because the stopband attenuation, passband ripple, and filter length are all specified as inputs to the Parks-McClellan filter design program, it is often difficult to determine the filter length required for a particular filter specification. Guessing the filter length will eventually reach a reasonable solution but can take a long time. For one stopband and one passband, the following approximation for the filter length (N) of an optimal lowpass filter has been developed by Kaiser:

$$N = \frac{-20\log_{10}\sqrt{\delta_1\delta_2} - 13}{14.6\Delta f} + 1 \tag{4.5}$$

where:

$$\delta_1 = 1 - 10^{-A_{max}/40}$$

$$\delta_2 = 10^{-A_{stop}/20}$$

$$\Delta f = (f_{stop} - f_{pass})/f_s$$

A_{max} is the total passband ripple (in dB) of the passband from 0 to f_{pass}. If the maximum of the magnitude response is 0 dB, then A_{max} is the maximum attenuation throughout the passband. A_{stop} is the minimum stopband attenuation (in dB) of the stopband from f_{stop} to $f_s/2$. The approximation for N is accurate within about 10 percent of the actual required filter length (usually on the low side). The ratio of the passband error (δ_1) to the stopband error (δ_2) is entered by choosing appropriate weights for each band. Higher weighting of stopbands will increase the minimum attenuation; higher weighting of the passband will decrease the passband ripple.

The coefficients for the Kaiser window design (variable name **fir_lpf37k**) and the Parks-McClellan design (variable name **fir_lpf35**) are contained in the include file FILTER.H.

```
/* Linear phase FIR filter coefficient computation using the Kaiser window
design method. Filter length is odd. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "rtdspc.h"

double get_float(char *title_string,double low_limit,double up_limit);
void filter_length(double att,double deltaf,int *nfilt,int *npair,double *beta);
double izero(double y);

void main()
{
    static float h[500], w[500], x[500];
    int eflag, filt_cat, npair, nfilt, n;
    double att, fa, fp, fa1, fa2, fp1, fp2, deltaf, d1, d2, fl, fu, beta;
    double fc, fm, pifc, tpifm, i, y, valizb;
    char ft_s[128];

    char  fp_s[] = "Passband edge frequency Fp";
    char  fa_s[] = "Stopband edge frequency Fa";
    char fp1_s[] = "Lower passband edge frequency Fp1";
    char fp2_s[] = "Upper passband edge frequency Fp2";
    char fa1_s[] = "Lower stopband edge frequency Fa1";
    char fa2_s[] = "Upper stopband edge frequency Fa2";

    printf("\nFilter type (lp, hp, bp, bs) ? ");
    gets(ft_s);
    strupr( ft_s );
    att = get_float("Desired stopband attenuation (dB)", 10, 200);
    filt_cat = 0;
    if( strcmp( ft_s, "LP" ) == 0 ) filt_cat = 1;
    if( strcmp( ft_s, "HP" ) == 0 ) filt_cat = 2;
    if( strcmp( ft_s, "BP" ) == 0 ) filt_cat = 3;
    if( strcmp( ft_s, "BS" ) == 0 ) filt_cat = 4;
    if(!filt_cat) exit(0);

    switch ( filt_cat ){
        case 1: case 2:
            switch ( filt_cat ){
                case 1:
                    fp = get_float( fp_s, 0, 0.5 );
                    fa = get_float( fa_s, fp, 0.5 ); break;
                case 2:
                    fa = get_float( fa_s, 0, 0.5 );
                    fp = get_float( fp_s, fa, 0.5 );
```

LISTING 4.2 Program KSRFIR to calculate FIR filter coefficients using the Kaiser window method. (*Continued*)


```
            }
            deltaf = (fa-fp) ; if(filt_cat == 2) deltaf = -deltaf;
            filter_length( att, deltaf, &nfilt, &npair, &beta );
            if( npair > 500 ){
                printf("\n*** Filter length %d is too large.\n", nfilt );
                exit(0);
            }
            printf("\n...filter length: %d ...beta: %f", nfilt, beta );
            fc = (fp + fa); h[npair] = fc;
            if ( filt_cat == 2 ) h[npair] = 1 - fc;
            pifc = PI * fc;
            for ( n=0; n < npair; n++){
                i = (npair - n);
                h[n] = sin(i * pifc) / (i * PI);
                if( filt_cat == 2 ) h[n] = - h[n];
            }
            break;
    case 3: case 4:
            printf("\n--> Transition bands must be equal <--");
            do {
                eflag = 0;
                switch (filt_cat){
                    case 3:
                        fa1 = get_float( fa1_s,    0, 0.5);
                        fp1 = get_float( fp1_s, fa1, 0.5);
                        fp2 = get_float( fp2_s, fp1, 0.5);
                        fa2 = get_float( fa2_s, fp2, 0.5); break;
                    case 4:
                        fp1 = get_float( fp1_s,    0, 0.5);
                        fa1 = get_float( fa1_s, fp1, 0.5);
                        fa2 = get_float( fa2_s, fa1, 0.5);
                        fp2 = get_float( fp2_s, fa2, 0.5);
                }
                d1 = fp1 - fa1; d2 = fa2 - fp2;
                if ( fabs(d1 - d2) > 1E-5 ){
                    printf( "\n...error...transition bands not equal\n");
                    eflag = -1;
                }
            } while (eflag);
            deltaf = d1; if(filt_cat == 4) deltaf = -deltaf;
            filter_length( att, deltaf, &nfilt, &npair, &beta);
            if( npair > 500 ){
                printf("\n*** Filter length %d is too large.\n", nfilt );
                exit(0);
            }
            printf( "\n..filter length: %d ...beta: %f", nfilt, beta);
            fl = (fa1 + fp1) / 2; fu = (fa2 + fp2) / 2;
            fc = (fu - fl); fm = (fu + fl) / 2;
            h[npair] = 2 * fc; if( filt_cat == 4 ) h[npair] = 1 - 2 * fc;
```

LISTING 4.2 (*Continued*)

```
            pifc = PI * fc; tpifm = 2 * PI * fm;
            for (n = 0; n < npair; n++){
                i = (npair - n);
                h[n] = 2 * sin(i * pifc) * cos(i * tpifm) / (i * PI);
            if ( filt_cat == 4) h[n] = -h[n];
            } break;
        default: printf( "\n## error\n" ); exit(0);
    }

/* Compute Kaiser window sample values */
    y = beta; valizb = izero(y);
    for (n = 0; n <= npair; n++){
        i = (n - npair);
        y = beta * sqrt(1 - (i / npair) * (i / npair));
        w[n] = izero(y) / valizb;
    }

/* first half of response */
    for(n = 0; n <= npair; n++) x[n] = w[n] * h[n];

    printf("\n—First half of coefficient set...remainder by symmetry—");
    printf("\n #     ideal        window     actual     ");
    printf("\n          coeff        value    filter coeff");
    for(n=0; n <= npair; n++){
        printf("\n %4d    %9.6f    %9.6f    %9.6f",n, h[n], w[n], x[n]);
    }
}

/* Use att to get beta (for Kaiser window function) and nfilt (always odd
    valued and = 2*npair +1) using Kaiser's empirical formulas */
void filter_length(double att,double deltaf,int *nfilt,int *npair,double *beta)
{
    *beta = 0;      /* value of beta if att < 21 */
    if(att >= 50) *beta = .1102 * (att - 8.71);
    if (att < 50 & att >= 21)
        *beta = .5842 * pow( (att-21), 0.4) + .07886 * (att - 21);
    *npair = (int)( (att - 8) / (29 * deltaf) );
    *nfilt = 2 * *npair +1;
}

/* Compute Bessel function Izero(y) using a series approximation */
double izero(double y){
    double s=1, ds=1, d=0;
    do{
        d = d + 2; ds = ds * (y*y)/(d*d);
        s = s + ds;
    } while( ds > 1E-7 * s);
    return(s);
}
```

LISTING 4.2 *(Continued)*

4.1.3 IIR Filter Function

Infinite impulse response (IIR) *filters* are realized by feeding back a weighted sum of past output values and adding this to a weighted sum of the previous and current input values. In terms of the structure shown in Figure 4.1, IIR filters have nonzero values for some or all of the a_n values. The major advantage of IIR filters compared to FIR filters is that a given order IIR filter can be made much more frequency selective than the same order FIR filter. In other words, IIR filters are computationally efficient. The disadvantage of the recursive realization is that IIR filters are much more difficult to design and implement. Stability, roundoff noise and sometimes phase nonlinearity must be considered carefully in all but the most trivial IIR filter designs.

The direct form IIR filter realization shown in Figure 4.1, though simple in appearance, can have severe response sensitivity problems because of coefficient quantization, especially as the order of the filter increases. To reduce these effects, the transfer function is usually decomposed into second order sections and then realized either as parallel or cascade sections (see chapter 1, section 1.3). In section 1.3.1 an IIR filter design and implementation method based on cascade decomposition of a transfer function into second order sections is described. The C language implementation shown in Listing 4.3 uses single-precision floating-point numbers in order to avoid coefficient quantization effects associated with fixed-point implementations that can cause instability and significant changes in the transfer function.

Figure 4.4 shows a block diagram of the cascaded second order IIR filter implemented by the **`iir_filter`** function shown in Listing 4.3. This realization is known as a direct form II realization because, unlike the structure shown in Figure 4.1, it has only

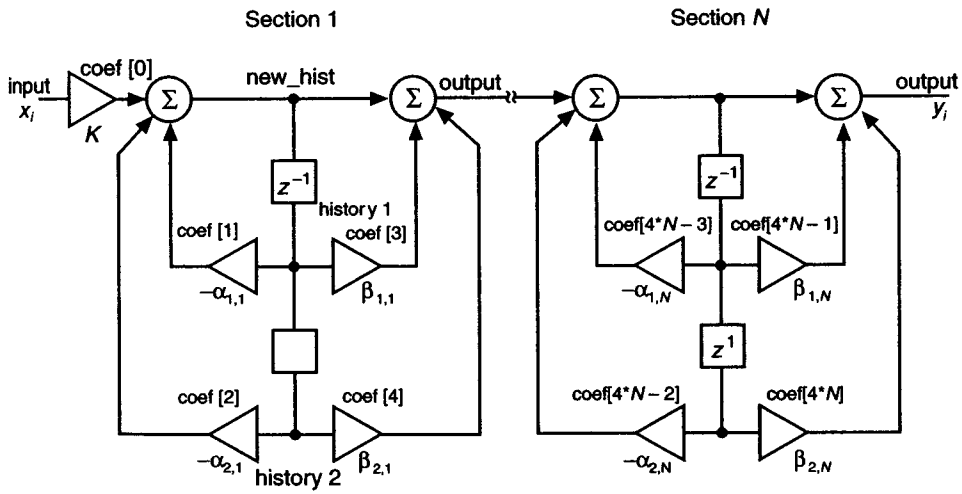

FIGURE 4.4 Block diagram of real-time IIR filter structure as implemented by function **`iir_filter`**.

```
/************************************************************************

iir_filter - Perform IIR filtering sample by sample on floats

Implements cascaded direct form II second order sections.
Requires arrays for history and coefficients.
The length (n) of the filter specifies the number of sections.
The size of the history array is 2*n.
The size of the coefficient array is 4*n + 1 because
the first coefficient is the overall scale factor for the filter.
Returns one output sample for each input sample.

float iir_filter(float input,float *coef,int n,float *history)

    float input         new float input sample
    float *coef         pointer to filter coefficients
    int n               number of sections in filter
    float *history      history array pointer

Returns float value giving the current output.

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

float iir_filter(float input,float *coef,int n,float *history)
{
    int i;
    float *hist1_ptr,*hist2_ptr,*coef_ptr;
    float output,new_hist,history1,history2;

    coef_ptr = coef;                /* coefficient pointer */

    hist1_ptr = history;            /* first history */
    hist2_ptr = hist1_ptr + 1;           /* next history */

    output = input * (*coef_ptr++);     /* overall input scale factor */

    for(i = 0 ; i < n ; i++) {
        history1 = *hist1_ptr;           /* history values */
        history2 = *hist2_ptr;

        output = output - history1 * (*coef_ptr++);
        new_hist = output - history2 * (*coef_ptr++);     /* poles */

        output = new_hist + history1 * (*coef_ptr++);
        output = output + history2 * (*coef_ptr++);      /* zeros */

        *hist2_ptr++ = *hist1_ptr;
        *hist1_ptr++ = new_hist;
```

LISTING 4.3 Function **iir_filter(input,coef,n,history)**. (*Continued*)


```
        hist1_ptr++;
        hist2_ptr++;
    }

    return(output);
}
```

LISTING 4.3 (*Continued*)

two delay elements for each second-order section. This realization is canonic in the sense that the structure has the fewest adds (4), multiplies (4), and delay elements (2) for each second order section. This realization should be the most efficient for a wide variety of general purpose processors as well as many of the processors designed specifically for digital signal processing.

