A Simple Interactive Program for Real-Time FIR Digital Filters Written in the C-Language

James E. Cross
Southern University

Abstract

Introduction

The main design task in using a digital signal processor (DSP) for the real-time processing of signals is that of algorithm development. An algorithm must be developed that will transform the signal in a manner to meet the design specifications. This paper is primarily concerned with the design of such an algorithm that is user friendly. The fundamentals of real-time digital signal processing will first be presented. A discussion of possible computer languages for digital signal processors will be given. The approach to developing a user friendly algorithm will be divided into two parts, that of first developing a non-real-time algorithm as a prototype followed by a real-time algorithm. Finally, some test results will be given, followed by some concluding remarks.

Some Fundamentals of Real-time Digital Signal Processing

The main components of a real-time digital signal processor is an analog to digital (A/D) converter, a computer and a digital to analog (D/A) converter. A number of vendors sell analog interface boards(AIB) that include an A/D converter and a D/A converter. Also included is an anti-aliasing filter and signal reconstruction circuitry. It is possible to use a sound card for this purpose. For the sampled analog signal to be accurately reproduced, the requirement is that the signal must be sampled at a rate that is greater than twice the highest frequency present in the signal. This is called the Nyquist rate. Since the highest frequency for sound is about 20KHz, to be certain of reproducing all frequencies accurately, the signal must be sampled at a rate greater than 40KHz. A signal with frequencies going only to 8 KHz is usually considered adequate for telephone quality sound whereas a CD player is expected to responded to the highest possible sound frequencies. A 44.1 Khz sampling rate is used for CD- quality sound. Some vendors use a sampling rate of 48 KHz for their analog interfacing boards. There has been considerable interest recently in the direct sampling of radio signals. This is as opposed to having the radio signal demodulate so that the sound signal is recovered and then sampling the lower frequency sound signal. Considering the Nyquist sampling rate, a 10 MHz signal must be sampled at a rate above 20 MHz.

Another important consideration, especially for filter design, is the speed and memory requirements of the computer to execute the associated algorithm. Several vendors supply special computers or processors that are designed specifically for digital signal processing. These are called digital signal processors or DSPs. The convolution function is the most common function used in digital filters. The convolution function is written as y = h*x where h,

y and x are vectors (row or column matrices). The convolution process is executed directly for a finite impulse response (FIR) filter. In considering a 21 tap (or state) low-pass filter as an example, h = h(n) = sin(6.2832*fl*n)/(3.1416*n) where fl is the normalized pass-band cutoff frequency. For an ideal filter, n goes from minus infinity to plus infinity which of course is not practical. A window function is used that in some way truncates n to some acceptable value, N, which is the number of taps. Thus, for our example, N = 21. There are several popular windowing functions. For the algorithm presented in this paper, the Hamming window was used. It is given as w = 0.54 + 0.46cos(6.2832*n)/N. The x term is the input from the A/D converter. There must be the same number of elements of x as there are elements of h when performing the convolution . The signal is sampled and stored in memory as x[0], x[1], x[2], ..., x[N-1]. Performing the convolution is a multiplication and accumulation process. A major difference between a conventional processor and a The speed and cost of computers are now such that digital methods are rapidly replacing analog methods for the real-time processing of signals. A typical real-time digital processing system will first use an analog to digital converter to change the signal to a digital form. An algorithm will be used to manipulate the signal in some desired manner. The signal will then be changed back into an analog form. Given the digital signal processing equipment, the main task of the designer is that of developing an algorithm to transform the signal to meet the design specifications. This paper describes a simple interactive algorithm written in the C-language that will permit the user to easily design and implement real-time finite impulse response (FIR) digital filters. DSP is that a DSP is specifically designed to efficiently execute the convolution function. Many DSPs have three separate memory locations and three separate buses, one for the instructions and two for data. Typically, the h values are loaded into one of the data memory sections and the values of x are loaded into the other data memory section. Most DSPs have a special instruction for performing the convolution. It is the multiply and accumulate instruction or MAC. It is very powerful in that one value of h and one value of x can be fetched from memory simultaneously, multiplied and stored back into the accumulator. Using a repeat instruction with the MAC instruction, a given value for y (or y[n]) can be efficiently calculated. To handle the process of having the proper values of x with h multiplied, some DSPs will automatically shift the old values of x in memory as new sampled values are acquired while other DSPs will have a special register to handle this function.

