User's Guide for the

PF1 Programmable Filter

Version 1.01

© 1997 TDT
Printed 10/29/97

4550 NW 6th St
Gainesville, FL 32609 USA
Phone: (352)375-1623
Fax: (352) 375-4523
E-mail: quikki@tdt-quikki.com

Table of Contents

Overview 1
PF1 Supplemental Software 2
Built-In Filter Functions 3
- Corner Frequency Placement 3
- Bypass and Nopass 3
- Gain Adjustment 4
- Saving and Loading Configurations 5

Frequency Scaling 5
Coefficient Formats and Downloading 7
- FIR Filters 8
- Direct II IIR Filters 9
- Cascaded Bi-quad IIR Filters 10

Using the PF1_LOAD Utility 12
MATLAB 'm-file' Examples 12
- FIR_COE1.M 13
- IIR_COE1.M 14
- IIR_COE2.M 15
- BIQ_COE1.M 17
- BIQ_COE2.M 20

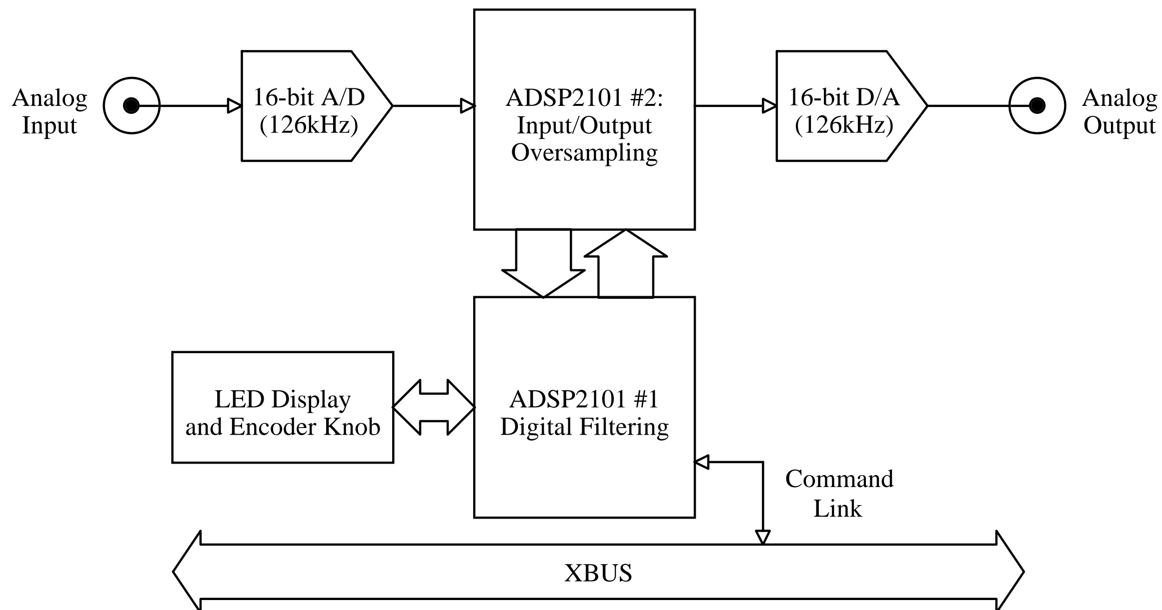
Filter Coefficient Libraries 22
- Using PF1BFILT 22
- Notes/Warnings on Filter Coefficient Libraries 24
- The PF1bfilt Function Call 24

FIR.EXE and XFUNC.EXE 25

Overview

The PF1 is a high-performance, audio-band digital filter module for both FIR and IIR filtering applications. The PF1 has built-in IIR lowpass and highpass filter functions that can be combined to yield bandpass and notch type frequency responses. The built-in filters can be placed either manually using the front-panel controls, or using software driver calls.

The PF1's analog performance is enhanced by oversampling the analog input and output. Basically the A/D and D/A converters *over*sample at 3 times the main digital filter's sampling rate of 42kHz. A second DSP chip implements ***digital*** pre- and post-filtering for anti-aliasing and anti-imaging requirements, and as a result, the PF1 does not need any expensive, high-quality analog filters on its input and output. A logical diagram of the PF1 is shown below:



In addition to basic PF1 operation, this supplemental document explains the numeric format and scaling requirements for custom FIR or IIR filter coefficients. To program the PF1 with custom filters, coefficients are downloaded using XBDRV software drivers incorporated into your application, or using one of the utility programs provided. The PF1_LOAD.EXE utility is used for downloading coefficients from standard ASCII text files. The PF1BFILT.EXE utility is used in conjunction with TDT filter coefficient library files and the FIR.EXE and XFUNC.EXE programs are used for generating FIR filter functions from Microsoft Windows.

PF1 Supplemental Software

Custom Filters and PF1_LOAD.EXE:

Custom filter functions and associated coefficients can be calculated using a variety of DSP type software packages such as MATLAB, DADiSP, and Hypersignal. The "TDT\PF1UTILS\LOADER" directory contains MATLAB m-files that demonstrate the process of digital filter design, construction, and coefficient downloading. The m-files run under MATLAB and save coefficients to disk in standard ASCII floating-point format. The supplied PF1_LOAD.EXE utility can then be used to download the coefficient files to the PF1. See **Using the PF1_LOAD Utility** for more information.

Filter Libraries and PF1BFILT.EXE*:

For customers who do not have the software to generate filters, the "FILT_LIB" library includes filter coefficient library files with standard response types and a large number of corner/center frequencies. The filters in these files can be accessed and loaded to the PF1 directly from DOS with PF1BFILT.EXE or from a user program using the PF1bfilt function call. The source code for both options is provided allowing for modification and porting by the end user. See the **Filter Coefficient Libraries** section for more information.

FIR.EXE and XFUNC.EXE:

The "TDT\PF1UTILS\WINUTILS" directory contains two programs for use in Microsoft Windows. The FIR.EXE program allows the user to specify arbitrary FIR filter functions using a graphic design method. The program then generates the appropriate FIR filter coefficients using the frequency-sampling/window method. The filter coefficients can be downloaded to a PF1 and verified using the XFUNC.EXE program. See the **FIR.EXE and XFUNC.EXE** section for more information.

