

Programming the PD1 POWER SDAC –Version 1.0

Copyright

© 1994 TDT. All rights reserved.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose without the express written permission of TDT.

Licenses and Trademarks

Microsoft and Windows are registered trademarks of Microsoft Corporation.



Contents

Preface

Chapter 1 PD1 Basics

PD1 Architecture Overview 1-1

PD1 Resources 1-4

Analog I/O Interface 1-6

AP2 – PD1 Data Interface 1-6

Convolver DSPs 1-7

DP2 Delay Processor 1-7

Inherent Resource Delays 1-9

Chapter 2 Programming Overview

Example 2.0: A First PD1 Application 2-2

Clearing the PD1 2-2

Specifying Conversion Parameters 2-3

Defining the Routing Schedule 2-3

Setting-up the Delay Processor 2-5

Setting-up the DSPs (convolvers) 2-6

RUNning the PD1 2-7

Dynamically Controlling PD1 Resources 2-7

Stopping the PD1 2-7

Putting it All Together 2-8

Example 2.1: Using an IB to Carry Signal Data 2-10

Clearing the PD1 and Specifying Conversion Parameters 2-10

Defining the Routing Schedule 2-10

Setting-up the Delay Processor and DSPs 2-10

RUNning the PD1 2-11

Dynamically Controlling PD1 Resources 2-11

Stopping the PD1 2-11

Putting it All Together 2-11

Example 2.2: Dynamic Updating via an IB Stream 2-12

Clearing the PD1 and Specifying Conversion Parameters 2-12

Defining the Routing Schedule 2-13

Setting-up the Delay Processor and DSPs 2-13

Setting-up to RUN 2-14

Controlling Delay Times 2-15

Putting it All Together 2-15

Chapter 3 PD1 Data Handling

Raw Data 3-1

Resource Control Data (Packets) 3-2

Packet Formats 3-3

Mono Coefficient Data Packet 3-3

Stereo Coefficient Data Packet 3-4

Delay Time Data Packet 3-4

Latching Data 3-5

Organizing Data Packets on IB Streams 3-6

Controlling DAMA Data Flow from the AP2 3-8

Using Play-Pause 3-8

Example 3.0: Signal and Data Packet Control 3-10

Clearing the PD1 3-11

Specifying Conversion Parameters 3-11

Defining the Routing Schedule 3-11

Initializing the Delay Processor 3-12

Setting up the DSPs 3-12

Running the PD1 3-12

Run-time Control of the PD1 3-13

Putting it All Together 3-13

Example 3.1: Using Multiple DSPs to Implement a Long Filter 3-14

Load Long Filter File from disk 3-15

Build Filter Glue 3-15

Define Routing Schedule 3-15

Calculate and Load Long Filter 3-15

Chapter 4 Designing Routing Schedules

Using the Real-Time Router 4-1

What is a route? 4-2

Route Types 4-2

Routing Rules 4-3

Avoiding a Write-Before-Read Problem	4-5
IB and OB Data Stream Routing	4-9
Solving Routing Problems with Isolation Registers	4-10

Chapter 5 The PD1 and 3D Audio

Head Related Transfer Function 5-1

Example 5.0: A 3-D Audio Application 5-2

Clearing the PD1	5-3
Specifying Conversion Parameters	5-3
Defining the Routing Schedule	5-4
Initializing the Delay Processor	5-4
Setting-up the DSPs (convolvers)	5-5
Setting-up to RUN	5-5
Dynamically Controlling Filter Coefficients	5-7
Stopping the PD1	5-7
Putting it All Together	5-7

HRTF Files 5-8

HRTF File Formats	5-8
-------------------	-----

Appendix A PD1 Supplemental Calls

Constants A-1

Function Calls A-2

PreLoadRaw(din, . . .)	A-2
PreLoadHRTF(din, . . .)	A-3
LoadHRTFFile(hrtf, . . .)	A-4
PushHRTF(hrtf, . . .)	A-4

Appendix B Running a Separate A/D Module

Preface

The PD1 POWER SDAC is a powerful addition to the System II family. This versatile convolver provides the processing power to handle the most demanding signal processing models. This document provides the user with the information necessary to implement such models.