IIR filtering will be illustrated using a lowpass filter with similar specifications as used in the FIR filter design example in section 4.1.2. The only difference is that in the IIR filter specification, linear phase response is not required. Thus, the passband is 0 to $0.2\,f_s$ and the stopband is $0.25\,f_s$ to $0.5\,f_s$. The passband ripple must be less than 0.5 dB and the stopband attenuation must be greater than 40 dB. Because elliptic filters (also called *Cauer filters*) generally give the smallest transition bandwidth for a given order, an elliptic design will be used. After referring to the many elliptic filter tables, it is determined that a fifth order elliptic filter will meet the specifications. The elliptic filter tables in Zverev (1967) give an entry for a filter with a 0.28 dB passband ripple and 40.19 dB stopband attenuation as follows:

$\Omega_s = 1.3250$	(stopband start of normalized prototype)
$\sigma_0 = -0.5401$	(first order real pole)
$\sigma_1 = -0.5401$	(real part of first biquad section)
$\sigma_3 = -0.5401$	(real part of second biquad section)
$\Omega_1 = 1.0277$	(imaginary part of first biquad section)
$\Omega_2 = 1.9881$	(first zero on imaginary axis)
$\Omega_3 = 0.7617$	(imaginary part of second biquad section)
$\Omega_4 = 1.3693$	(second zero on imaginary axis)

As shown above, the tables in Zverev give the pole and zero locations (real and imaginary coordinates) of each biquad section. The two second-order sections each form a conjugate pole pair and the first-order section has a single pole on the real axis. Figure 4.5(a) shows the locations of the 5 poles and 4 zeros on the complex s-plane. By expanding the complex pole pairs, the s-domain transfer function of a fifth-order filter in terms of the above variables can be obtained. The z-domain coefficients are then determined using the bilinear transform (see Embree and Kimble, 1991). Figure 4.5(b) shows the locations of the poles and zeros on the complex z-plane. The resulting z-domain transfer function is as follows:

$$\frac{0.0553(1+z^{-1})}{1-0.436z^{-1}}\,\frac{1+0.704z^{-1}+z^{-2}}{1-0.523z^{-1}-0.86z^{-2}}\,\frac{1-0.0103z^{-1}+z^{-2}}{1-0.696z^{-1}-0.486z^{-2}}$$

Figure 4.6 shows the frequency response of this 5th order digital IIR filter.

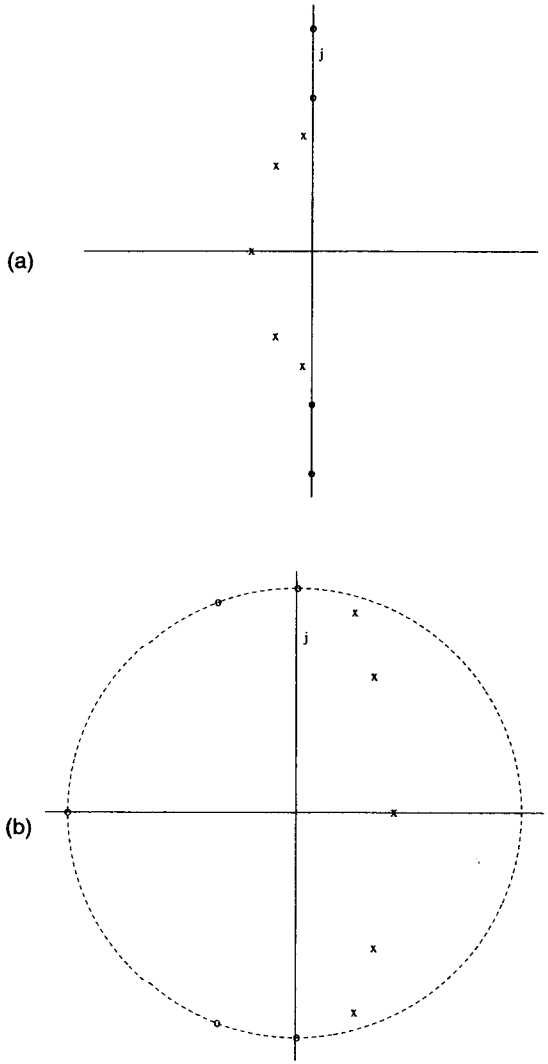

FIGURE 4.5 Pole-zero plot of fifth-order elliptic IIR lowpass filter. **(a)** *s*-plane representation of analog prototype fifth-order elliptic filter. Zeros are indicated by "o" and poles are indicated by "x". **(b)** *z*-plane representation of lowpass digital filter with cutoff frequency at 0.2 f_s. In each case, poles are indicated with "x" and zeros with "o".

The function **`iir_filter`** (shown in Listing 4.3) implements the direct form II cascade filter structure illustrated in Figure 4.4. Any number of cascaded second order sections can be implemented with one overall input (x_i) and one overall output (y_i). The coefficient array for the fifth order elliptic lowpass filter is as follows:

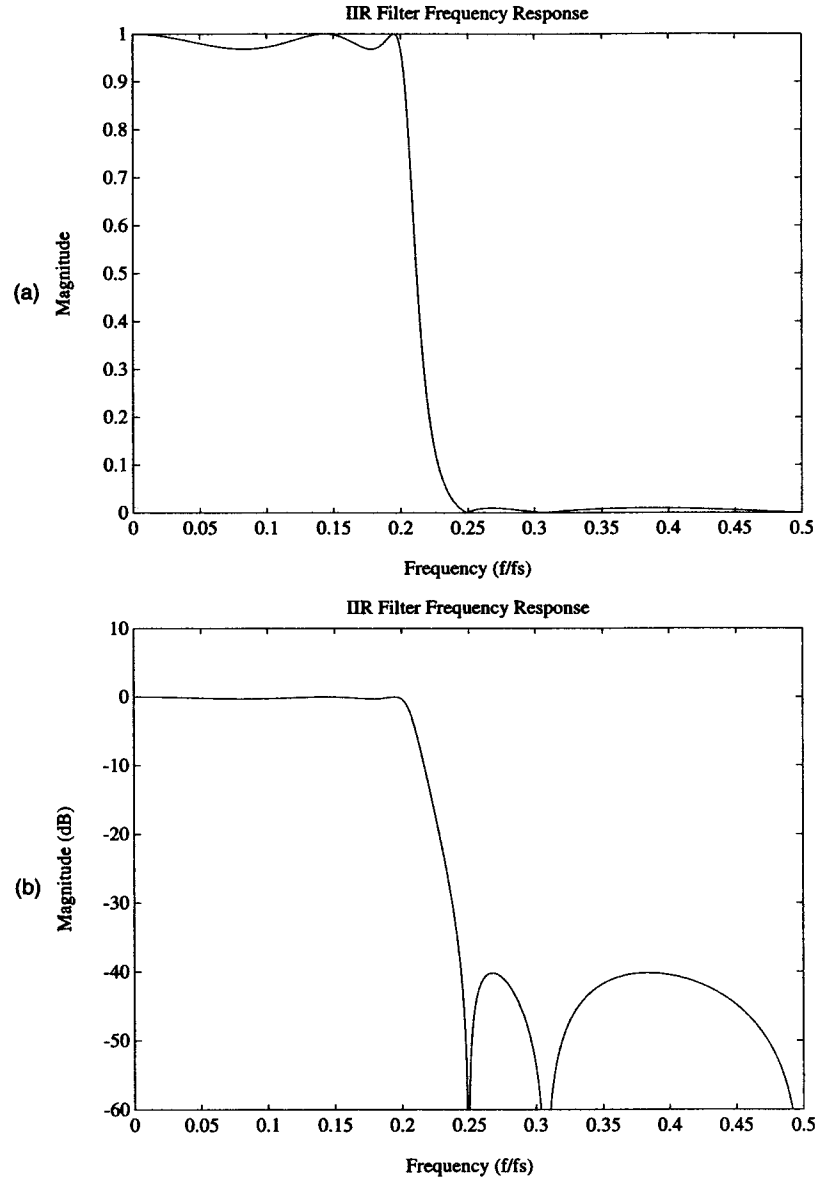

FIGURE 4.6 **(a)** Lowpass fifth-order elliptic IIR filter linear magnitude frequency response. **(b)** Lowpass fifth-order elliptic IIR filter frequency response. Log magnitude in decibels versus frequency.

```
float iir_lpf5[13] = {
     0.0552961603,
     -0,4363630712, 0.0000000000, 1.0000000000, 0.0000000000,
    -0.5233039260, 0.8604439497, 0.7039934993, 1.0000000000,
    -0.6965782046, 0.4860509932, -0.0103216320, 1.0000000000
};
```

The number of sections required for this filter is three, because the first-order section is implemented in the same way as the second-order sections, except that the second-order terms (the third and fifth coefficients) are zero. The coefficients shown above were obtained using the bilinear transform and are contained in the include file FILTER.H. The definition of this filter is, therefore, global to any module that includes FILTER.H. The **iir_filter** function filters the floating-point input sequence on a sample-by-sample basis so that one output sample is returned each time **iir_filter** is invoked. The history array is used to store the two history values required for each second-order section. The history data (two elements per section) is allocated by the calling function. The initial condition of the history variables is zero if **calloc** is used, because it sets all the allocated space to zero. If the history array is declared as static, most compilers initialize static space to zero. Other initial conditions can be loaded into the filter by allocating and initializing the history array before using the **iir_filter** function. The coefficients of the filter are stored with the overall gain constant (K) first, followed by the denominator coefficients that form the poles, and the numerator coefficients that form the zeros for each section. The input sample is first scaled by the K value, and then each second-order section is implemented. The four lines of code in the **iir_filter** function used to implement each second-order section are as follows:

```
output = output - history1 * (*coef_ptr++);
new_hist = output - history2 * (*coef_ptr++);    /* poles */

output = new_hist + history1 * (*coef_ptr++);
output = output + history2 * (*coef_ptr++);      /* zeros */
```

The **history1** and **history2** variables are the current history associated with the section and should be stored in floating-point registers (if available) for highest efficiency. The above code forms the new history value (the portion of the output which depends on the past outputs) in the variable **new_hist** to be stored in the history array for use by the next call to **iir_filter.** The history array values are then updated as follows:

```
*hist2_ptr++ = *hist1_ptr;
*hist1_ptr++ = new_hist;
hist1_ptr++;
hist2_ptr++;
```

This results in the oldest history value (***hist2_ptr**) being lost and updated with the more recent ***hist1_ptr** value. The **new_hist** value replaces the old

***hist1_ptr** value for use by the next call to **iir_filter.** Both history pointers are incremented twice to point to the next pair of history values to be used by the next second-order section.