Installation

Be sure your TDT hardware is installed and functioning properly. Refer to the System II Installation and Reference Guide for details. If necessary, update your version of AP.OBJ (AP2 onboard code).

- Run the INSTALL program on Disk #1 of your PF1 Supplemental Drivers. This will install the files to TDT standard directories and will create icons in the TDT program group for Windows applications it installs.

---

* Available from TDT on request

Built-In Filter Functions

The PF1 is preprogrammed with coefficients for Butterworth IIR type lowpass and highpass filtering. The PF1 can run a lowpass and a highpass filter simultaneously to yield bandpass or band-reject (notch) filtering. The lowpass and highpass corner frequencies can be placed over a 50Hz to 15kHz range with placement resolution, order (number of poles) and numeric precision as follows:

Frequency Range	Placement Resolution	Order/Prec.
50 Hz - 1000 Hz	5 Hz	6/32-bit
1000 Hz - 2990 Hz	10 Hz	10/16-bit
3 kHz - 9.95 kHz	50 Hz	10/16-bit
10 kHz - 15 kHz	100 Hz	10/16-bit

Corner Frequency Placement

Corner Frequency placement can be done with a software driver call or using the front panel controls. If the highpass corner frequency is set greater than the lowpass, a notch filter is implemented. Conversely, if the lowpass corner frequency is set greater than the highpass, a bandpass filter is implemented.
To place filters manually with the front panel controls, do the following:

1. From the main menu, select "LP_f" or "HP_f" with the Encoder, then press ENTER
2. Dial in the desired corner frequency.
   (to disable the filter, dial counter-clockwise until "OFF" appears)
3. Press ENTER to select, ESC to abort.

To place filters from software, use the XBDRV **PF1freq** driver call. For example, the call:

```
PF1freq(1, 600, 1400);
```

will set a lowpass corner frequency of 600Hz and highpass corner frequency of 1.4kHz, which is equivalent to a notch-filter with 800Hz bandwidth, centered at about 1kHz. To disable a filter, specify a corner frequency of zero, for example:

```
PF1freq(1, 600, 0);
```

which implements lowpass only filtering with 600Hz corner frequency.

Bypass and Nopass

On power up and reset, and after setting filter frequencies (manually or from software), the main digital filter is active. For convenience, the main filter can be

temporarily bypassed or “nopassed” (zero output). In bypass mode, the oversampling A/D - D/A circuit is still active, but no other filtering is active. From software, the bypass and nopass features are controlled by the **PF1bypass** and **PF1nopass** drivers. After calling either of these, the main filter is cleared. To manually bypass the main filter,

1. Press ESC, and then confirm by pressing ENTER
2. “ByPs” should appear in the display
3. Press ESC or ENTER to reactivate the main filter

NOTE: If a custom filter has been downloaded from a host computer it will NOT reactivate after manual ByPass is selected. You must re-download the filter.

Gain Adjustment

In some cases, the filtered output signal level may be significantly reduced from the input level. Dynamic range is lost because fewer bits of the 16-bit D/A converter are effectively utilized. However, internally the PF1 computes filter output values to a full 32-bits, then uses the upper 16-bit word for D/A conversion. The lower 16-bits are still valid, but the D/A converter can use only 16-bits. Therefore, the PF1 allows internal digital gain adjustment by first shifting the 32-bit result to the left so that more precision is obtained from the upper 16-bits. Each bit shift is a doubling of gain or 6dB. Lowpass and highpass gains are set independently.

To manually adjust PF1 gain,

1. From the main menu, select “LP_g” or “HP_g” with the Encoder, then press ENTER.
2. Dial in the desired gain.
3. Press ENTER to select, or ESC to abort.

From software, the **PF1gain** driver call is provided. For example:

```
PF1gain(1, _0DB, _6DB);
```

sets the low and highpass filter gains to 0dB (unity gain) and 6dB (2x gain) respectively.

Saving and Loading Configurations

Up to eight filter configurations can be saved in the PF1's non-volatile memory for instant recall after power up or reset. To save the current PF1 configuration,

1. Select “Save” with the Encoder, then press ENTER
2. Select a preset number: “PS-1”' to “PS-8” .
3. Press ENTER to select, and again to confirm (ESC to abort).

To load a preset and make it the current configuration,

1. Select “Load” with the Encoder, then press ENTER
2. Select a preset number: “PS-1” to “PS-8” .
3. Press ENTER to select, and again to confirm (ESC to abort).