The speed of the processor must be such that it can complete the execution of the algorithm by the time a new sample value is acquired. Special techniques have been developed to speed up the execution of the algorithm. This includes the use of the Fast Fourier Transform (FFT) and the use of look-up tables.

Computer Languages for real-time Digital Signal Processing

Computers can be programmed at three levels, using machine language, assembly language or using a high-level language. There are trade-offs in considering the language to be used. Machine language is the most efficient but the most tedious and time consuming. Assembly language and high-level languages must ultimately be converter to machine language in that machine language code is the only instructions the computer really understands. Assembly language instructions are, for the most part, converted on a one-to-one basis to machine

language. It uses words that are easily recognized such as ADD for add and SUB for subtract. Assembly programming is often the language of choice for the real-time programming of computers. However, the down side to using assembly language is its lack of transportability from one type of DSP to the other and the fact that you must write your own algorithms for such functions as the sine, cosine, exponential, square root, etc. High-level languages such as C, FORTRAN or MATLAB are designed with the user in mind so as to make programming easy and efficient. The down side to using such high-level languages is that although programming is efficient, the compiler may not efficiently translate the instructions into machine language. A major advantage in using a high-level language is that programs will be portable from one DSP type to the other. The portability problem is solved by vendors writing compilers that will operate on standard C instructions and translated them into the unique code for their DSP. Each DSP type will require a unique compiler. The algorithm being presented in this paper was written in the C language. A Texas Instruments compiler for the Texas Instruments TMS320C30 DSP was used for the real-time processing on this DSP.

The Non-Real-Time User Friendly FIR DSP Algorithm

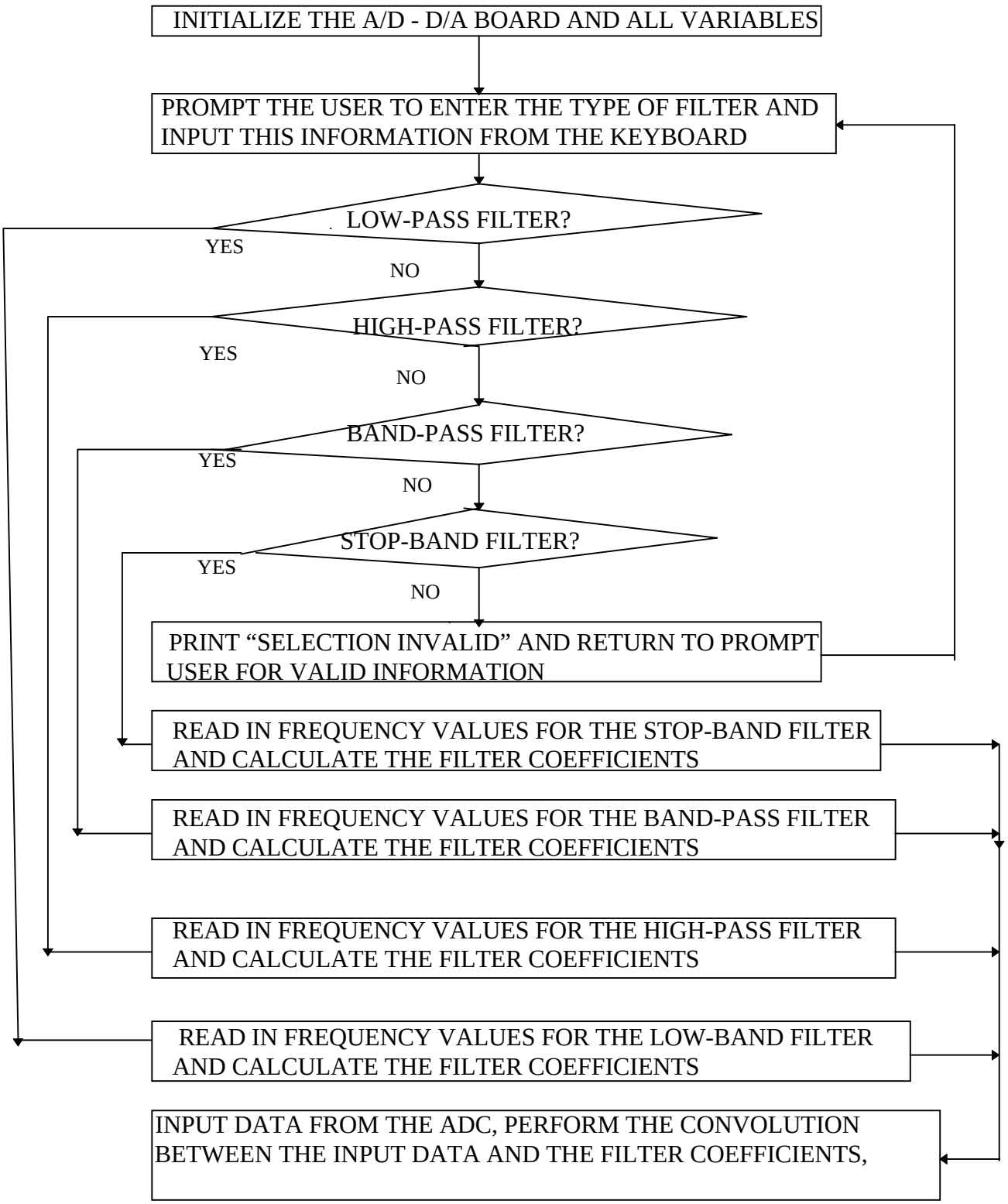
The algorithm used was written in standard C and was first tested in non-real-time. The program is to implement low-pass, high-pass, band-pass and stop-band FIR filters using the Hamming window. The equations for these filters are readily available in text books. It is menu driven, prompting the user for the type of filter and the relevant cutoff frequencies. A fixed value of N equal to 21 was used for this presentation. However, the program can easily be rewritten so that the value of N can be selected by the user. The values of the impulse response coefficients are generated and written to the screen. In calculating these values, a causal filter was used by using a formula that shifted the indices (N-1)/2 units to the right so that the first value of h was h[0] rather than h[-(n-1)/2). A test sine wave signal was generated in software. The program prompted the user for the frequency of the test signal. Typically, three or four cycles of the test signal were generated and written out for viewing. The non-real-time program is not presented. It includes much of the same code that is presented below for the real-time algorithm.