4.1.4 Real-Time Filtering Example

Real-time filters are filters that are implemented so that a continuous stream of input samples can be filtered to generate a continuous stream of output samples. In many cases, real-time operation restricts the filter to operate on the input samples individually and generate one output sample for each input sample. Multiple memory accesses to previous input data are not possible, because only the current input is available to the filter at any given instant in time. Thus, some type of history must be stored and updated with each new input sample. The management of the filter history almost always takes a portion of the processing time, thereby reducing the maximum sampling rate which can be supported by a particular processor. The functions **fir_filter** and **iir_filter** are implemented in a form that can be used for real-time filtering. Suppose that the functions **getinput()** and **sendout()** return an input sample and generate an output sample at the appropriate time required by the external hardware. The following code can be used with the **iir_filter** function to perform continuous real-time filtering:

```
static float histi[6];
for(;;)
     sendout(iir_filter(getinput(),iir_lpf5,3,histi));
```

In the above infinite loop **for** statement, the total time required to execute the **in, iir_filter,** and **out** functions must be less than the filter sampling rate in order to insure that output and input samples are not lost. In a similar fashion, a continuous real-time FIR filter could be implemented as follows:

```
static float histf[34];
for(;;)
     sendout(fir_filter(getinput(),fir_lpf35,35,histf));
```

Source code for **sendout()** and **getinput()** interrupt driven input/output functions is available on the enclosed disk for several DSP processors. C code which emulates **getinput()** and **sendout()** real-time functions using disk read and write functions is also included on the disk and is shown in Listing 4.4. These routines can be used to debug real-time programs using a simpler and less expensive general purpose computer environment (IBM-PC or UNIX system, for example). The functions shown in Listing 4.4 read and write disk files containing floating-point numbers in an ASCII text format. The functions shown in Listings 4.5 and 4.6 read and write disk files containing fixed-point numbers in the popular WAV binary file format. The WAV file format is part of the Resource Interchange File Format (RIFF), which is popular on many multimedia platforms.

(text continues on page 158)

```
#include <stdlib.h>
#include <stdio.h>

/* getinput - get one sample from disk to simulate real-time input */

float getinput()
{
    static FILE *fp = NULL;
    float x;
/* open input file if not done in previous calls */
    if(!fp) {
        char s[80];
        printf("\nEnter input file name ? ");
        gets(s);
        fp = fopen(s,"r");
        if(!fp) {
            printf("\nError opening input file in GETINPUT\n");
            exit(1);
        }
    }
/* read data until end of file */
    if(fscanf(fp,"%f",&x) != 1) exit(1);
    return(x);
}

/* sendout - send sample to disk to simulate real-time output */

void sendout(float x)
{
    static FILE *fp = NULL;
/* open output file if not done in previous calls */
    if(!fp) {
        char s[80];
        printf("\nEnter output file name ? ");
        gets(s);
        fp = fopen(s,"w");
        if(!fp) {
            printf("\nError opening output file in SENDOUT\n");
            exit(1);
        }
    }
/* write the sample and check for errors */
    if(fprintf(fp,"%f\n",x) < 1) {
        printf("\nError writing output file in SENDOUT\n");
        exit(1);
    }
}
```

LISTING 4.4 Functions **sendout(output)** and **getinput()** used to emulate real-time input/output using ASCII text data files (contained in GETSEND.C).


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <conio.h>
#include "wavfmt.h"
#include "rtdspc.h"

/* code to get samples from a WAV type file format */

/* getinput - get one sample from disk to simulate realtime input */

/* input WAV format header with null init */
   WAVE_HDR win = { "", 0L };
   CHUNK_HDR cin = { "", 0L };
   DATA_HDR din = { "", 0L };
   WAVEFORMAT wavin = { 1, 1, 0L, 0L, 1, 8 };

/* global number of samples in data set */
   unsigned long int number_of_samples = 0;

float getinput()
{

   static FILE *fp_getwav = NULL;
   static channel_number = 0;
   short int int_data[4];          /* max of 4 channels can be read */
   unsigned char byte_data[4];     /* max of 4 channels can be read */
   short int j;
   int i;

/* open input file if not done in previous calls */
   if(!fp_getwav) {
       char s[80];
       printf("\nEnter input .WAV file name ? ");
       gets(s);
       fp_getwav = fopen(s,"rb");
       if(!fp_getwav) {
           printf("\nError opening *.WAV input file in GETINPUT\n");
           exit(1);
       }

/* read and display header information */
       fread(&win,sizeof(WAVE_HDR),1,fp_getwav);
       printf("\n%c%c%c%c",
           win.chunk_id[0],win.chunk_id[1],win.chunk_id[2],win.chunk_id[3]);
       printf("\nChunkSize = %ld bytes",win.chunk_size);
```

LISTING 4.5 Function **getinput()** used to emulate real-time input using WAV format binary data files (contained in GETWAV.C). (*Continued*)

```
    if(strnicmp(win.chunk_id,"RIFF",4) != 0) {
        printf("\nError in RIFF header\n");
        exit(1);
    }

    fread(&cin,sizeof(CHUNK_HDR),1,fp_getwav);
    printf("\n");
    for(i = 0 ; i < 8 ; i++) printf("%c",cin.form_type[i]);
    printf("\n");
    if(strnicmp(cin.form_type,"WAVEfmt ",8) != 0) {
        printf("\nError in WAVEfmt header\n");
        exit(1);
    }

    if(cin.hdr_size != sizeof(WAVEFORMAT)) {
        printf("\nError in WAVEfmt header\n");
        exit(1);
    }

    fread(&wavin,sizeof(WAVEFORMAT),1,fp_getwav);
    if(wavin.wFormatTag != WAVE_FORMAT_PCM) {
        printf("\nError in WAVEfmt header - not PCM\n");
        exit(1);
    }
    printf("\nNumber of channels = %d",wavin.nChannels);
    printf("\nSample rate = %ld",wavin.nSamplesPerSec);
    printf("\nBlock size of data = %d bytes",wavin.nBlockAlign);
    printf("\nBits per Sample = %d\n",wavin.wBitsPerSample);

/* check channel number and block size are good */
    if(wavin.nChannels > 4 || wavin.nBlockAlign > 8) {
        printf("\nError in WAVEfmt header - Channels/BlockSize\n");
        exit(1);
    }

    fread(&din,sizeof(DATA_HDR),1,fp_getwav);
    printf("\n%c%c%c%c",

din.data_type[0],din.data_type[1],din.data_type[2],din.data_type[3]);
    printf("\nData Size = %ld bytes",din.data_size);

/* set the number of samples (global) */
    number_of_samples = din.data_size/wavin.nBlockAlign;
    printf("\nNumber of Samples per Channel = %ld\n",number_of_samples);

    if(wavin.nChannels > 1) {
     do {
        printf("\nError Channel Number [0..%d] - ",wavin.nChannels-1);
```

LISTING 4.5 (*Continued*)


```
        i = getche() - '0';
        if(i < (4-'0')) exit(1);
      } while(i < 0 || i >= wavin.nChannels);
      channel_number = i;
     }
    }

 /* read data until end of file */
    if(wavin.wBitsPerSample == 16) {
        if(fread(int_data,wavin.nBlockAlign,1,fp_getwav) != 1) {
            flush(); /* flush the output when input runs out */
            exit(1);
        }
        j = int_data[channel_number];
    }
    else {
        if(fread(byte_data,wavin.nBlockAlign,1,fp_getwav) != 1) {
            flush(); /* flush the output when input runs out */
            exit(1);
        }
        j = byte_data[channel_number];
        j ^= 0x80;
        j <<= 8;
    }

    return((float)j);
}
```

LISTING 4.5 (*Continued*)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "wavfmt.h"
#include "rtdspc.h"

/* code to send samples to a WAV type file format */

/* define BITS16 if want to use 16 bit samples */

/* sendout - send sample to disk to simulate realtime output */

    static FILE *fp_sendwav = NULL;
    static DWORD samples_sent = 0L; /* used by flush for header */
```

LISTING 4.6 Functions **sendout(output)** and **flush()** used to emulate real-time output using WAV format binary data files (contained in SENDWAV.C). (*Continued*)

```
/* WAV format header init */
    static WAVE_HDR wout = { "RIFF", 0L }; /* fill size at flush */
    static CHUNK_HDR cout = { "WAVEfmt " , sizeof(WAVEFORMAT) };
    static DATA_HDR dout = { "data" , 0L }; /* fill size at flush */
    static WAVEFORMAT wavout = { 1, 1, 0L, 0L, 1, 8 };

    extern WAVE_HDR win;
    extern CHUNK_HDR cin;
    extern DATA_HDR din;
    extern WAVEFORMAT wavin;

void sendout(float x)
{
    int BytesPerSample;
    short int j;

/* open output file if not done in previous calls */
    if(!fp_sendwav) {
        char s[80];
        printf("\nEnter output *.WAV file name ? ");
        gets(s);
        fp_sendwav = fopen(s,"wb");
        if(!fp_sendwav) {
            printf("\nError opening output *.WAV file in SENDOUT\n");
            exit(1);
        }
/* write out the *.WAV file format header */

#ifdef BITS16
        wavout.wBitsPerSample = 16;
        wavout.nBlockAlign = 2;
        printf("\nUsing 16 Bit Samples\n");
#else
        wavout.wBitsPerSample = 8;
#endif

        wavout.nSamplesPerSec = SAMPLE_RATE;
        BytesPerSample = (int)ceil(wavout.wBitsPerSample/8.0);
        wavout.nAvgBytesPerSec = BytesPerSample*wavout.nSamplesPerSec;

        fwrite(&wout,sizeof(WAVE_HDR),1,fp_sendwav);
        fwrite(&cout,sizeof(CHUNK_HDR),1,fp_sendwav);
        fwrite(&wavout,sizeof(WAVEFORMAT),1,fp_sendwav);
        fwrite(&dout,sizeof(DATA_HDR),1,fp_sendwav);
    }
/* write the sample and check for errors */

/* clip output to 16 bits */
    j = (short int)x;
```

LISTING 4.6 (*Continued*)


```
    if(x > 32767.0) j = 32767;
    else if(x < -32768.0) j = -32768;

#ifdef BITS16
    j ^= 0x8000;

    if(fwrite(&j,sizeof(short int),1,fp_sendwav) != 1) {
        printf("\nError writing 16 Bit output *.WAV file in SENDOUT\n");
        exit(1);
    }
#else
/* clip output to 8 bits */
    j = j >> 8;
    j ^= 0x80;

    if(fputc(j,fp_sendwav) == EOF) {
        printf("\nError writing output *.WAV file in SENDOUT\n");
        exit(1);
    }
#endif

    samples_sent++;
}

/* routine for flush - must call this to update the WAV header */

void flush()
{
    int BytesPerSample;

    BytesPerSample = (int)ceil(wavout.wBitsPerSample/8.0);
    dout.data_size=BytesPerSample*samples_sent;

    wout.chunk_size=
        dout.data_size+sizeof(DATA_HDR)+sizeof(CHUNK_HDR)+sizeof(WAVEFORMAT);

/* check for an input WAV header and use the sampling rate, if valid */
    if(strnicmp(win.chunk_id,"RIFF",4) == 0 && wavin.nSamplesPerSec != 0) {
        wavout.nSamplesPerSec = wavin.nSamplesPerSec;
        wavout.nAvgBytesPerSec = BytesPerSample*wavout.nSamplesPerSec;
    }

    fseek(fp_sendwav,0L,SEEK_SET);
    fwrite(&wout,sizeof(WAVE_HDR),1,fp_sendwav);
    fwrite(&cout,sizeof(CHUNK_HDR),1,fp_sendwav);
    fwrite(&wavout,sizeof(WAVEFORMAT),1,fp_sendwav);
    fwrite(&dout,sizeof(DATA_HDR),1,fp_sendwav);
}
```

LISTING 4.6 (*Continued*)

4.2 FILTERING TO REMOVE NOISE

Noise is generally unwanted and can usually be reduced by some type of filtering. Noise can be highly correlated with the signal or in a completely different frequency band, in which case it is uncorrelated. Some types of noise are impulsive in nature and occur relatively infrequently, while other types of noise appear as narrowband tones near the signal of interest. The most common type of noise is wideband thermal noise, which originates in the sensor or the amplifying electronic circuits. Such noise can often be considered white Gaussian noise, implying that the power spectrum is flat and the distribution is normal. The most important considerations in deciding what type of filter to use to remove noise are the type and characteristics of the noise. In many cases, very little is known about the noise process contaminating the digital signal and it is usually costly (in terms of time and/or money) to find out more about it. One method to study the noise performance of a digital system is to generate a model of the signal and noise and simulate the system performance in this ideal condition. System noise simulation is illustrated in the next two sections. The simulated performance can then be compared to the system performance with real data or to a theoretical model.