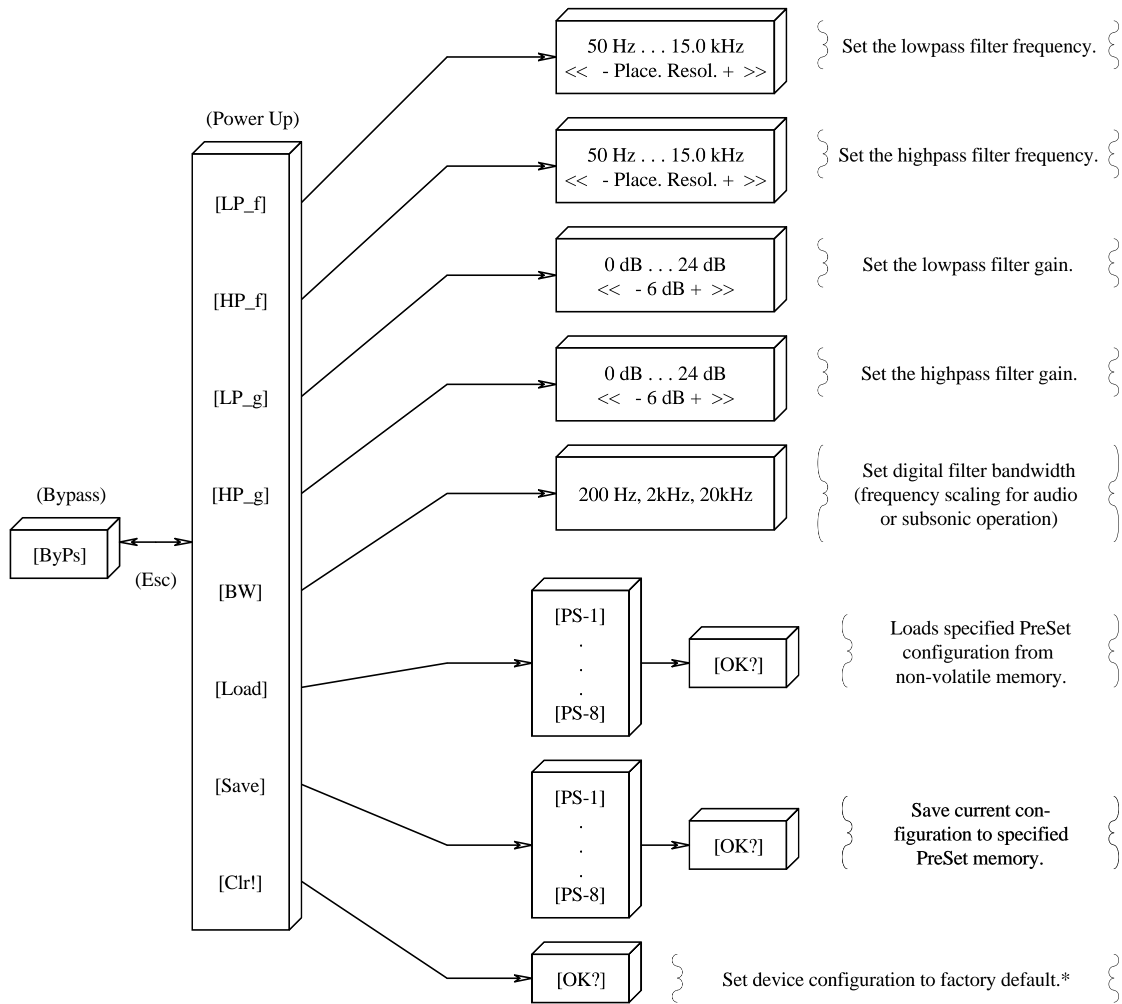
NOTE: Downloaded filters can not be saved in the non-volatile memory.

Frequency Scaling

The bandwidth of the PF1 can be lowered by reducing the sampling rate of the main digital filter, a technique known as Frequency Scaling. This allows more precise placement of filters in the low-frequency range and below 50Hz down to 0.5Hz for infrasonic applications. The PF1 can be set for bandwidths of 20kHz (normal audio operation), 2kHz or 200Hz. **The bandwidth setting applies to both built-in and custom downloaded filters.** For example, a digital filter designed with a corner frequency of 1500Hz at the 20kHz bandwidth setting will have 150Hz and 15Hz corner frequencies at the 2kHz and 200Hz settings.
It is important to note that the PF1's anti-aliasing circuitry is set permanently at 20kHz. So operation at lower bandwidths, and in turn slower sampling rates, may require an additional external anti-aliasing filter.

To select bandwidth, simply select “BW” with the encoder, press ENTER, select the bandwidth, and press ENTER again.

A menu diagram for the PF1 is shown below:



For information on programming the PF1 from a host computer, refer to the XBDRV software reference and the following sections of this supplement.

Coefficient Formats and Downloading

When programming the PF1 with custom FIR or IIR filters, certain numeric constraints must be observed. Namely, when designing the filter, the coefficient values must not exceed a certain magnitude, which depends on the filter type and computational precision. This is because the PF1 uses a fixed-point processor (ADSP2101). The PF1's internal filtering algorithms are designed with the radix point position chosen for best dynamic performance with a given filter type and precision. For fixed point numbers, the bits to the left of the radix point represent the *signed* integer part of the number and the bits to the right of the radix point represent the fractional part of the number. For example, the 1.15 format indicates a 16-bit number with 1 bit for the sign and 15 bits for the fractional part. This format is used to represent decimal values from -1.0 to $+\frac{2^{15}-1}{2^{15}} = +0.999969$. The table below of corresponding decimal and binary values illustrates the 1.15 fixed-point format:

Dec. Value	Binary Value	
- 1.0	1.000 | 0000 | 0000 | 0000	
- 0.999969	1.000 | 0000 | 0000 | 0001	(negative values are
:	:	2's complement)
- 0.0000305	1.111 | 1111 | 1111 | 1111	
0.0	0.000 | 0000 | 0000 | 0000	
+0.0000305	0.000 | 0000 | 0000 | 0001	
:	:	
+0.999969	0.111 | 1111 | 1111 | 1111	

The table below summarizes the fixed-point coefficient formats and decimal value ranges for the PF1's various filter types:

Filter Type	Prec.	Code	Max.Taps /Order	Coeff. Format	Min. Value	Max. Value	Scale By
FIR	16-bit	FIR16	160/159	1.15	-1.0	+0.999969	2^{15}
FIR	32-bit	FIR32	64/63	2.30	-2.0	+1.999999999	2^{30}
Direct II IIR	32-bit	IIR32	8/7	8.24	-128.0	+127.99999994	2^{24}
Bi-quad IIR	16-bit	BIQ16	39/26	2.14	-2.0	+1.9999	2^{14}
Bi-quad IIR	32-bit	BIQ32	24/16	2.30	-2.0	+1.999999999	2^{30}

In general for format I.Q, decimal coefficient values can range from

$$-2^{I-1} \text{ to } +\frac{2^{I+Q-1}-1}{2^{Q}}.$$

FIR Filters

The PF1's 16-bit precision FIR filter algorithm uses the 1.15 format. When designing a filter for use with this algorithm, one must ensure that the coefficient values resulting from the design algorithm are in the range -1.0 to +0.999969. These values are then scaled by 2^{15} = 32768 = 0x8000 (hex) to 16-bit whole integers to facilitate downloading to the PF1 through the XBUS serial communication link. The XBDRV procedure **PF1fir16** illustrates this process. It scales an array of FIR coefficients in PC memory and downloads them to the PF1:

```
void PF1fir16( int din, float bcoes[], int ntaps)
{
  int i;
  PF1begin(din);
  for(i=0; i<ntaps; i++)
  /* values in bcoes[] must be in 1.15 format */
    PF1b16(din, (int)(bcoes[i]*0x8000+0.5));
  PF1type(din, FIR16, ntaps);
}
```

The FIR's feedforward tap coefficients are sent to the PF1, one at a time, with **PF1b16** in the order b_0, b_1,… to b_N where N = ntaps - 1 is the filter order. This procedure also sets the filter algorithm to FIR, 16-bit precision (FIR16) and starts the filtering process. Note that this procedure combines the basic PF1 driver calls **PF1begin**, **PF1b16** and **PF1type**, which would be employed similarly in a user-customized downloading routine.