Programming the PD1 is similar to programming any other System II device. For this reason, this document focuses on the PD1 only. It is assumed that the user is familiar with the AP2, APOS, and the basic operation of TDT System II hardware.

If you are new to the System II, you may find it helpful to begin by familiarizing yourself with System II programming basics. Basic programming information and a variety of examples can be found in the document *Signal Processing Applications Using TDT System II*.

Chapter 1 PD1 Basics

The PD1 POWER SDAC high performance convolver is the first real-time digital filtering system designed to meet the needs of those designing complex signal processing models. Using Auto-Route, the PD1's windows-based circuit design application, such models can be created with ease. This ease of design, combined with the power and speed of the PD1's real-time convolvers, make the PD1 the ideal platform for the design and implementation of complex signal processing circuits. Used with TDT's acoustic modeling software, Sound Stage, the PD1 is a powerful and easy to use tool for performing research in areas such as sound localization research, binaural hearing aid research, 3D sound generation, architectural acoustics, and transducer correction.

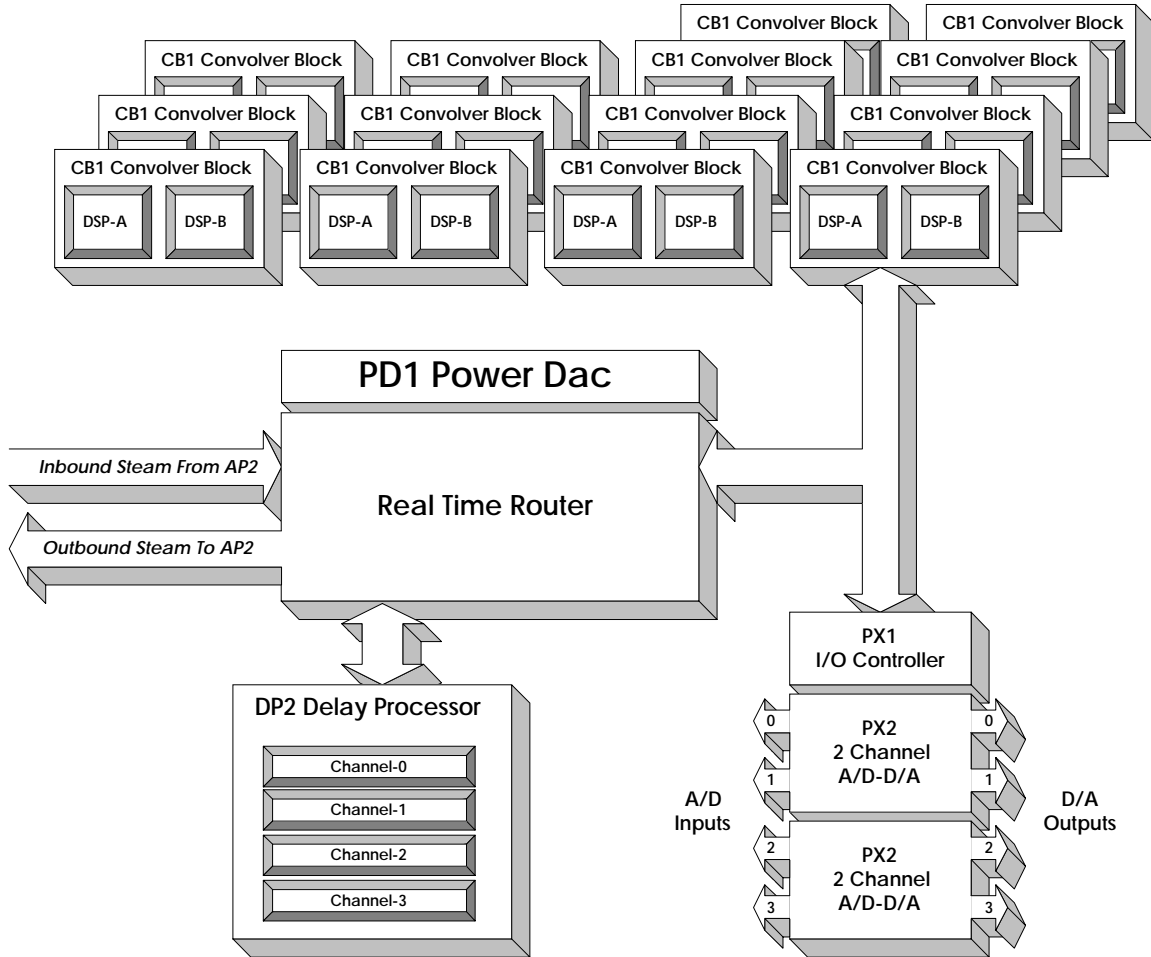
PD1 Architecture Overview

The PD1 is compatible with the TDT System II signal processing platform. Although the module has additional DSP capabilities, it is 100% XBUS compatible and by default will function like other TDT D/A – A/D modules. The PD1 is also supported by System II's auto-detecting DA/AD programming calls.

When performing straight-forward D/A – A/D, the PD1 functions in a manner similar to other TDT devices such as the DD1. Programming the PD1 to perform real-time signal processing tasks is a bit more involved than DD1 programming and will require you to have a basic understanding of the PD1 architecture.

The PD1 POWER SDAC achieves its high level of processing capability by performing numerous tasks in parallel. A PD1 (fully loaded) can have as many as 28 Digital Signal Processors (DSPs) each doing 20 million operations per second. This ability to perform over half a billion operations per second, along with the PD1's specially designed architecture, give the PD1 its amazing processing and dynamic updating abilities.

The PD1 architecture has been designed to offer a powerful and extremely flexible real-time signal processing platform. The PD1 module contains a number of signal processing elements called resources. PD1 resources include D/A converters, A/D converters, convolving DSPs, delay processors, and inbound and outbound data streams. These resources are not "hardwired" into a fixed circuit, but instead can be logically connected into any processing configuration. The PD1 contains a dedicated DSP processor and special addressing hardware to allow the user to effectively "wire-up" the needed circuit. This special hardware is called the **Real-Time Router**. A block diagram of the PD1 architecture is shown below:








- **Real-Time Router** The Real-Time Router (RTR) is the heart of the PD1 architecture. It consists of a DSP and special addressing hardware. It is the RTR that handles the routing of digital data signals to the various PD1 resources. The RTR can be programmed to effectively "wire-up" any circuit between convolvers, analog I/O channels, and delay lines.
- **Analog Interface** The Analog Interface consists of up to four digital to analog (DAC) and four analog to digital (ADC) conversion channels. These input and output channels run from the same PD1 sample clock and are therefore sampled synchronously.
- **Delay Processor** The PD1 delay processor (optional) is a four-channel, 32,768 sample delay line capable of generating arbitrary and dynamically updated signal delays.
- **Convolver** PD1 convolvers are DSPs programmed to perform Finite Impulse Response (FIR) filtering. A PD1 can be equipped with up to 28 convolvers each able to run a 480 tap filter at 40KHz (that's over 12 thousand taps). Each convolver can be made to process a mono or stereo signal and is capable of dynamic updating of filter coefficients.
- **Inbound and Outbound Streams (XBUS Interface)** The PD1's XBUS interface is similar to other TDT D/A – A/D devices in that it uses the bus to send and receive both commands (low speed serial) and data (high speed serial). The PD1 has additional abilities for processing high speed data via the Real-Time Router. The RTR allows the user to send the PD1 filter coefficients, scale factors, and delay times, in addition to waveform data. It is this ability to send filter coefficients across the high-speed XBUS link that allows the PD1 to dynamically update filters in real time.
- **Isolation Register** The isolation register is a temporary holding register. Isolation registers have many signal processing uses. These uses are explained throughout this document.

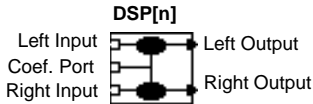
PD1 Resources

As discussed above, the RTR is used to connect the various PD1 resources into a functional signal processing circuit. The input and output ports of PD1 resources are connected with data *routes*. These routes are simply paths over which data flows. For example, connecting an ADC to a DAC with a route will cause the 16-bit data being generated by the ADC to be sent directly to the DAC, thus forming a simple bypass circuit. The collection of routes used to form a PD1 circuit is called a *routing schedule*. The RTR is programmed with the routing schedule using special XBUS commands.