4.2.1 Gaussian Noise Generation

The function **gaussian** (shown in Listing 4.7) is used for noise generation and is contained in the FILTER.C source file. The function has no arguments and returns a single random floating-point number. The standard C library function **rand** is called to generate uniformly distributed numbers. The function **rand** normally returns integers from 0 to some maximum value (a defined constant, **RAND_MAX**, in ANSI implementations). As shown in Listing 4.7, the integer values returned by **rand** are converted to **float** values to be used by **gaussian**. Although the random number generator provided with most C compilers gives good random numbers with uniform distributions and long periods, if the random number generator is used in an application that requires truly random, uncorrelated sequences, the generator should be checked carefully. If the **rand** function is in question, a standard random number generator can be easily written in C (see Park and Miller, 1988). The function **gaussian** returns a zero mean random number with a unit variance and a Gaussian (or normal) distribution. It uses the Box-Muller method (see Knuth, 1981; or Press, Flannary, Teukolsky, and Vetterling, 1987) to map a pair of independent uniformly distributed random variables to a pair of Gaussian random variables. The function **rand** is used to generate the two uniform variables **v1** and **v2** from –1 to +1, which are transformed using the following statements:

```
r = v1*v1 + v2*v2;
fac = sqrt(-2.*log(r)/r);
gstore = v1*fac;
gaus = v2*fac;
```

The **r** variable is the radius squared of the random point on the (**v1, v2**) plane. In the **gaussian** function, the **r** value is tested to insure that it is always less than 1 (which it usually is), so that the region uniformly covered by (**v1, v2**) is a circle and so that **log(r)** is always negative and the argument for the square root is positive. The variables **gstore** and **gaus** are the resulting independent Gaussian random variables. Because **gaussian** must return one value at a time, the **gstore** variable is a **static** floating-point variable used to store the **v1*fac** result until the next call to **gaussian**.


```
/*************************************************************************

gaussian - generates zero mean unit variance Gaussian random numbers

Returns one zero mean unit variance Gaussian random numbers as a double.
Uses the Box-Muller transformation of two uniform random numbers to
Gaussian random numbers.

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

float gaussian()
{
    static int ready = 0;         /* flag to indicated stored value */
    static float gstore;          /* place to store other value */
    static float rconst1 = (float)(2.0/RAND_MAX);
    static float rconst2 = (float)(RAND_MAX/2.0);
    float v1,v2,r,fac,gaus;

/* make two numbers if none stored */
    if(ready == 0) {
        do {
            v1 = (float)rand() - rconst2;
            v2 = (float)rand() - rconst2;
            v1 *= rconst1;
            v2 *= rconst1;
            r = v1*v1 + v2*v2;
        } while(r > 1.0f);          /* make radius less than 1 */

/* remap v1 and v2 to two Gaussian numbers */
        fac = sqrt(-2.0f*log(r)/r);
        gstore = v1*fac;          /* store one */
        gaus = v2*fac;            /* return one */
        ready = 1;                /* set ready flag */
    }

    else {
        ready = 0;      /* reset ready flag for next pair */
        gaus = gstore;  /* return the stored one */
    }
    return(gaus);
}
```

LISTING 4.7 Function **gaussian()**.

The `static` integer variable `ready` is used as a flag to indicate if `gstore` has just been stored or if two new Gaussian random numbers should be generated.

4.2.2 Signal-to-Noise Ratio Improvement

One common application of digital filtering is *signal-to-noise ratio enhancement.* If the signal has a limited bandwidth and the noise has a spectrum that is broad, then a filter can be used to remove the part of the noise spectrum that does not overlap the signal spectrum. If the filter is designed to match the signal perfectly, so that the maximum amount of noise is removed, then the filter is called a *matched* or *Wiener filter.* Wiener filtering is briefly discussed in section 1.7.1 of chapter 1.

Figure 4.7 shows a simple example of filtering a single tone with added white noise. The MKGWN program (see Listing 4.8) was used to add Gaussian white noise with a standard deviation of 0.2 to a sine wave at a 0.05 f_s frequency as shown in Figure 4.7(a). The standard deviation of the sine wave signal alone can be easily found to be 0.7107. Because the standard deviation of the added noise is 0.2, the signal-to-noise ratio of the noisy signal is 3.55 or 11.0 dB. Figure 4.7(b) shows the result of applying the 35-tap lowpass FIR filter to the noisy signal. Note that much of the noise is still present but is smaller and has predominantly low frequency components. By lowpass filtering the 250 noise samples added to the sine wave separately, the signal-to-noise ratio of Figure 4.7(b) can be estimated to be 15 dB. Thus, the filtering operation improved the signal-to-noise ratio by 4 dB.

4.3 SAMPLE RATE CONVERSION

Many signal processing applications require that the output sampling rate be different than the input sampling rate. Sometimes one section of a system can be made more efficient if the sampling rate is lower (such as when simple FIR filters are involved or in data transmission). In other cases, the sampling rate must be increased so that the spectral details of the signal can be easily identified. In either case, the input sampled signal must be resampled to generate a new output sequence with the same spectral characteristics but at a different sampling rate. Increasing the sampling rate is called *interpolation* or *upsampling.* Reducing the sampling rate is called *decimation* or *downsampling.* Normally, the sampling rate of a band limited signal can be interpolated or decimated by integer ratios such that the spectral content of the signal is unchanged. By cascading interpolation and decimation, the sampling rate of a signal can be changed by any rational fraction, P/M, where P is the integer interpolation ratio and M is the integer decimation ratio. Interpolation and decimation can be performed using filtering techniques (as described in this section) or by using the fast Fourier transform (see section 4.4.2).

Decimation is perhaps the simplest resampling technique because it involves reducing the number of samples per second required to represent a signal. If the input signal is strictly band-limited such that the signal spectrum is zero for all frequencies above $f_s/(2M)$, then decimation can be performed by simply retaining every Mth sample and

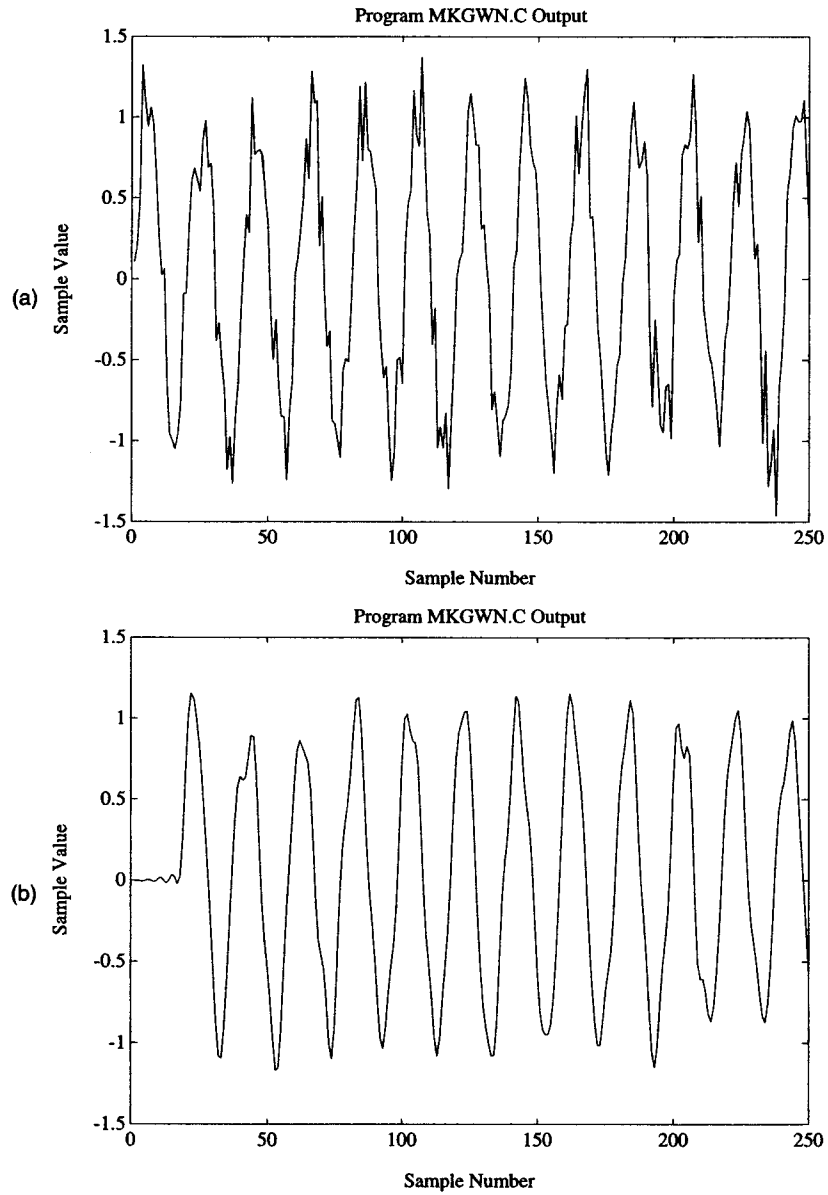

FIGURE 4.7 MKGWN program example output. Filtering a sine wave with added noise (frequency = 0.05). **(a)** Unfiltered version with Gaussian noise (standard deviation = 0.2). **(b)** Output after lowpass filtering with 35-point FIR filter.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "rtdspc.h"
#include "filter.h"

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

MKGWN.C - Gaussian Noise Filter Example

This program performs filters a sine wave with added Gaussian noise
It performs the filtering to implement a 35 point FIR filter
(stored in variable fir_lpf35) on an generated signal.
The filter is a LPF with 40 dB out of band rejection. The 3 dB
point is at a relative frequency of approximately .25*fs.

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

float sigma = 0.2;