The **PF1fir32** procedure performs a similar process for the PF1's 32-bit precision FIR algorithm (FIR32):

```
void PF1fir32( int din, float bcoes[], int ntaps)
{
  int i;
  PF1begin(din);
  for(i=0; i<ntaps; i++)
  /* values in bcoes[] must be in 2.30 format */
    PF1b32(din, (long)(bcoes[i]*0x40000000+0.5));
  PF1type(din, FIR32, ntaps);
}
```

Note that **PF1b32** is used to send 32-bit coefficients to the PF1. The FIR16 algorithm can run more taps than FIR32, and 16-bit precision is generally

adequate for FIR filtering. The FIR32 algorithm is mainly used when coefficients larger than ±1 are required, which allows for signal boosts of about 6dB. The FIR16 algorithm can only attenuate the input signal.

Direct II IIR Filters

Downloading of Direct II IIRs is basically an extension of the procedure for downloading FIRs: IIRs also require downloading of feed*back* tap coefficients a_0, a_1,… to a_N in addition to feedforward tap coefficients b_0, b_1,… to b_N. Note, a_0 **must be sent** to preserve alignment but the value is not actually used (Direct II designs always return 1 for a_0). The frequency domain transfer function of this filter is

$$F(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}}.$$

The XBDRV procedure **PF1iir32** illustrates the scaling and downloading process for Direct II IIRs: The elements of two arrays containing the "b" and "a" coefficients are scaled and sent one at a time to the PF1:

```
void PF1iir32( int din, float bcoes[], float acoes[], int ntaps)
{
  int i;
  PF1begin(din);
  for(i=0; i<ntaps; i++)
  {
  /* values in bcoes[] & acoes[] must be in 8.24 format: */
    PF1b32(din, (long)(bcoes[i]*0x800000+0.5));
    PF1a32(din, (long)(-acoes[i]*0x800000+0.5));
  }
  PF1type(din, IIR32, ntaps);
}
```

Note that there are separate PF1 drivers **PF1a32** and **PF1b32** for sending "a" and "b" coefficients respectively (**PF1a16** and **PF1b16** are used for 16-bit precision filter algorithms). This procedure also sets the filtering algorithm to Direct II IIR, 32-bit precision (IIR32) and starts the filtering process.

The IIR32 algorithm is limited to implementation of low-order filters. However, generating coefficients for IIR32 is more straightforward than for BIQ16 or BIQ32 (see below) because no additional coefficient manipulation is required.

Cascaded Bi-quad IIR Filters

Cascaded Bi-quad IIRs consist of cascaded second-order Direct II IIR sections. This configuration results in very stable higher-order filters, whereas fixed-point Direct II IIR implementations greater than order 5 tend to be numerically unstable. However if your filter design software does not automatically realize this type of cascaded architecture, some coefficient manipulation is required in the design process. This involves factoring the "a" and "b" coefficients (from a Direct II design) into groups of second-order "a" and "b" coefficients, a technique that is explained in the **MATLAB m-file Examples** section.

The downloading process for Bi-quad IIR coefficients is similar to Direct II IIRs. The coefficients must be sent as "m" groups of second order sections:

$$a_{10},\ a_{11},\ a_{12},\ a_{20},\ a_{21},\ a_{22},\ldots\ a_{m0},\ a_{m1},\ a_{m2}\ (a_{i0}=1,\ i=1\text{ to }m)$$

and

$$b_{10},\ b_{11},\ b_{12},\ b_{20},\ b_{21},\ b_{22},\ldots\ b_{m0},\ b_{m1},\ b_{m2}$$

The filter order is N = 2m, but a total of 3m feedforward and 3m feedback tap coefficients are sent. Note, a_{i0} of each group **must be sent** to preserve alignment, but the values are not actually used (as indicated above, the value is always 1 for this design). The frequency domain transfer function of this filter architecture is

$$F(z)=\frac{b_{10}+b_{11}z^{-1}+b_{12}z^{-2}}{1+a_{11}z^{-1}+a_{12}z^{-2}}\cdot\frac{b_{20}+b_{21}z^{-1}+b_{22}z^{-2}}{1+a_{21}z^{-1}+a_{22}z^{-2}}\cdot\ \ldots\cdot\frac{b_{m0}+b_{m1}z^{-1}+b_{m2}z^{-2}}{1+a_{m1}z^{-1}+a_{m2}z^{-2}}.$$

The XBDRV procedure **PF1biq16** illustrates the scaling and downloading process for Bi-quad IIRs: The elements of two arrays containing the "b" and "a" coefficients groups ordered as above are scaled and sent one at a time to the PF1. This procedure is given the number of bi-quad sections, nbiqs (= m), instead of the number of taps:

```
void PF1biq16( int din, float bcoes[], float acoes[], int nbiqs)
{
  int i,n;
  n = nbiqs*3;
  PF1begin(din);
  for(i=0; i<n; i++)
  {
  /* values in bcoes[] & acoes[] must be in 2.14 format:  */
    PF1b16(din, (int)(bcoes[i]*0x4000+0.5));
    PF1a16(din, (int)(-acoes[i]*0x4000+0.5));
  }
  PF1type(din, BIQ16, nbiqs);
}
```

This procedure also sets the filter algorithm to Bi-quad IIR, 16-bit precision (BIQ16) and starts the filtering process.

The **PF1biq32** procedure performs a similar process for the PF1's 32-bit precision Bi-quad IIR algorithm (BIQ32):

```
void PF1biq32( int din, float bcoes[], float acoes[], int nbiqs)
{
  int i,n;
  n = nbiqs*3;
  PF1begin(din);
  for(i=0; i<n; i++)
  {
    PF1b32(din, (long)(bcoes[i]*0x40000000+0.5));
    PF1a32(din, (long)(-acoes[i]*0x40000000+0.5));
  }
  PF1type(din, BIQ32, nbiqs);
}
```

The BIQ32 algorithm is used for medium-order (<=12) filter implementation and is in general better for use at corner frequencies below about 1000Hz. The BIQ16 algorithm runs faster and is used for high-order (>12) filter implementations. The lower precision may result in increased noise and distortion.