The real-time User Friendly FIR DSP Program

For the real-time program, the non-real-time program was modified mainly by adding an I/O section to input and output real values of the signal rather than generating a test signal. The flow chart for this program is shown below. The Texas Instrument TMS320C30 Banshee DSP board purchased from Atlanta Signal Processors, Inc. (ASPI) was used along with their AD16 A-D/D-A Interface Board. The basic I/O section was written and supplied by ASPI. The documentation of the basic program easily facilitated the adding of the algorithm for filtering the signal. The program as presented used 21 taps (N equal 21). However, it was tested with different values of N. The program is written for a steady state number of 20 old values of x retained in memory. The value of y being calculated is always y[20]. The values of x used are x[0], x[1], ... x[20]. A sampling rate of 48 KHz was used. The new value of x acquired is called x[20]. As a new value of x is acquired, all of the old values are shifted with the oldest value discarded. This is the same procedure that was performed with the non-real-time algorithm. The non-real-time algorithm was

changed so that instead of writing the value of y to the computer screen, the output is written to the D/A converter on the analog interface board. An oscilloscope and an amplifier with speaker

FLOW CHART FOR THE REAL-TIME FIR DSP ALGORITHM



was connected to the analog interface board. The program is as follows:

```
 /**********************************
WE ARE MAKING USE OF THE BASIC I/O ECHOING PROGRAM  SUPPLIED BY
ATLANTA SIGNAL PROCESSORS, INC. (ASPI).
 THE COMMENTS IMMEDIATELY BELOW REFER TO THE ASPI I/O BOARD AND TO
ASPI DOCUMENTATION.

NOTE: As is, this program uses channel 0 and timer 0. Timer 0 must be set from the PC side.
(This is important information.  If you don't understand what timer to use, look at the AD16
appendix) To compile:  cl30 ad16_c.c  To run (from ashell):   ad16_c :C0=nnnn where nnnnn is
the sample rate to set.  */

#define AD_CONTROL 0x40000      /* Control for AD16 (p D.5) */
#define RINT1 0x4000 /*Mask for RINT1 in status reg. (p9.16)*/

#define RINT2 0x2000 /*Mask for RINT2 in status reg.(p9.16)*/
#include <stdio.h >
#include <math.h>
int *status=(int *)0x80580d;/*Banshee status register(p 9.16) */
int *chan0  = (int *) 0x804000; /* AD16 chan0 data reg.(p D.5) */
int *chan1  = (int *) 0x804001; /* AD16 chan1 data reg.(p D.5) */
int *chan0_cnt = (int *) 0x804004; /* AD16 chan0 control register
(p D.5) */
int *chan1_cnt= int *) 0x804005; /* AD16 chan1 control reg.D.5)*/
main()
{
     int in,out;
     float y, phi, x[50],h[50],f1,Fs,c1;
     float hi,hl;
     float lp,hp,bp,sb;
     float flo,fhi,ftest,rad,ft;
     int l,k,m,n,i,N;
     int f,f2;
     char c;
     phi = 3.1416;
     Fs = 48.0;  /* Fs in KHz so enter all frequencies in KHz  */
     N = 21;
   for (k = 0; k<=50; k++)
h[k] = x[k] = 0.0; /* Initialize all values of h and x to zero.
*/
/* Set up AD16 chan0 to use timer 0 and get D/A data from data
register */
  *chan0_cnt = AD_CONTROL;
/* print menu */
   f=0;
   printf(" MENU FOR FIR FILTER USING A HAMMING WINDOW \n");
   printf("Enter A a for Low-pass Filter, B for a High-pass
Filter, n");
```

```
   printf("C for a Band-pass Filter and D for a Stop-band Filter
\n");
   printf("<Q>uit\n\n");
   printf("Enter selection >");
     while(f==0)
   {
   scanf("%c",&c);
       switch(c)
       {
      case 'A':case 'a':
      /* Case of a Low-pass Filter  */
   printf ("LP Case \n");
   printf("Input the value of the cutoff freq. in KHz. Must be
<Fs.\n");
   scanf ("%f", &f1);
   printf("Input a test frequency in Khz. \n");
   scanf ("%f", &ftest);
      for (i = -(N-1)/2; i <= (N-1)/2; i++)
      {