void main()
{
 int         i, j;
 float x;
 static float hist[34];
 for(i = 0 ; i < 250 ; i++) {
     x = sin(0.05*2*PI*i) + sigma*gaussian();
     sendout(fir_filter(x,fir_lpf35,35,hist));
 }
}
```

LISTING 4.8 Program MKGWN to add Gaussian white noise to cosine wave and then perform FIR filtering.

discarding the $M - 1$ samples in between. Unfortunately, the spectral content of a signal above $f_s/(2M)$ is rarely zero, and the aliasing caused by the simple decimation almost always causes trouble. Even when the desired signal is zero above $f_s/(2M)$, some amount of noise is usually present that will alias into the lower frequency signal spectrum. Aliasing due to decimation can be avoided by lowpass filtering the signal before the samples are decimated. For example, when $M = 2$, the 35-point lowpass FIR filter introduced in section 4.1.2 can be used to eliminate almost all spectral content above $0.25f_s$ (the attenuation above $0.25\ f_s$ is greater than 40 dB). A simple decimation program could then be used to reduce the sampling rate by a factor of two. An IIR lowpass filter (discussed in section 4.1.3) could also be used to eliminate the frequencies above $f_s/(2M)$ as long as linear phase response is not required.

4.3.1 FIR Interpolation

Interpolation is the process of computing new samples in the intervals between existing data points. Classical interpolation (used before calculators and computers) involves estimating the value of a function between existing data points by fitting the data to a low-order polynomial. For example, *linear* (first-order) or *quadratic* (second-order) polynomial interpolation is often used. The primary attraction of polynomial interpolation is computational simplicity. The primary disadvantage is that in signal processing, the input signal must be restricted to a very narrow band so that the output will not have a large amount of aliasing. Thus, band-limited interpolation using digital filters is usually the method of choice in digital signal processing. Band-limited interpolation by a factor P:1 (see Figure 4.8 for an illustration of 3:1 interpolation) involves the following conceptual steps:

(1) Make an output sequence P times longer than the input sequence. Place the input sequence in the output sequence every P samples and place $P - 1$ zero values between each input sample. This is called *zero-packing* (as opposed to *zero-padding*). The zero values are located where the new interpolated values will appear. The effect of zero-packing on the input signal spectrum is to replicate the spectrum P times within the output spectrum. This is illustrated in Figure 4.8(a) where the output sampling rate is three times the input sampling rate.

(2) Design a lowpass filter capable of attenuating the undesired $P - 1$ spectra above the original input spectrum. Ideally, the passband should be from 0 to $f_s'/(2P)$ and the stopband should be from $f_s'/(2P)$ to $f_s'/2$ (where f_s' is the filter sampling rate that is P times the input sampling rate). A more practical interpolation filter has a transition band centered about $f_s'/(2P)$. This is illustrated in Figure 4.8(b). The passband gain of this filter must be equal to P to compensate for the inserted zeros so that the original signal amplitude is preserved.

(3) Filter the zero-packed input sequence using the interpolation filter to generate the final P:1 interpolated signal. Figure 4.8(c) shows the resulting 3:1 interpolated spectrum. Note that the two repeated spectra are attenuated by the stopband attenuation of the interpolation filter. In general, the stopband attenuation of the filter must be greater than the signal-to-noise ratio of the input signal in order for the interpolated signal to be a valid representation of the input.

4.3.2 Real-Time Interpolation Followed by Decimation

Figure 4.8(d) illustrates 2:1 decimation after the 3:1 interpolation, and shows the spectrum of the final signal, which has a sampling rate 1.5 times the input sampling rate. Because no lowpass filtering (other than the filtering by the 3:1 interpolation filter) is performed before the decimation shown, the output signal near $f_s''/2$ has an unusually shaped power spectrum due to the aliasing of the 3:1 interpolated spectrum. If this aliasing causes a problem in the system that processes the interpolated output signal, it can be

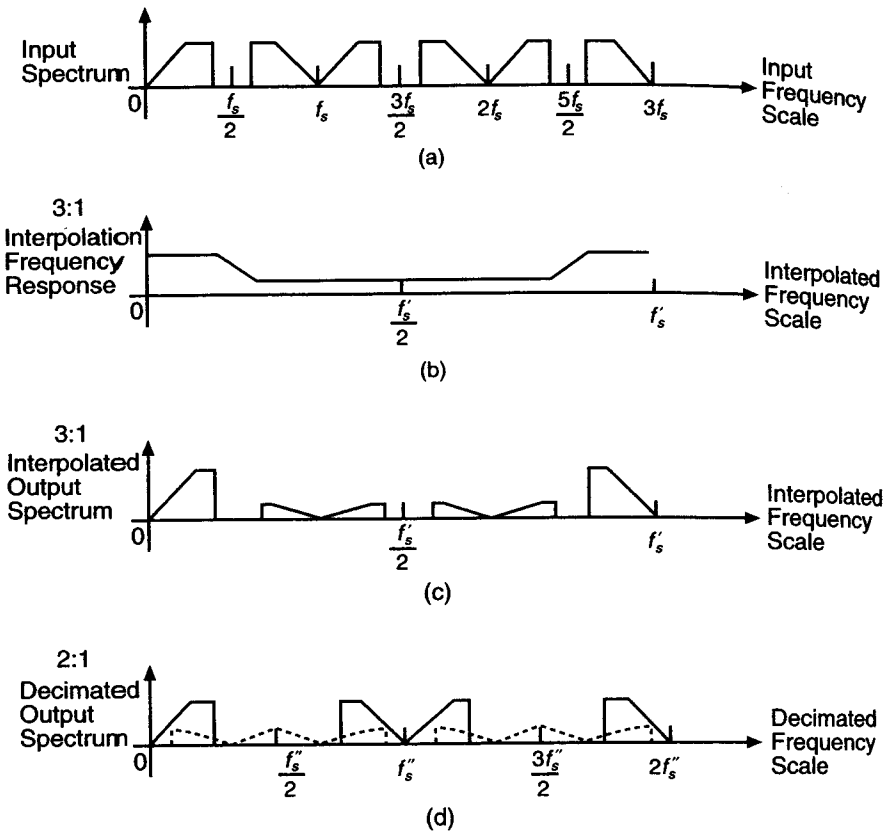

FIGURE 4.8 Illustration of 3:1 interpolation followed by 2:1 decimation. The aliased input spectrum in the decimated output is shown with a dashed line. **(a)** Example real input spectrum. **(b)** 3:1 interpolation filter response ($f_s' = 3f_s$). **(c)** 3:1 interpolated spectrum. **(d)** 2:1 decimated output ($f_s'' = f_s'/2$).

eliminated by either lowpass filtering the signal before decimation or by designing the interpolation filter to further attenuate the replicated spectra.

The interpolation filter used to create the interpolated values can be an IIR or FIR lowpass filter. However, if an IIR filter is used the input samples are not preserved exactly because of the nonlinear phase response of the IIR filter. FIR interpolation filters can be designed such that the input samples are preserved, which also results in some computational savings in the implementation. For this reason, only the implementation of FIR interpolation will be considered further. The FIR lowpass filter required for interpolation can be designed using the simpler windowing techniques. In this section, a Kaiser

window is used to design 2:1 and 3:1 interpolators. The FIR filter length must be odd so that the filter delay is an integer number of samples and the input samples can be preserved. The passband and stopband must be specified such that the center coefficient of the filter is unity (the filter gain will be P) and P coefficients on each side of the filter center are zero. This insures that the original input samples are preserved, because the result of all the multiplies in the convolution is zero, except for the center filter coefficient that gives the input sample. The other $P - 1$ output samples between each original input sample are created by convolutions with the other coefficients of the filter. The following passband and stopband specifications will be used to illustrate a P:1 interpolation filter:

Passband frequencies:	$0–0.8 f_s/(2P)$
Stopband frequencies:	$1.2 f_s/(2P)–0.5 f_s$
Passband gain:	P
Passband ripple:	< 0.03 dB
Stopband attenuation:	> 56 dB

The filter length was determined to be $16P - 1$ using Equation (4.2) (rounding to the nearest odd length) and the passband and stopband specifications. Greater stopband attenuation or a smaller transition band can be obtained with a longer filter. The interpolation filter coefficients are obtained by multiplying the Kaiser window coefficients by the ideal lowpass filter coefficients. The ideal lowpass coefficients for a very long odd length filter with a cutoff frequency of $f_s/2P$ are given by the following sinc function:

$$c_k = \frac{P \sin(k\pi / P)}{k\pi}. \tag{4.6}$$

Note that the original input samples are preserved, because the coefficients are zero for all $k = nP$, where n is an integer greater than zero and $c_0 = 1$. Very poor stopband attenuation would result if the above coefficients were truncated by using the $16P - 1$ coefficients where $|k| < 8P$. However, by multiplying these coefficients by the appropriate Kaiser window, the stopband and passband specifications can be realized. The symmetrical Kaiser window, w_k, is given by the following expression:

$$w_k = \frac{I_0\left\{\beta\sqrt{1-\left(\frac{2k}{N-1}\right)^2}\right\}}{I_0(\beta)}, \tag{4.7}$$

where $I_0(\beta)$ is a modified zero order Bessel function of the first kind, β is the Kaiser window parameter which determines the stopband attenuation and N in equation (4.7) is $16P + 1$. The empirical formula for β when A_{stop} is greater than 50 dB is $\beta = 0.1102*(A_{stop} - 8.71)$. Thus, for a stopband attenuation of 56 dB, $\beta = 5.21136$. Figure 4.9(a) shows the frequency response of the resulting 31-point 2:1 interpolation filter, and Figure 4.9(b) shows the frequency response of the 47-point 3:1 interpolation filter.

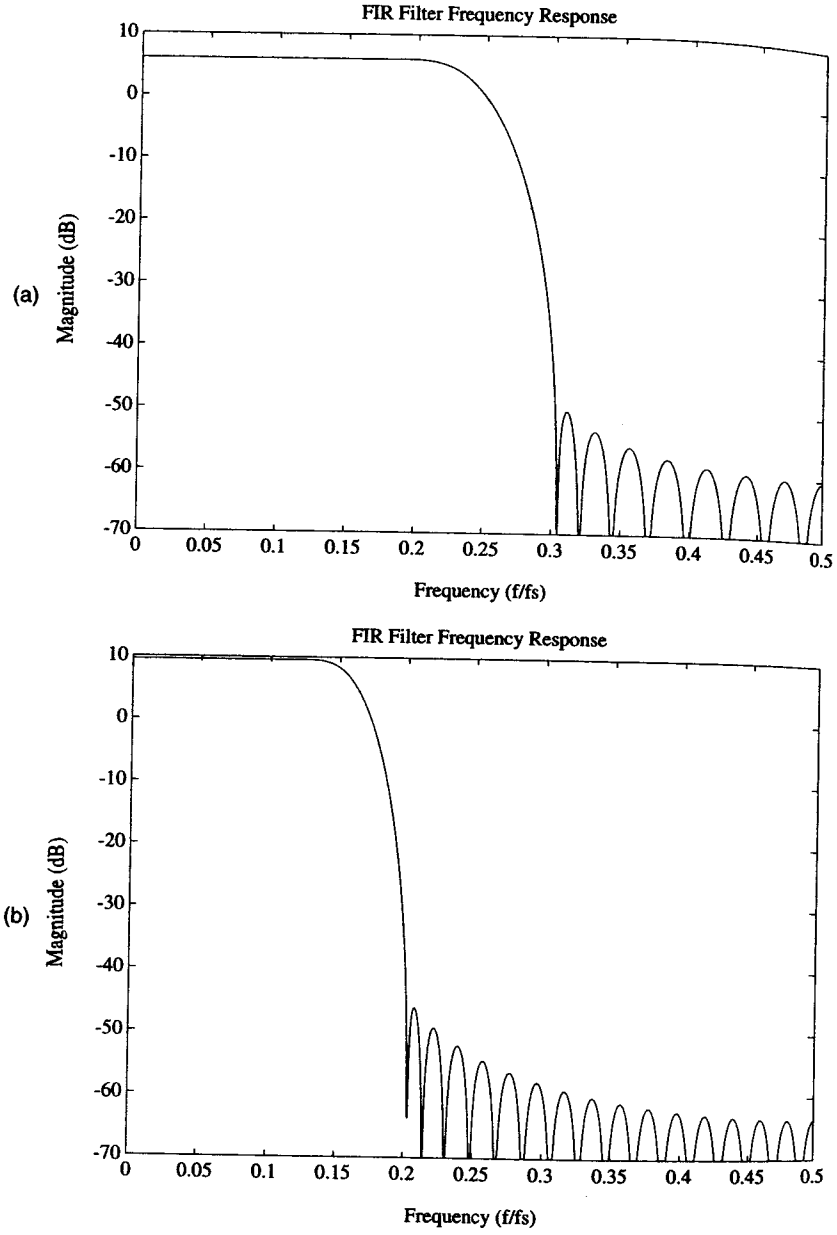

FIGURE 4.9 **(a)** Frequency response of 31-point FIR 2:1 interpolation filter (gain = 2 or 6 dB). **(b)** Frequency response of 47-point FIR 3:1 interpolation filter (gain = 3 or 9.54 dB).

4.3.3 Real-Time Sample Rate Conversion

Listing 4.9 shows the example interpolation program INTERP3.C, which can be used to interpolate a signal by a factor of 3. Two coefficient arrays are initialized to have the decimated coefficients each with 16 coefficients. Each of the coefficient sets are then used individually with the **`fir_filter`** function to create the interpolated values to be sent to **`sendout()`**. The original input signal is copied without filtering to the output every *P* sample (where *P* is 3). Thus, compared to direct filtering using the 47-point original filter, 15 multiplies for each input sample are saved when interpolation is performed using INTERP3. Note that the rate of output must be exactly three times the rate of input for this program to work in a real-time system.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "rtdspc.h"

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

INTERP3.C - PROGRAM TO DEMONSTRATE 3:1 FIR FILTER INTERPOLATION
            USES TWO INTERPOLATION FILTERS AND MULTIPLE CALLS TO THE
            REAL TIME FILTER FUNCTION fir_filter().

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

main()
{
    int i;
    float signal_in;
/* interpolation coefficients for the decimated filters */
    static float coef31[16],coef32[16];
/* history arrays for the decimated filters */
    static float hist31[15],hist32[15];

/* 3:1 interpolation coefficients, PB 0-0.133, SB 0.2-0.5 */
    static float interp3[47] = {
   -0.00178662, -0.00275941, 0.,  0.00556927,  0.00749929, 0.,
   -0.01268113, -0.01606336, 0.,  0.02482278,  0.03041984, 0.,
   -0.04484686, -0.05417098, 0.,  0.07917613,  0.09644332, 0.,
   -0.14927754, -0.19365910, 0.,  0.40682136,  0.82363913, 1.0,
    0.82363913,  0.40682136, 0., -0.19365910, -0.14927754, 0.,
    0.09644332,  0.07917613, 0., -0.05417098, -0.04484686, 0.,
    0.03041984,  0.02482278, 0., -0.01606336, -0.01268113, 0.,
```

LISTING 4.9 Example INTERP3.C program for 3:1 FIR interpolation. (*Continued*)

```
    0.00749928, 0.00556927, 0., -0.00275941, -0.00178662
            };

    for(i = 0 ; i < 16 ; i++) coef31[i] = interp3[3*i];

    for(i = 0 ; i < 16 ; i++) coef32[i] = interp3[3*i+1];

/* make three samples for each input */
    for(;;) {
        signal_in = getinput();
        sendout(hist31[7]); /* delayed input */
        sendout(fir_filter(signal_in,coef31,16,hist31));
        sendout(fir_filter(signal_in,coef32,16,hist32));
    }
}
```

LISTING 4.9 *(Continued)*

Figure 4.10 shows the result of running the INTERP3.C program on the WAVE3.DAT data file contained on the disk (the sum of frequencies 0.01, 0.02 and 0.4). Figure 4.10(a) shows the original data. The result of the 3:1 interpolation ratio is shown in Figure 4.10(b). Note that the definition of the highest frequency in the original data set ($0.4\,f_s$) is much improved, because in Figure 4.10(b) there are 7.5 samples per cycle of the highest frequency. The startup effects and the 23 sample delay of the 47-point interpolation filter is also easy to see in Figure 4.10(b) when compared to Figure 4.10(a).