Using the PF1_LOAD Utility

The program PF1_LOAD.EXE can be found in the "TDT\PF1UTILS\LOADER" directory. It can be run from the DOS prompt to implement custom filters on the PF1. The filter coefficients are read from standard ASCII files, scaled and downloaded to the PF1. Note, the same ASCII files can be loaded from a user application program using the appropriate XBDRV library call.

The usage of PF1_LOAD is as follows:

Syntax:

PF1_LOAD *din* *fileprefix* *filttype*

Parameters:

din	PF1 device index number
fileprefix	prefix of coefficient data file(s)
filttype	filter type code: FIR16, FIR32, IIR32, BIQ16, BIQ32, BYPASS, NOPASS

Coefficient Files:

Standard floating-point ASCII format (values separated by CR-LF).
File extensions ".b" and ".a" are assumed for feedforward (numerator) and feedback (denominator) coefficients respectively.

Example:

A sample coefficient file is provided in the "LOADER" directory. Load the file using the following PF1_LOAD command to implement a 20 pole Butterworth highpass at 5000Hz. With the filter coefficient files "sampfilt.b" and "sampfilt.a" in the current directory, type:

PF1_LOAD 1 sampfilt BIQ16 <enter>

The program automatically determines the number of filter taps from the length of the coefficient files. PF1_LOAD is used in the next section to download filters created in MATLAB.

MATLAB "m-file" Examples

The "TDT\PF1UTILS\LOADER" directory also contains five "m-files" (.m extension) which can be run under the MATLAB environment to demonstrate the filter design and coefficient computation process. MATLAB is a widely used, multi-purpose mathematical software package with optional "toolboxes" for specific applications. The demos in this section require MATLAB's Signal

Processing Toolbox. Design examples of FIR, Direct II IIR, and Cascaded Bi-quad IIR filters are covered.

NOTE: MATLAB's filter design functions require corner frequency specifications as a fraction of the Nyquist frequency (1/2 the sampling frequency), referred to as normalized frequency in the following examples.

FIR_COE1.M

This file computes a 128th-order FIR filter from an arbitrary magnitude frequency response specification using the Frequency Sampling technique.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This short routine will generate FIR coes that can    %
%  be loaded into the PF1 using the provided PF1_load    %
%  program.                                              %
%     - Run this routine under MATLAB                    %
%       ( you must have the optional SIGNAL package)     %
%     - In MATLAB type: >>!PF1_load 1 tstcoes FIR16      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     nyqfreq = 21005.128;  % 1/2 PF1's sampling frequency
     ntaps_1 = 128;      % filter order = # of taps -1

     fp = [ 0.0  1000.0  2000.0  3000.0  4000.0  6000.0  8000.0 nyqfreq ];
     mp = [ 1.0     1.0     0.1     0.1    0.01    0.01     1.0     1.0 ];

     % Generate filter coes using FIR2
     b = fir2(ntaps_1,fp/nyqfreq,mp);

     % Save them to disk
     fprintf('Saving to Disk...\n\n');
     !del tstcoes.b
     for i=1:ntaps_1+1,
       fprintf('tstcoes.b','%g\n',b(i));
     end
     fprintf('  Done.\n\n');

     % Gernerate a screen plot
     d2 = log10(nyqfreq*.99);   % don't exceed nyqfreq for freq. response plot
     f = logspace(1,d2,300);    % frequency axis--10Hz to nyqfreq
     th = f*pi/nyqfreq;         % unit circle frequency angle

     fz=abs(freqz(b,1,th));     % calculate frequency response at
                                % points on the unit circle btwn 0 and pi.

     % compare the actual and desired frequency response plots:
     axis([min(f) max(f) -90 10]);
     plot(f,20.0*log10(fz),fp,20.0*log10(mp));
     axis;
```

Two arrays **fp** and **mp** specify the frequency (in Hz) and magnitude (gain) breakpoints desired of the filter response. Then **fp** is normalized by one-half the PF1's sampling frequency and passed to the Signal Toolbox procedure **fir2** which returns the array of filter coefficients, **b**.

The coefficients are then saved to disk in a file called “tstcoes.b”. To implement this filter on the PF1, PF1_LOAD can be run at the MATLAB prompt by typing

>>! PF1_LOAD 1 tstcoes FIR16

The rest of the m-file verifies the design result by generating the frequency response with **freqz**, which evaluates the filter transfer function at points on the unit circle $z = e^{j\theta}, 0 < \theta < \pi$. The frequency response magnitude is then plotted (in dB) along with the original magnitude specification to show how well the filter fits the desired response.

IIR_COE1.M

This file will generate a Direct II IIR digital filter from a Chebyshev lowpass analog prototype. The Bi-linear Z-transform method is used to find the discrete equivalent. The Signal Toolbox function **cheby1** performs all the necessary filter design computations.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This MATLAB program will generate Direct II IIR coes     %
%  that can be loaded into the PF1 using the provided       %
%  PF1_load program.                                        %
%     - Run this routine under MATLAB                       %
%       ( you must have the optional SIGNAL package)        %
%     - In MATLAB type: >>!PF1_load 1 tstcoes IIR32         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     nyqfreq = 21005.128;   % 1/2 PF1's sampling frequency
     N = 5;                 % filter order
     freq = 1500;           % -3dB corner frequencies of stop band
     fn = freq/nyqfreq;

     %Generate filter coes using cheby1
     [b, a] = cheby1(N, 1.0, fn);       % Chebyshev lowpass filter

     %Save coes to disk
     fprintf('Saving to Disk...\n\n');
     !del tstcoes.b
     !del tstcoes.a
     % write coeffs of 2nd order section to file:
     for j=1:N+1,
       fprintf('tstcoes.b','%g\n',b(j));
       fprintf('tstcoes.a','%g\n',a(j));
     end;
     fprintf('   Done.\n\n');


     %Graph results to screen
     d = log10(freq);                         % center frequency decade
     d2 = min([log10(nyqfreq*.99) d+1]);      % don't exceed nyqfreq for
                                              % freq. response plot
     f = logspace(d-1,d2,300);                % frequency axis; decade below and
                                              % above center frequency
     th = f*pi/nyqfreq;                       % unit circle frequency angle
```

```
    fz=abs(freqz(b,a,th));      % calculate frequency response at
                                % points on the unit circle btwn 0 and pi.

    axis([min(f) max(f) -90 10]);
    plot(f,20*log10(fz));
    axis;
```

The lowpass corner frequency (in Hz) is specified in **freq**, then normalized and passed to **cheby1** along with the filter order **N** and passband ripple (0.1 dB). The Direct II form “b” and “a” coefficients are returned in arrays **b** and **a** respectively.