To help simplify the route scheduling process, TDT has developed a simple iconic library to represent the currently available resources of the PD1. This iconic library is supported by AutoRoute, TDT's Windows application. AutoRoute allows the user to graphically design complex PD1 circuits and generate the XBUS commands needed to program the RTR. Refer to the *AutoRoute User's Guide* for more information about this program.

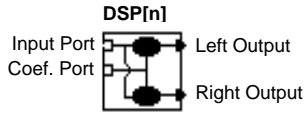
The various PD1 resource icons are shown below. Note that the shape of each icon indicates the function it performs. Each icon contains some number of inputs and outputs called *ports*. Routes are used to interconnect these ports thus forming a circuit. The following diagram shows all icons along with their port specifications.

 <p>ADC[n] Output Port</p>	<p>Analog to Digital Converter (ADC)</p>
 <p>DAC[n] Input Port</p>	<p>Digital to Analog Converter (DAC)</p>
 <p>IB[n] Output Port</p>	<p>Inbound Data Stream (IB)</p> <p>IB streams may be sent from the AP2 to the PD1 across the XBUS link. These streams may consist of raw waveform data or packets composed of filter or delay coefficients.</p>
 <p>OB[n] Input Port</p>	<p>Outbound Data Stream (OB)</p> <p>Data may be output from the PD1 to the AP2 across the XBUS link via an OB stream.</p>
 <p>DSP[n] Input Port Coef. Port Output Port</p>	<p>Convolver DSP running in MONO mode</p> <p>The mono DSP provides a simple means for real-time filtering of single channel data.</p>



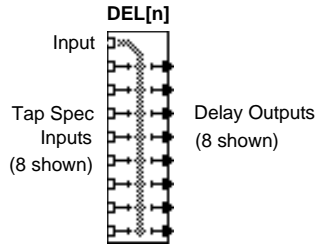
Convolver DSP running in STEREO mode

The stereo DSP provides dual-channel input and output. Separate sets of filter coefficients may be applied to the right and left channels.



Convolver DSP, MONO in, STEREO out

This resource divides a single input into dual-channel output. Separate sets of filter coefficients may be applied to the right and left channels.



Delay Processor Channel

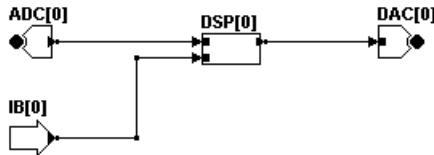
The PD1 delay processor is an optional four-channel, 32,768 sample delay line. Each delay line may contain 8 or 16 taps. The DP2 delay processor hardware actually allows up to 32 arbitrary delay taps, but graphical limitations make the use of a 32 taps icon impractical.



Isolation Register

The isolation register is a temporary holding register. Isolation registers may be used to solve a variety of routing schedule design problems including: violation of routing rules, summing of inbound data stream data, and control of signal flow.

Consider the simple example (shown below) in which an ADC, DSP, and DAC are configured to form a simple single channel filter. In this example, ADC[0] converts the analog signal at its input into a digital signal. The RTR transfers this digital data to the data input port of DSP[0]. The output port of DSP[0] is sent to DAC[0] where it is converted back into an analog signal. Filter coefficients are provided by the AP2 via inbound data stream IB[0].



As stated previously, the RTR is programmed with simple XBUS calls. The calls required to program the circuit shown above are as follows:

- 1) PD1clrsched(din);
- 2) PD1addsimp(din, DSPout[0], DAC[0]);
- 3) PD1addsimp(din, ADC[0], DSPin[0]);
- 4) PD1specIB(din, IB[0], COEF[0]);

The first line 1) clears any previously specified routes from the router schedule. Line 2) specifies the connection from DSP[0]'s output to DAC[0]'s input. The third line 3) adds the connection from ADC[0]'s output to DSP[0]'s input. Finally line 4) specifies that IB[0] should be connected to the COEF input of DSP[0].

Analog I/O Interface



Each PD1 is equipped with an Analog I/O module. Currently, two and four channel options are available. With either option, high quality 16-bit audio DACs and ADCs are used.

A dedicated DSP controller aids in the interfacing of the Analog I/O channels with the PD1's Real-Time Router (RTR). This DSP also monitors signal level on all channels, flashing LED indicators on the PD1's front panel to indicate various levels of signal activity.