FAST FILTERING ALGORITHMS

The FFT is an extremely useful tool for spectral analysis. However, another important application for which FFTs are often used is fast convolution. The formulas for convolution were given in chapter 1. Most often a relatively short sequence 20 to 200 points in length (for example, an FIR filter) must be convolved with a number of longer input sequences. The input sequence length might be 1,000 samples or greater and may be changing with time as new data samples are taken.

One method for computation given this problem is straight implementation of the time domain convolution equation as discussed extensively in chapter 4. The number of real multiplies required is $M * (N - M + 1)$, where N is the input signal size and M is the length of the FIR filter to be convolved with the input signal. There is an alternative to this rather lengthy computation method—the convolution theorem. The convolution theorem states that time domain convolution is equivalent to multiplication in the frequency domain. The convolution equation above can be rewritten in the frequency domain as follows:

$$Y(k) = H(k)\,X(k) \tag{4.8}$$

Because interpolation is also a filtering operation, fast interpolation can also be performed in the frequency domain using the FFT. The next section describes the implemen-

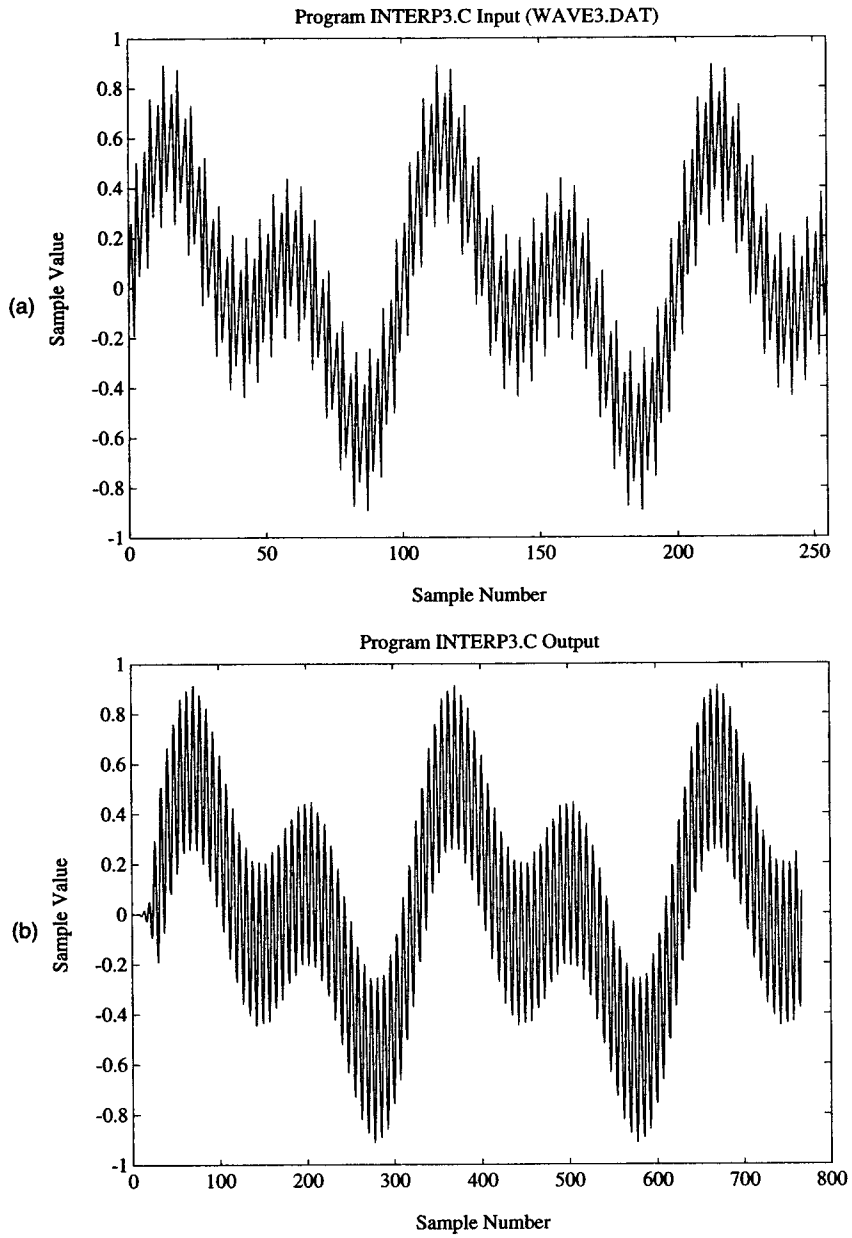

FIGURE 4.10 **(a)** Example of INTERP3 for 3:1 interpolation. Original WAVE3.DAT. **(b)** 3:1 interpolated WAVE3.DAT output.

tation of real-time filters using FFT fast convolution methods, and section 4.4.2 describes a real-time implementation of frequency domain interpolation.

4.4.1 Fast Convolution Using FFT Methods

Equation (4.8) indicates that if the frequency domain representations of $h(n)$ and $x(n)$ are known, then $Y(k)$ can be calculated by simple multiplication. The sequence $y(n)$ can then be obtained by inverse Fourier transform. This sequence of steps is detailed below:

(1) Create the array $H(k)$ from the impulse response $h(n)$ using the FFT.
(2) Create the array $X(k)$ from the sequence $x(n)$ using the FFT.
(3) Multiply H by X point by point thereby obtaining $Y(k)$.
(4) Apply the inverse FFT to $Y(k)$ in order to create $y(n)$.

There are several points to note about this procedure. First, very often the impulse response $h(n)$ of the filter does not change over many computations of the convolution equation. Therefore, the array $H(k)$ need only be computed once and can be used repeatedly, saving a large part of the computation burden of the algorithm.

Second, it must be noted that $h(n)$ and $x(n)$ may have different lengths. In this case, it is necessary to create two equal length sequences by adding zero-value samples at the end of the shorter of the two sequences. This is commonly called *zero filling* or *zero padding*. This is necessary because all FFT lengths in the procedure must be equal. Also, when using the radix 2 FFT all sequences to be processed must have a power of 2 length. This can require zero filling of both sequences to bring them up to the next higher value that is a power of 2.

Finally, in order to minimize circular convolution edge effects (the distortions that occur at computation points where each value of $h(n)$ does not have a matching value in $x(n)$ for multiplication), the length of $x(n)$ is often extended by the original length of $h(n)$ by adding zero values to the end of the sequence. The problem can be visualized by thinking of the convolution equation as a process of sliding a short sequence, $h(n)$, across a longer sequence, $x(n)$, and taking the sum of products at each translation point. As this translation reaches the end of the $x(n)$ sequence, there will be sums where not all $h(n)$ values match with a corresponding $x(n)$ for multiplication. At this point the output $y(n)$ is actually calculated using points from the beginning of $x(n)$, which may not be as useful as at the other central points in the convolution. This circular convolution effect cannot be avoided when using the FFT for fast convolution, but by zero filling the sequence its results are made predictable and repeatable.

The speed of the FFT makes convolution using the Fourier transform a practical technique. In fact, in many applications fast convolution using the FFT can be significantly faster than normal time domain convolution. As with other FFT applications, there is less advantage with shorter sequences and with very small lengths the overhead can create a penalty. The number of real multiply/accumulate operations required for fast convolution of an N length input sequence (where N is a large number, a power of 2 and real FFTs are used) with a fixed filter sequence is $2*N*[1 + 2*\log_2(N)]$. For example, when N is 1,024 and M is 100, fast convolution is as much as 2.15 times faster.

The program RFAST (see Listing 4.10) illustrates the use of the **`fft`** function for fast convolution (see Listing 4.11 for a C language implementation). Note that the inverse FFT is performed by swapping the real and imaginary parts of the input and output of the **`fft`** function. The overlap and save method is used to filter the continuous real-time input and generate a continuous output from the 1024 point FFT. The convolution problem is filtering with the 35-tap low pass FIR filter as was used in section 4.2.2. The filter is defined in the FILTER.H header file (variable **`fir_lpf35`**). The RFAST program can be used to generate results similar to the result shown in Figure 4.7(b).

(text continues on page 176)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "rtdspc.h"
#include "filter.h"

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

RFAST.C - Realtime fast convolution using the FFT

This program performs fast convolution using the FFT. It performs
the convolution required to implement a 35 point FIR filter
(stored in variable fir_lpf35) on an
arbitrary length realtime input. The filter is
a LPF with 40 dB out of band rejection. The 3 dB point is at a
relative frequency of approximately .25*fs.

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

/* FFT length must be a power of 2 */
#define FFT_LENGTH 1024
#define M 10              /* must be log2(FFT_LENGTH) */
#define FILTER_LENGTH 35

void main()
{
 int          i, j;
 float        tempflt;
 COMPLEX      *samp, *filt;
 static float input_save[FILTER_LENGTH];

/* power of 2 length of FFT and complex allocation */
  samp = (COMPLEX *) calloc(FFT_LENGTH, sizeof(COMPLEX));
  if(!samp){
```

LISTING 4.10 Program RFAST to perform real-time fast convolution using the overlap and save method. *(Continued)*

```
    exit(1);
 }

/* Zero fill the filter to the sequence length */
 filt = (COMPLEX *) calloc(FFT_LENGTH, sizeof(COMPLEX));
 if(!filt){
    exit(1);
 }

/* copy the filter into complex array and scale by 1/N for inverse FFT */
 tempflt = 1.0/FFT_LENGTH;
 for(i = 0 ; i < FILTER_LENGTH ; i++)
     filt[i].real = tempflt*fir_lpf35[i];

/* FFT the zero filled filter impulse response */
 fft(filt,M);

/* read in one FFT worth of samples to start, imag already zero */
 for(i = 0 ; i < FFT_LENGTH-FILTER_LENGTH ; i++)
    samp[i].real = getinput();

/* save the last FILTER_LENGTH points for next time */
 for(j = 0 ; j < FILTER_LENGTH ; j++, i++)
    input_save[j] = samp[i].real = getinput();

 while(1) {

/* do FFT of samples */
    fft(samp,M);

/* Multiply the two transformed sequences */
/* swap the real and imag outputs to allow a forward FFT instead of
inverse FFT */
    for(i = 0 ; i < FFT_LENGTH ; i++) {
      tempflt = samp[i].real * filt[i].real
                             - samp[i].imag * filt[i].imag;
      samp[i].real = samp[i].real * filt[i].imag
                             + samp[i].imag * filt[i].real;
      samp[i].imag = tempflt;
    }