The coefficients are then saved to disk files “tstcoes.a” and “tstcoes.b”. To implement the filter on the PF1, PF1_LOAD can be run at the MATLAB prompt by typing

>>! PF1_LOAD 1 tstcoes IIR32

The rest of the m-file verifies the design result by generating the frequency response with **freqz**, which evaluates the filter transfer function at points on the unit circle $z = e^{j\theta}, 0 < \theta < \pi$. The frequency response magnitude is then plotted (in dB)

IIR_COE2.M

This file will generate a Direct II IIR digital filter from a Chebyshev notch, or bandstop, analog prototype. The Bi-linear Z-transform method is used to find the discrete equivalent. The Signal Toolbox function **cheby1** performs all the necessary filter design computations.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This MATLAB program will generate Direct II IIR coes      %
%  that can be loaded into the PF1 using the provided        %
%  PF1_load program.                                         %
%     - Run this routine under MATLAB                        %
%       ( you must have the optional SIGNAL package)         %
%     - In MATLAB type: >>!PF1_load 1 tstcoes IIR32          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    nyqfreq = 21005.128; % 1/2 PF1's sampling frequency
    N = 6;               % filter order (must be even for bandpass or notch)
    freq = [800 8000];   % 0dB corner frequencies of pass band
    fn = freq/nyqfreq;
```

```
%Generate filter coes using cheby1
[b, a] = cheby1(N/2, 0.5, fn);   % Bandpass filter
                                  % (specify order of N/2)

%Save coes to disk
fprintf('Saving to Disk...\n\n');
!del tstcoes.b
!del tstcoes.a
% write coeffs of 2nd order section to file:
for j=1:N+1,
  fprintf('tstcoes.b','%g\n',b(j));
  fprintf('tstcoes.a','%g\n',a(j));
end;
fprintf('   Done.\n\n');


%Graph results to screen
d = log10(sqrt(freq(1)*freq(2)));      % center frequency decade
d2 = min([log10(nyqfreq*.99) d+1]);    % don't exceed nyqfreq for
                                       % freq. response plot
f = logspace(d-1,d2,300);              % frequency axis; decade below and
                                       % above center frequency
th = f*pi/nyqfreq;                     % unit circle frequency angle

fz=abs(freqz(b,a,th));                 % calculate frequency response at
                                % points on the unit circle btwn 0 and pi.

axis([min(f) max(f) -90 10]);
plot(f,20*log10(fz));
axis;
```

The notch corner frequencies (in Hz) are specified in **freq**, then normalized and passed to **cheby1** along with 1/2 the filter order (N/2, as required by MATLAB) and passband ripple (0.1 dB). The Direct II form “b” and “a” coefficients are returned in arrays **b** and **a** respectively.

The coefficients are then saved to disk files “tstcoes.a” and “tstcoes.b”. To implement the filter on the PF1, PF1_LOAD can be run at the MATLAB prompt by typing

>>! PF1_LOAD 1 tstcoes IIR32

The rest of the m-file verifies the design result by generating the frequency response with **freqz**, which evaluates the filter transfer function at points on the unit circle $z = e^{j\theta}$, $0 < \theta < \pi$. The frequency response magnitude is then plotted (in dB).

BIQ_COE1.M

This file will generate a Cascaded Bi-quad IIR digital filter from a Butterworth highpass analog prototype. The Bi-linear Z-transform method is used to find the discrete equivalent. The Signal Toolbox function **butter** performs all the necessary filter design computations.

As mentioned previously, this example section explains the additional manipulation required to obtain second-order bi-quad sections from the poles and zeros of the filter's transfer function (which are directly related to the Direct II filter coefficients). A general discussion of the method follows the explanation of the m-file example below.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This MATLAB program will generate Biquad IIR coes        %
%  that can be loaded into the PF1 using the provided       %
%  PF1_load program.                                        %
%     - Run this routine under MATLAB                       %
%       ( you must have the optional SIGNAL package)        %
%     - In MATLAB type: >>!PF1_load 1 tstcoes BIQ16         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     nyqfreq = 21005.128;  % 1/2 PF1's sampling frequency
     N = 20;               % filter order (must be even)
     freq = 5000;          % -3dB corner frequency

     [z,p,k] = butter(N,freq/nyqfreq,'high');    % Highpass filter

     % -- z is not used --
     k  = k^(2/N);                % divide gain equally btwn N/2 stages
     p  = cplxpair(p);            % pair complex conjugate poles
     zz = [1 1];                  % zeros all at +1 for Butterworth highpass
     b  = poly(zz) * k;           % all 2nd order numerator sections are same

     fprintf('Saving to Disk...\n\n');
     !del tstcoes.b
     !del tstcoes.a

     i = 1;
     fz = 1;
     d = log10(freq);                     % Corner frequency decade
     d2 = min([log10(nyqfreq*.99) d+1]);% Don't exceed nyqfreq for freq.
                                          %   response plot.
     f = logspace(d-1,d2,300);            % Frequency axis; decade below and
                                          %   above center freq.
     th = f*pi/nyqfreq;                   % Unit circle frequency angle

     % pair poles for 2nd order denominator sections:
     while i<N,
       pp = [p(i) p(i+1)];
       a = poly(pp);
       i = i + 2;
       fz=fz.*abs(freqz(b,a,th));        % Calculate frequency response at
                                         % points on unit circle btwn 0 and pi.
                                         % Overall response is product of stages
       % write coeffs of 2nd order section to file:
       for j=1:3,
         fprintf('tstcoes.b','%g\n',b(j));
         fprintf('tstcoes.a','%g\n',a(j));
       end;
     end;
```

```
    fprintf('   Done.\n\n');

    axis([min(f) max(f) -90 10]);
    plot(f,20*log10(fz));
    axis;
```