      if(i == 0)
     {
       h[i +(N-1)/2] = 2.0*f1/Fs;
      }
      else
h[i+(N-1)/2]=
        ((sin(2*phi*f1/Fs*i))/(phi*i))*(.54+.46*cos(2*phi*i/N));
    printf(" %d %f\n", i,h[i+(N-1)/2]);
       }
        f = 1;  /* To quit case A */
        break;
      case 'B':case 'b':
      /* Case of a High-pass Filter  */
    printf ("HP Case \n");
    printf("Input the value of the cutoff freq. in KHz. Must be
<Fs.\n");
    scanf ("%f", &f1);
    printf("Input a test frequency in KHz. \n");
    scanf ("%f", &ftest);
      for (i = -(N-1)/2; i <= (N-1)/2; i++)
      {
      if(i == 0)
     {
       h[i +(N-1)/2] = -2.0*f1/Fs;
      }
      else
h[i+(N-1)/2]=-
((sin(2*phi*f1/Fs*i))/(phi*i))*(.54+.46*cos(2*phi*i/N));
     printf(" %d %f\n", i,h[i+(N-1)/2]);
       }
        f = 1;  /* To quit case B */
        break;
      case 'C':case 'c':
      /* Case of a Band-pass Filter  */
      printf ("BP Case \n");
```

```
     printf("Input the value of the lower cutoff freq. in KHz. Must be
<Fs.\n");
     scanf ("%f", &flo);
     printf(" Input the value of the upper cutoff freq. in KHz.
Must be <Fs\n");
     printf(" and greater than the lower cutoff freq. \n");
     scanf ("%f", &fhi);
     printf("Input a value for a test frequency in Khz. \n");
     scanf ("%f", &ftest);
      for (i = -(N-1)/2; i <= (N-1)/2; i++)
      {
      if(i == 0)
     {
       h[i +(N-1)/2] = 2.0*(fhi - flo)/Fs;
      }
       else
       {
        hi = ((sin(2*phi*fhi/Fs*i))/(phi*i));
        hl = ((sin(2*phi*flo/Fs*i))/(phi*i));
        h[i+(N-1)/2]= (hi - hl)*(.54+.46*cos(2*phi*i/N));
        }
     printf(" %d %f\n", i,h[i+(N-1)/2]);
       }

        f = 1;  /* To quit case C */
        break;
      case 'D':case 'd':
      /* Case of a Stop-band Filter  */
     printf ("SBA Case \n");

     printf("Input the value of the lower cutoff freq. in KHz.
Must be    <Fs.\n");
     scanf ("%f", &flo);
     printf(" Input the value of the upper cutoff freq. in KHz.
Must be <Fs\n");
     printf(" and greater than the lower cutoff freq. \n");
     scanf ("%f", &fhi);
     printf("Input a value for a test frequency. \n");
     scanf ("%f", &ftest);

      for (i = -(N-1)/2; i <= (N-1)/2; i++)
      {
      if(i == 0)
     {
       h[i +(N-1)/2] = 2.0*(flo - fhi)/Fs;
      }
      else
       {
       hi = ((sin(2*phi*fhi/Fs*i))/(phi*i));
       hl = ((sin(2*phi*flo/Fs*i))/(phi*i));
       h[i+(N-1)/2] = (hl - hi)*(.54+.46*cos(2*phi*i/N));
       }
     printf(" %d %f\n", i,h[i+(N-1)/2]);
       }
```

```
        f = 1;  /* To quit case D */
        break;
      case 'q':case 'Q':
       f=1;
/* Changing f from 0 to 1 will terminate the While function. */
      printf ("Will quit. \n");
       break;
      default:
      printf("Selection invalid: Re-enter. \n>");
       break;
    }
   }

/*  Begin processing loop */

    while (1)
        {
        /* Wait for new data to arrive */
        while (!(*status & RINT1));
        /* Write out processed data */
        *chan0 = out;
        /* Read in new data */
        in = *chan0;
        /* Shift to lower 16 bits and convert to float */
        x[20] = (float) (in >> 16);
    /****************************************
    *
    *  Processing goes here.
    *  We have a value of x from the A/D converter.
    *  Use this value of x in an algorithm to produce y.
    *  Then output y.
    *
    ****************************************/
       m = N - 1;
       y = 0.0;
      for (n=0; n<=m; n++)
       y=y+x[n]*h[m-n];
/*  Now output y through the D/A converter   */
/* Convert to integer and shift to upper 16 bits */
       out = (int) y;
       out = out << 16;
/*    Now shift all the values of x   */
     for (k=m; k>=0; k--)
           x[k] = x[k+1];
    } /* end while(1) */
} /* end main */
```