/* Inverse fft the multiplied sequences */
    fft(samp,M);

/* Write the result out to a dsp data file */
/* because a forward FFT was used for the inverse FFT,
```

LISTING 4.10 *(Continued)*


```
the output is in the imag part */
    for(i = FILTER_LENGTH ; i < FFT_LENGTH ; i++) sendout(samp[i].imag);

/* overlap the last FILTER_LENGTH-1 input data points in the next FFT */
    for(i = 0; i < FILTER_LENGTH ; i++) {
      samp[i].real = input_save[i];
      samp[i].imag = 0.0;
    }

    for( ; i < FFT_LENGTH-FILTER_LENGTH ; i++) {
      samp[i].real = getinput();
      samp[i].imag = 0.0;
    }

/* save the last FILTER_LENGTH points for next time */
    for(j = 0 ; j < FILTER_LENGTH ; j++, i++) {
      input_save[j] = samp[i].real = getinput();
      samp[i].imag = 0.0;
    }
 }
}
```

LISTING 4.10 *(Continued)*

```
/*************************************************************************

fft - In-place radix 2 decimation in time FFT

Requires pointer to complex array, x and power of 2 size of FFT, m
(size of FFT = 2**m). Places FFT output on top of input COMPLEX array.

void fft(COMPLEX *x, int m)

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

void fft(COMPLEX *x,int m)
{
    static COMPLEX *w;            /* used to store the w complex array */
    static int mstore = 0;        /* stores m for future reference */
    static int n = 1;             /* length of fft stored for future */

    COMPLEX u,temp,tm;
    COMPLEX *xi,*xip,*xj,*wptr;

    int i,j,k,l,le,windex;

    double arg,w_real,w_imag,wrecur_real,wrecur_imag,wtemp_real;
```

LISTING 4.11 Radix 2 FFT function `fft(x,m)`. *(Continued)*

```
    if(m != mstore) {

/* free previously allocated storage and set new m */

        if(mstore != 0) free(w);
        mstore = m;
        if(m == 0) return;          /* if m=0 then done */

/* n = 2**m = fft length */

        n = 1 << m;
        le = n/2;

/* allocate the storage for w */

        w = (COMPLEX *) calloc(le-1,sizeof(COMPLEX));
        if(!w) {
            exit(1);
        }

/* calculate the w values recursively */

        arg = 4.0*atan(1.0)/le;          /* PI/le calculation */
        wrecur_real = w_real = cos(arg);
        wrecur_imag = w_imag = -sin(arg);
        xj = w;
        for (j = 1 ; j < le ; j++) {
            xj->real = (float)wrecur_real;
            xj->imag = (float)wrecur_imag;
            xj++;
            wtemp_real = wrecur_real*w_real - wrecur_imag*w_imag;
            wrecur_imag = wrecur_real*w_imag + wrecur_imag*w_real;
            wrecur_real = wtemp_real;
        }
    }

/* start fft */

    le = n;
    windex = 1;
    for (l = 0 ; l < m ; l++) {
        le = le/2;

/* first iteration with no multiplies */

        for(i = 0 ; i < n ; i = i + 2*le) {
            xi = x + i;
            xip = xi + le;
```

LISTING 4.11 (*Continued*)


```
            temp.real = xi->real + xip->real;
            temp.imag = xi->imag + xip->imag;
            xip->real = xi->real - xip->real;
            xip->imag = xi->imag - xip->imag;
            *xi = temp;
        }

/* remaining iterations use stored w */

        wptr = w + windex - 1;
        for (j = 1 ; j < le ; j++) {
            u = *wptr;
            for (i = j ; i < n ; i = i + 2*le) {
                xi = x + i;
                xip = xi + le;
                temp.real = xi->real + xip->real;
                temp.imag = xi->imag + xip->imag;
                tm.real = xi->real - xip->real;
                tm.imag = xi->imag - xip->imag;
                xip->real = tm.real*u.real - tm.imag*u.imag;
                xip->imag = tm.real*u.imag + tm.imag*u.real;
                *xi = temp;
            }
            wptr = wptr + windex;
        }
        windex = 2*windex;
    }

/* rearrange data by bit reversing */

    j = 0;
    for (i = 1 ; i < (n-1) ; i++) {
        k = n/2;
        while(k <= j) {
            j = j - k;
            k = k/2;
        }
        j = j + k;
        if (i < j) {
            xi = x + i;
            xj = x + j;
            temp = *xj;
            *xj = *xi;
            *xi = temp;
        }
    }
}
```

LISTING 4.11 (*Continued*)

4.4.2 Interpolation Using the FFT

In section 4.3.2 time domain interpolation was discussed and demonstrated using several short FIR filters. In this section, the same process is demonstrated using FFT techniques. The steps involved in 2:1 interpolation using the FFT are as follows:

(1) Perform an FFT with a power of 2 length (N) which is greater than or equal to the length of the input sequence.
(2) Zero pad the frequency domain representation of the signal (a complex array) by inserting $N - 1$ zeros between the positive and negative half of the spectrum. The Nyquist frequency sample output of the FFT (at the index $N/2$) is divided by 2 and placed with the positive and negative parts of the spectrum, this results in a symmetrical spectrum for a real input signal.
(3) Perform an inverse FFT with a length of $2N$.
(4) Multiply the interpolated result by a factor of 2 and copy the desired portion of the result that represents the interpolated input, this is all the inverse FFT samples if the input length was a power of 2.

Listing 4.12 shows the program INTFFT2.C that performs 2:1 interpolation using the above procedure and the **fft** function (shown in Listing 4.11). Note that the inverse FFT is performed by swapping the real and imaginary parts of the input and output of the **fft** function. Figure 4.11 shows the result of using the INTFFT2 program on the 128 samples of the WAVE3.DAT input file used in the previous examples in this chapter (these 256 samples are shown in detail in Figure 4.10(a)). Note that the output length is twice as large (512) and more of the sine wave nature of the waveform can be seen in the interpolated result. The INTFFT2 program can be modified to interpolate by a larger power of 2 by increasing the number of zeros added in step (2) listed above. Also, because the FFT is employed, frequencies as high as the Nyquist rate can be accurately interpolated. FIR filter interpolation has a upper frequency limit because of the frequency response of the filter (see section 4.3.1).

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "rtdspc.h"

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

INTFFT2.C - Interpolate 2:1 using FFT

Generates 2:1 interpolated time domain data.

*************************************************************************/
```

LISTING 4.12 Program INTFFT2.C used to perform 2:1 interpolation using the FFT. (*Continued*)


```
#define LENGTH 256
#define M 8             /* must be log2(FFT_LENGTH) */

main()
{
    int         i;
    float        temp;
    COMPLEX     *samp;

/* allocate the complex array (twice as long) */
    samp = (COMPLEX *) calloc(2*LENGTH, sizeof(COMPLEX));
    if(!samp) {
        printf("\nError allocating fft memory\n");
        exit(1);
    }

/* copy input signal to complex array and do the fft */
    for (i = 0; i < LENGTH; i++) samp[i].real = getinput();

    fft(samp,M);

/* swap the real and imag to do the inverse fft */
    for (i = 0; i < LENGTH; i++) {
        temp = samp[i].real;
        samp[i].real = samp[i].imag;
        samp[i].imag = temp;
    }

/* divide the middle frequency component by 2 */
    samp[LENGTH/2].real = 0.5*samp[LENGTH/2].real;
    samp[LENGTH/2].imag = 0.5*samp[LENGTH/2].imag;

/* zero pad and move the negative frequencies */
    samp[3*LENGTH/2] = samp[LENGTH/2];
    for (i = LENGTH/2 + 1; i < LENGTH ; i++) {
        samp[i+LENGTH] = samp[i];
        samp[i].real = 0.0;
        samp[i].imag = 0.0;
    }

/* do inverse fft by swapping input and output real & imag */
    fft(samp,M+1);

/* copy to output and multiply by 2/(2*LENGTH) */
    temp = 1.0/LENGTH;
    for (i=0; i < 2*LENGTH; i++) sendout(temp*samp[i].imag);
}
```

LISTING 4.12 (*Continued*)

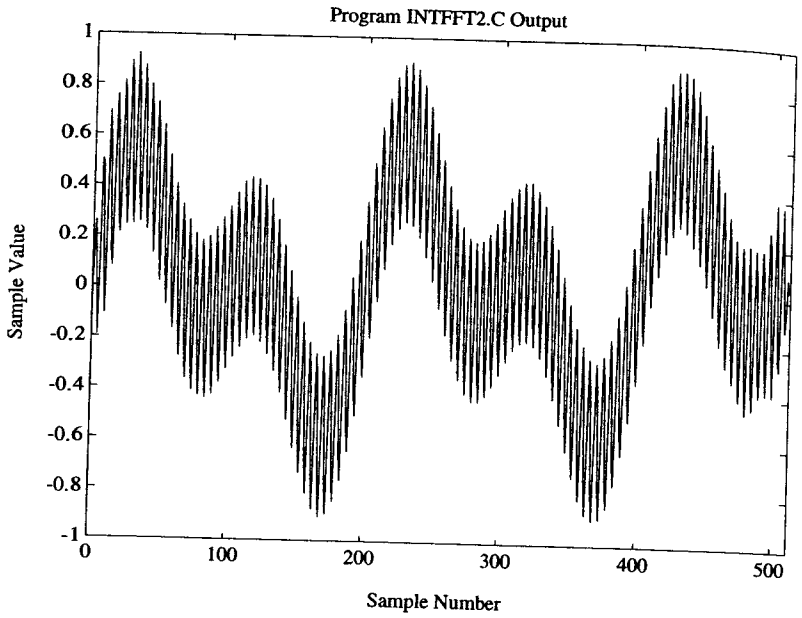

FIGURE 4.11 Example use of the INTFFT2 program used to interpolate WAVE3 signal by a 2:1 ratio.

4.5 OSCILLATORS AND WAVEFORM SYNTHESIS

The generation of pure tones is often used to synthesize new sounds in music or for testing DSP systems. The basic oscillator is a special case of an IIR filter where the poles are on the unit circle and the initial conditions are such that the input is an impulse. If the poles are moved outside the unit circle, the oscillator output will grow at an exponential rate. If the poles are placed inside the unit the circle, the output will decay toward zero. The state (or history) of the second order section determines the amplitude and phase of the future output. The next section describes the details of this type of oscillator. Section 4.5.2 considers another method to generate periodic waveforms of different frequencies —the wave table method. In this case any period waveform can be used to generate a fundamental frequency with many associated harmonics.

4.5.1 IIR Filters as Oscillators

The impulse response of a continuous time second order oscillator is given by

$$y(t) = e^{-dt}\frac{\sin(\omega t)}{\omega}. \tag{4.9}$$

If $d > 0$ then the output will decay toward zero and the peak will occur at

$$t_{peak} = \frac{\tan^{-1}(\omega/d)}{\omega}. \tag{4.10}$$

The peak value will be

$$y(t_{peak}) = \frac{e^{-dt_{peak}}}{\sqrt{d^2+\omega^2}}. \tag{4.11}$$

A second-order difference can be used to generate a response that is an approximation of this continuous time output. The equation for a second-order discrete time oscillator is based on an IIR filter and is as follows:

$$y_{n+1} = c_1 y_n - c_2 y_{n-1} + b_1 x_n, \tag{4.12}$$

where the x input is only present for $t = 0$ as an initial condition to start the oscillator and

$$c_1 = 2e^{-d\tau}\cos(\omega\tau)$$
$$c_2 = e^{-2d\tau}$$

where τ is the sampling period ($1/f_s$) and ω is 2π times the oscillator frequency.

The frequency and rate of change of the envelope of the oscillator output can be changed by modifying the values of d and ω on a sample by sample basis. This is illustrated in the OSC program shown in Listing 4.13. The output waveform grows from a peak value of 1.0 to a peak value of 16000 at sample number 5000. After sample 5000 the envelope of the output decays toward zero and the frequency is reduced in steps every 1000 samples. A short example output waveform is shown in Figure 4.12.