The highpass corner frequency **freq** is normalized and passed to the **butter** function along with the filter order **N** and switch for highpass computation. The zeros, poles, and gain of the digital filter's transfer function are returned in arrays **z**, **p** and **k** respectively. The task then becomes how to pair zeros and poles and create a 2nd-order numerator section from each pair of zeros, and a 2nd-order denominator section from each pair of poles. In the case of Butterworth highpass (or lowpass) filters, pairing of zeros is simple because they are all at +1 (or -1) so all 2nd-order *numerator* sections are the same. The gain k is divided equally between the N/2 sections. The MATLAB function **cplxpair** is used to pair poles, which are complex-conjugates of each other. In the "while" loop, the pole pairs are made into 2nd-order coefficient sections and saved to disk resulting in files "tstcoes.b" and "tstcoes.a" ordered as required by the PF1's BIQ16 and BIQ32 algorithms. The overall frequency response of the filter is computed recursively inside the loop, by accumulating the product (point-wise) of each second-order section's frequency response.

To implement the filter on the PF1, PF1_LOAD can be run at the MATLAB prompt by typing

```
>>! PF1_LOAD  1  tstcoes  BIQ16
```

Note that BIQ32 cannot be used unless the order N is lowered.

In the above example, the design algorithm returned the zeros and poles of the filter transfer function F(z), which are just the roots of the polynomials made from Direct II coefficients "b" and "a" respectively:

$$F(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_N z^{-N}} = k\,\frac{(z - z_1)(z - z_2)\cdots(z - z_N)}{(z - p_1)(z - p_2)\cdots(z - p_N)}$$

(zeros: $z_1, z_2, \cdots, z_N$ poles: $p_1, p_2, \cdots, p_N$ $k = b_0/a_0$)

where N is also the filter order. Recall that for Direct II,

$a_0 = 1,$ so $k = b_0.$

In general, to convert a set of Direct II filter coefficients into bi-quad sections, the zeros and poles must be first be obtained (by using the MATLAB **roots** function on the numerator and denominator coefficients, for example) then paired back together into second-order numerator/denominator sections, as was done in the m-file example above, to yield an equivalent transfer function

$$F(z) = \frac{b_{10} + b_{11}z^{-1} + b_{12}z^{-2}}{1 + a_{11}z^{-1} + a_{12}z^{-2}} \cdot \frac{b_{20} + b_{21}z^{-1} + b_{22}z^{-2}}{1 + a_{21}z^{-1} + a_{22}z^{-2}} \cdot \ldots \cdot \frac{b_{m0} + b_{m1}z^{-1} + b_{m2}z^{-2}}{1 + a_{m1}z^{-1} + a_{m2}z^{-2}}$$

where m is the number of sections and N = 2m. The pairing of zeros and poles requires some considerations: First, coefficients must all be real, so complex-conjugate zeros must be paired, and complex-conjugate poles must be paired. Second, as a rule, zero pairs should be grouped with pole pairs that are closest to each other in the complex plane. The overall gain is divided between the bi-quad sections according to the relation

$$k = b_0 = b_{10} b_{20} \ldots b_{m0}$$

Each zero pair is converted to a polynomial (e.g. using MATLAB's **poly** function), which is then scaled by the corresponding b_{i0} to yield numerator coefficient forms as in the F(z) expression above. The pole pairs are converted to polynomials likewise but are not scaled. For equal gain on all stages, set

$$b_{i0} = \sqrt[m]{k} = k^{2/N}$$

as was done in the m-file example. The cascade arrangement of the 2nd-order sections should not be critical in most cases, but overall fixed-point roundoff error may be reduced with certain configurations. Consult a text on the practical implementation of digital filters with microprocessors.

Be aware that root-finding algorithms such as **root** can return erroneous results for high-order, numerically sensitive polynomials especially those with repeated roots. Therefore, an algorithm that can return zeros and poles *directly* should be used if possible for the filter design. In the preceding example, the zeros were known to all be at +1, so the zeros returned by the **butter** algorithm were not used at all. In fact, the zeros returned are in error, skewed about +1.

BIQ_COE2.M

This file is a variation of the BIQ_COE1.M example above. It generates a Cascaded Bi-quad IIR digital filter from a Butterworth bandpass analog prototype. The Bi-linear Z-transform method is used to find the discrete equivalent. The Signal Toolbox function **butter** performs all the necessary filter design computations.

As in the previous example, some manipulation is required to obtain second-order bi-quad sections from the poles and zeros of the filter's transfer function.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  This MATLAB program will generate Biquad IIR coes           %
%  that can be loaded into the PF1 using the provided          %
%  PF1_load program.                                           %
%     - Run this routine under MATLAB                          %
%       ( you must have the optional SIGNAL package)           %
%     - In MATLAB type: >>!PF1_load 1 tstcoes BIQ16 (or BIQ32) %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     nyqfreq = 21005.128; % 1/2 PF1's sampling frequency
     N = 12;            % filter order (must be even)
     freq = [800 3000]; % -3dB corner frequencies of pass band

     [z,p,k] = butter(N/2,freq/nyqfreq);  % Bandpass
                                          % (specify order of N/2)
     % -- z is not used --
     k  = k^(2/N);              % divide gain equally btwn N/2 stages
     p  = cplxpair(p);          % pair complex conjugate poles
     zz = [1 -1];               % pair zero at +1 with zero at -1
     b  = poly(zz) * k;         % all 2nd order numerator sections are the same

     fprintf('Saving to Disk...\n\n');
     !del tstcoes.b
     !del tstcoes.a

     i = 1;
     fz = 1;
     d = log10(sqrt(freq(1)*freq(2)));  % Center frequency decade
     d2 = min([log10(nyqfreq*.99) d+1]);% Don't exceed nyqfreq for freq.
                                        %   response plot.
     f = logspace(d-1,d2,300);          % Frequency axis; decade below and
                                        %   above center freq.
     th = f*pi/nyqfreq;                 % Unit circle frequency angle

     % pair poles for 2nd order denominator sections:
     while i<N,
       pp = [p(i) p(i+1)];
       a = poly(pp);
       i = i + 2;
       fz=fz.*abs(freqz(b,a,th));  % Calculate frequency response at
                                   %   points on the unit circle btwn 0 and pi.
                                   % Overall response is product of stages.