Test Results

The program worked generally as designed except that the output from the filter was rather noisy. The noise could be as a result of some damage to the interface board, possibly by using too large

of an input signal. In all cases, the values of the h parameters were calculated and displayed on the screen. These values can be used independently of the program on other systems.

Concluding Remarks

The program is designed so that it can easily be expanded. It has been noted that ASPI uses the Parks-McClellan algorithm for designing FIR filters in its DFDP3 filter design package. This algorithm can easily be substituted for the windowing method. Also, it will be interesting to substitute the infinite impulse response (IIR) algorithm in the program. MATLAB has a C-compiler for the Real-Time Workshop. An evaluation needs to be made of this package for real-time DSP filtering.

References

[1] PECTRON Microsystems, Inc., SPOX The DSP Operating System Application Programming Manual, Santa Barbara: SPECTRON Microsystems, 1990.

[2] A. V. Oppenheim and R. W. Shafer, Digital Signal Processing,nglewood Cliffs: Prentice Hall, 1975.

[3] J. S. Lim and A. V. Oppenheim, Advanced Topics in Signal Processing, Englewood Cliffs: Prentice Hall, 1988.

[4] K. S. Lin (Editor), Digital Signal Processing Applications with the TMS320 Family, Dallas: Texas Instruments Incorporated, 1986.

[5] R. Chassaing and D. W. Horning, Digital Signal Processing Laboratory using the TMS320C25, Englewood Cliffs: Prentice Hall, 1990.

[6] P. A. Lynn and W. Fuerst, Digital Signal Processing with Computer Applications, New York: John Wiley & Sons, 1990.
B. W. Kernighan and Dennis M. Ritchie, The C Programming Language (second edition), Englewood Cliffs: Prentice Hall, 1988.

Biographical Information

James E. Cross is an associate professor of Electrical Engineering at Southern University in Baton Rouge, Louisiana. He has been a member of the faculty since 1962. Cross earned the BES degree in Electrical Engineering from Johns Hopkins University in 1960, served two years as an officer in the US Army and earned the MS degree in Electrical Engineering at Louisiana State University and the University of Florida. Cross has also earned the Bachelor's, Masters and Doctor of Theology degrees from Christian Bible College. He has worked on several Digital Signal Processing research projects for the Department of Defense.