4.5.2 Table-Generated Waveforms

Listing 4.14 shows the program WAVETAB.C, which generates a fundamental frequency at a particular musical note given by the variable **key**. The frequency in Hertz is related to the integer **key** as follows:

$$f = 440 \cdot 2^{key/12} \tag{4.13}$$

Thus, a **key** value of zero will give 440 Hz, which is the musical note A above middle C. The WAVETAB.C program starts at a key value of –24 (two octaves below A) and steps through a chromatic scale to key value 48 (4 octaves above A). Each sample output value is calculated using a linear interpolation of the 300 values in the table **gwave**. The 300 sample values are shown in Figure 4.13 as an example waveform. The **gwave** array is 301 elements to make the interpolation more efficient. The first element (0) and the last element (300) are the same, creating a circular interpolated waveform. Any waveform can be substituted to create different sounds. The amplitude of the output is controlled by the **env** variable, and grows and decays at a rate determined by **trel** and **amp** arrays.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "rtdspc.h"

    float osc(float,float,int);
    float rate,freq;
    float amp = 16000;

void main()
{
    long int i,length = 100000;

/* calculate the rate required to get to desired amp in 5000 samples */
    rate = (float)exp(log(amp)/(5000.0));

/* start at 4000 Hz */
    freq = 4000.0;

/* first call to start up oscillator */
    sendout(osc(freq,rate,-1));
/* special case for first 5000 samples to increase amplitude */
    for(i = 0 ; i < 5000 ; i++)
        sendout(osc(freq,rate,0));

/* decay the osc 10% every 5000 samples */
    rate = (float)exp(log(0.9)/(5000.0));

    for( ; i < length ; i++) {
        if((i%1000) == 0) {          /* change freq every 1000 samples */
            freq = 0.98*freq;
            sendout(osc(freq,rate,1));
        }
        else {                       /* normal case */
            sendout(osc(freq,rate,0));
        }
    }
    flush();
}

/* Function to generate samples from a second order oscillator
        rate = envelope rate of change parameter (close to 1).
    change_flag = indicates that frequency and/or rate have changed.
*/

float osc(float freq,float rate,int change_flag)
{
/* calculate this as a static so it never happens again */
    static float two_pi_div_sample_rate = (float)(2.0 * PI / SAMPLE_RATE);
    static float y1,y0,a,b,arg;
    float out,wosc;
```

LISTING 4.13 Program OSC to generate a sine wave signal with a variable frequency and envelope using a second-order IIR section. (*Continued*)


```
/* change_flag:
    -1 = start new sequence from t=0
     0 = no change, generate next sample in sequence
     1 = change rate or frequency after start
*/
    if(change_flag != 0) {
/* assume rate and freq change every time */
        wosc = freq * two_pi_div_sample_rate;
        arg = 2.0 * cos(wosc);
        a = arg * rate;
        b = -rate * rate;

        if(change_flag < 0) {    /* re-start case, set state variables */
            y0 = 0.0f;
            return(y1 = rate*sin(wosc));
        }
    }
/* make new sample */
    out = a*y1 + b*y0;
    y0 = y1;
    y1 = out;
    return(out);
}
```

LISTING 4.13 (*Continued*)

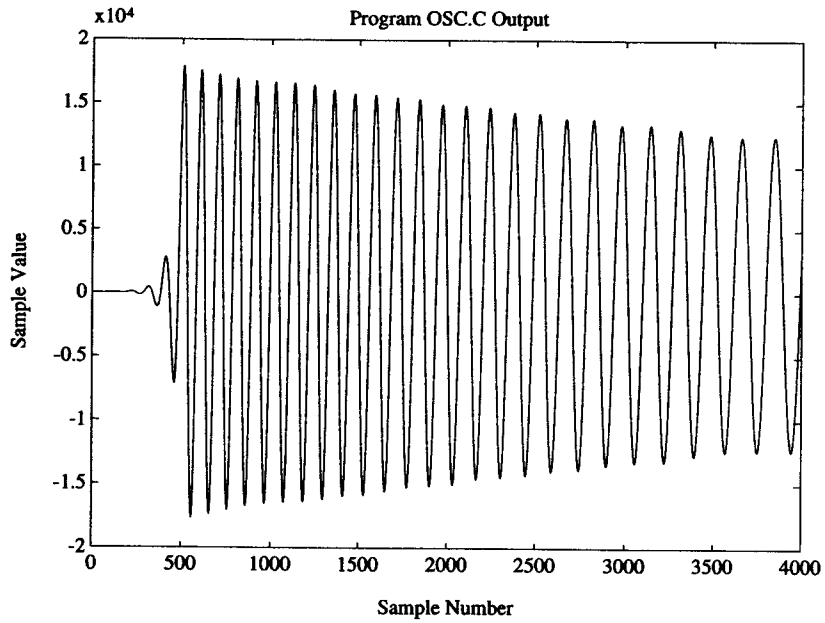

FIGURE 4.12 Example signal output from the OSC.C program (modified to reach peak amplitude in 500 samples and change frequency every 500 sample for display purposes).

```
#include <stdlib.h>
#include <math.h>
#include "rtdspc.h"
#include "gwave.h"          /* gwave[301] array */

/* Wavetable Music Generator 4-20-94 PME */

    int key;

void main()
{
    int t,told,ci,k;
    float ampold,rate,env,wave_size,dec,phase,frac,delta,sample;
    register long int i,endi;
    register float sig_out;

    static float trel[5] = {    0.02 ,    0.14,     0.6, 1.0, 0.0 };
    static float amps[5] = { 15000.0 , 10000.0, 4000.0, 10.0, 0.0 };
    static float rates[10];
    static int tbreaks[10];

    wave_size = 300.0;      /* dimension of original wave */
    endi = 96000;             /* 2 second notes */

    for(key = -24 ; key < 48 ; key++) {

/* decimation ratio for key semitones down */
        dec = powf(2.0,0.0833333333*(float)key);

/* calculate the rates required to get the desired amps */
        i = 0;
        told = 0;
        ampold = 1.0;          /* always starts at unity */
        while(amps[i] > 1.0) {
            t = trel[i]*endi;
            rates[i] = expf(logf(amps[i]/ampold)/(t-told));
            ampold = amps[i];
            tbreaks[i] = told = t;
            i++;
        }

        phase = 0.0;
        rate = rates[0];
        env = 1.0;
        ci = 0;
        for(i = 0 ; i < endi ; i++) {
/* calculate envelope amplitude */
```

LISTING 4.14 Program WAVETAB to generate periodic waveform at any frequency. (*Continued*)


```
            if(i == tbreaks[ci]) rate = rates[++ci];
            env = rate*env;
/* determine interpolated sample value from table */
            k = (int)phase;
            frac = phase - (float)k;
            sample = gwave[k];
            delta = gwave[k+1] - sample; /* possible wave_size+1 access */
            sample += frac*delta;
/* calculate output and send to DAC */
            sig_out = env*sample;
            sendout(sig_out);
/* calculate next phase value */
            phase += dec;
            if(phase >= wave_size) phase -= wave_size;
        }
    }
    flush();
}
```

LISTING 4.14 (*Continued*)

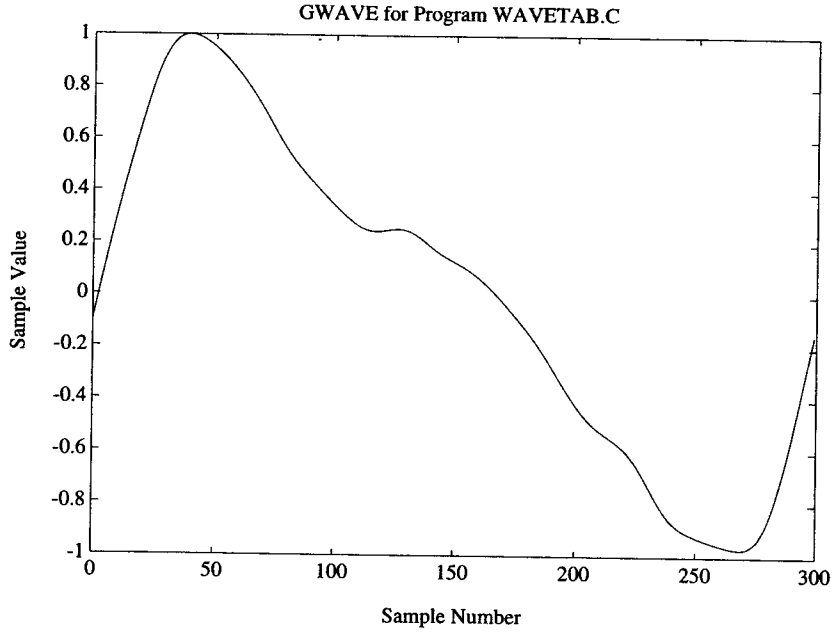

FIGURE 4.13 Example waveform (`gwave[301]` array) used by program WAVETAB.

4.6 REFERENCES

ANTONIOU, A. (1979). *Digital Fitters: Analysis and Design.* New York: McGraw-Hill.

BRIGHAM, E.O. (1988). *The Far Fourier Transform and Its Applications.* Englewood Cliffs, NJ: Prentice Hall.

CROCHIERE, R.E. and RABINER, L.R. (1983). *Multirate Digital Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

ELIOTT, D.F. (Ed.). (1987). *Handbook of Digital Signal Processing.* San Diego, CA: Academic Press.

EMBREE, P. and KIMBLE B. (1991). *C Language Algorithms for Digital Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

GHAUSI, M.S. and LAKER, K.R. (1981). *Modern Filter Design: Active RC and Switched Capacitor.* Englewood Cliffs, NJ: Prentice Hall.

IEEE DIGITAL SIGNAL PROCESSING COMMITTEE (Ed.). (1979). *Programs for Digital Signal Processing.* New York: IEEE Press.

JOHNSON, D.E., JOHNSON, J.R. and MOORE, H.P. (1980). *A Handbook of Active Filters.* Englewood Cliffs, NJ: Prentice Hall.

JONG, M.T. (1992). *Methods of Discrete Signal and System Analysis.* New York: McGraw-Hill.

KAISER, J. F. and SCHAFER, R. W. (Feb. 1980). On the Use of the /0-Sinh Window for Spectrum Analysis. *IEEE Transactions on Acoustics, Speech, and Signal Processing, (ASSP-28)* (1), 105–107.

KNUTH, D.E. (1981). *Seminumerical Algorithms, The Art of Computer Programming, Vol. 2.* (2nd ed.). Reading, MA: Addison-Wesley.

MCCLELLAN, J., PARKS, T. and RABINER, L.R. (1973). A Computer Program for Designing Optimum FIR Linear Phase Digital Filters. *IEEE Transactions on Audio and Electro-acoustics, AU-21.* (6), 506–526.

MOLER, C., LITTLE, J. and BANGERT, S. (1987). *PC-MATLAB User's Guide.* Natick, MA: The Math Works.

MOSCHUYTZ, G.S. and HORN, P. (1981). *Active Filter Design Handbook.* New York: John Wiley & Sons.

OPPENHEIM, A. and SCHAFER, R. (1975). *Digital Signal Processing,* Englewood Cliffs, NJ: Prentice Hall.

OPPENHEIM, A. and SCHAFER, R. (1989). *Discrete-time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

PAPOULIS, A. (1984). *Probability, Random Variables and Stochastic Processes,* (2nd ed.). New York: McGraw-Hill.

PARK, S.K. and MILLER, K.W. (Oct. 1988). Random Number Generators: Good Ones Are Hard to Find. *Communications of the ACM, (31)* (10).

PARKS, T.W. and BURRUS, C.S. (1987). *Digital Filter Design.* New York: John Wiley & Sons.

PRESS W.H., FLANNERY, B.P., TEUKOLSKY, S.A. and VETTERLING, W.T. (1987). *Numerical Recipes.* New York: Cambridge Press.

RABINER, L. and GOLD, B. (1975). *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

STEARNS, S. and DAVID, R. (1988). *Signal Processing Algorithms.* Englewood Cliffs, NJ: Prentice Hall.

VAN VALKENBURG, M.E. (1982). *Analog Filter Design.* New York: Holt, Rinehart and Winston.

ZVEREV, A. I. (1967). *Handbook of Filter Synthesis.* New York: John Wiley & Sons.