       % write coeffs of 2nd order section to file:
       for j=1:3,
         fprintf('tstcoes.b','%g\n',b(j));
         fprintf('tstcoes.a','%g\n',a(j));
       end;
     end;
```

```
    fprintf('   Done.\n\n');

    axis([min(f) max(f) -90 10]);
    plot(f,20*log10(fz));
    axis;
```

The bandpass corner frequencies (in Hz) are specified in **freq**, then normalized and passed to **butter** along with 1/2 the order (N/2). The zeros, poles, and gain of the digital filter's transfer function are returned in arrays **z**, **p** and **k** respectively. As in the previous example, the zeros and poles must be paired to create 2nd-order numerator and denominator sections respectively. In the case of an Nth order Butterworth bandpass filter, N/2 zeros are at +1 and the other N/2 are at -1, so N/2 identical 2nd-order *numerator* sections can be formed from +1, -1 pairs. The gain k is divided equally between the N/2 sections. The MATLAB function **cplxpair** is used to pair poles that are complex-conjugates of each other. In the "while" loop, the pole pairs are made into 2nd-order coefficient sections and saved to disk resulting in files "tstcoes.b" and "tstcoes.a" ordered as required by the PF1's BIQ16 and BIQ32 algorithms. The overall frequency response of the filter is computed recursively inside the loop, by accumulating the product (point-wise) of each second-order section's frequency response.

Filter Coefficient Libraries

The programs and files described in this section are not included with PF1 drivers as of August 1995. Contact TDT if you feel you could use these files.

The "FILT_LIB" library provides PF1 users with pre-computed filter coefficient libraries in the form of binary data files which can be read and loaded to the PF1 with the PF1BFILT.EXE program or the PF1bfilt function call. General-purpose libraries are provided with the PF1, as it is difficult to satisfy every customer's specific filtering requirements. As more filter libraries are generated, they will be available upon request or can be downloaded from our BBS. TDT can generate custom filter library files for a nominal charge.

Each library file is divided into filters sets. Each set contains filters of a certain order, response type, width (for bandpass & notch) and corner/center frequency placement resolution over a specific frequency range. For a given set, placement may be linear or logarithmic (octave spacing) over the range. Also, pass- or stop-band width (where appropriate) may be in hertz or octaves. Within a set, filter coefficients are accessed by specifying a desired corner/center frequency (with optional tolerance). The contents of a file can be listed with PF1BFILT.

The filters in a particular set have been designed for a specific numerical implementation on the PF1, i.e. FIR16, FIR32, IIR32, BIQ16, or BIQ32. Therefore, the implementation type code is <u>not</u> explicitly specified when using PF1BFILT to load the PF1 with coefficients

Using PF1BFILT

At the DOS prompt you can type PF1BFILT without any parameters to get information on proper usage. PF1BFILT can be used to list or print information about the filter sets in a library file, and also the frequencies available in a set. The program can then be used to download and implement a filter on the PF1.

To list the filter sets in a library file, use

PF1BFILT *fileprefix* (screen display)

or

PF1BFILT *fileprefix* > LPT1 (to send to printer)

Parameters:

fileprefix library file (.bin extension is assumed)

To list the frequencies available in a filter set, use

PF1BFILT *fileprefix* *set#* (screen display)

or

PF1BFILT *fileprefix* *set#* > LPT1 (to send to printer)

Parameters:

fileprefix	library file (.bin extension is assumed)
set#	filter set number

To download a filter from a set to the PF1, use

PF1BFILT *din* *fileprefix* *set#* *freq* [*deltaf*]

Parameters:

din	PF1 device index number
fileprefix	library file (.bin extension is assumed)
set#	filter set number
freq	corner/center frequency
deltaf	frequency match tolerance (optional-- if none specified, uses closest match)

Example: With the filter library file "nt_bp_12.bin" in the current directory, type

PF1BFILT 1 nt_bp_12 3 1200 <enter>

The program searches through filter set number 3 in "nt_bp_12" for a filter with center/corner frequency closest to 1200 Hz and downloads the coefficients for this filter to the PF1.

The program always searches for the best frequency match. If a match tolerance is specified, the program suspends loading only if the best frequency match is not within *deltaf*-Hz of *freq*, and prompts the user to determine whether the closest match is acceptable.

Notes/Warnings on Filter Coefficient Libraries

TDT has tried to ensure stability when generating coefficients for the library files. However, digital filters are subject to coefficient roundoff and overflow errors. The lowpass and highpass filters tend to be most stable, even at maximum input levels. However, bandpass and notch filters may exhibit overflow problems when the filter input is a large tone near a filter corner frequency. In most circumstances however, the input signal will be broad band with peak amplitude less than 10V, so overflow should not normally be a problem. If you suspect a filter is experiencing overflow reduce the level of the PF1 input signal until the overflow stops.

The PF1bfilt Function Call

The file PF1_LIB.C in the "SOURCE" directory contains the PF1bfilt function call. This call incorporates the functionality of the PF1BFILT.EXE loader directly into user application software. Simply add PF1_LIB.C to your program project list and include a prototype of the function call to your program header.

FIR.EXE and XFUNC.EXE

The FIR.EXE program is a Microsoft Windows program that generates FIR filters from a transfer function that you draw on the screen. FIR generates the filter based on the sampling rate and number of taps you specify, and superimposes the actual filter shape on the filter you specified. You can save the filter coefficients to disk for use by other programs. The FIR program includes a complete on-line help file that describes how to use the program.

The XFUNC.EXE ("transfer function") program plays out a series of clicks, noise bursts or tones, then averages the system response to those signals. You can verify the filtering action of a PF1 by installing it in the signal path between the D/A output and the A/D input. Then click the Run menu option to play a series of signals through the system. When signal presentation is complete, XFUNC displays the time, magnitude, or frequency response of the system you have connected.

System Requirements:

- TDT's AP2 Array Processor
- Version 1.28 (or higher) of APOS onboard code (AP.OBJ)
- Microsoft Windows 3.1 (or higher)
- PF1 Programmable Filter

for XFUNC.EXE only:

- D/A and A/D device

.