Two XBDRV calls are provided which program the PD1's I/O Interface. **PD1clrIO** will flush the PD1's data registers and zero its DAC outputs. Thresholds for the LED activity indicator can be set using the **PD1setIO** call.

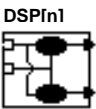
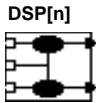
AP2 – PD1 Data Interface



Data is sent to and received from the PD1 via the AP2 on the XBUS fiber optic interface. In many respects, this interface is identical to the high speed data link supported by other TDT D/A–A/D converter modules. However, in addition to its ability to handle raw waveform data, the PD1 can also receive data in an encapsulated format known as a *packet*. Packets contain information used to route incoming data to the proper resource. Thus, the PD1 uses inbound (IB) and outbound (OB) data streams to aid in the handling of data movement between the PD1 and the AP2.

IB and OB streams are discussed at length in the following sections of this document.

Convolver DSPs



Each PD1 can be equipped with up to 28 DSPs. Each DSP can filter digital signals under user control. A DSP can be programmed to run in either MONO or STEREO mode.

Filter coefficients are sent to the PD1 via the device's coefficient input port (Coef. Port). Data streamed to this input can be used to specify device operation mode and to supply the filter coefficients to be used during digital filtering. DSPs can also be loaded with coefficients prior to running. These coefficients can then be locked, preventing further updating.

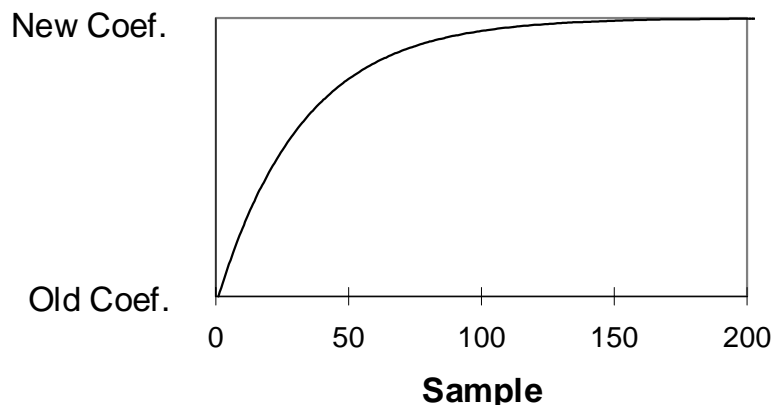
Five calls are provided in the XBDRV library for controlling the PD1's DSP resources. **PD1idleDSP** and **PD1bypassDSP** can be used to place one or more DSPs in a "nopass" or "all pass" mode during program trouble-shooting. **PD1resetDSP** is used to reset a DSP and prepare it to receive filter information via its coefficient input. **PD1lockDSP** can be used to lock the current set of filters into a DSP. Once locked, the DSP will ignore information sent via its coefficient input as well as any GSYNC and LSYNC calls. The DSP will remain locked until the **PD1resetDSP** call is issued.

Each DSP can perform optional coefficient interpolation. The feature is enabled using the **PD1interpDSP** call. When coefficient interpolation is enabled for a DSP, it will no longer instantaneously switch to new filter sets as they are loaded. Instead an exponential-like interpolation will be used.

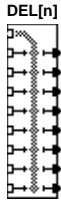
At time $t=0$, a new coefficient set is latched into a DSP. On each successive tick of the sample clock, the coefficients used in filtering migrate from the previous set towards the new set. The slope of this interpolated transition is controlled by the *interpolation factor* argument in the **PD1interpDSP**. The equation for calculating the current coefficients is :

$$\text{Current Coefficients} = \left(\frac{\text{InterpolationFactor}}{32768} \times \text{New Coefficients} \right) + \left(\left(1 - \frac{\text{InterpolationFactor}}{32768} \right) \times \text{Previous Coefficients} \right)$$

The curve shown below shows how the filter coefficients would change with an interpolation factor=1000.



DP2 Delay Processor



The PD1 can be optionally equipped with the DP2 multi-channel delay processor. This PD1 option, with its dedicated DSP processor, can generate multiple arbitrary delays on each of four channels. In addition, delay tap times can be updated dynamically via inbound data streams from the AP2. This functionality allows the DP2 to be used to implement complex and virtual reverberation and echo simulations.

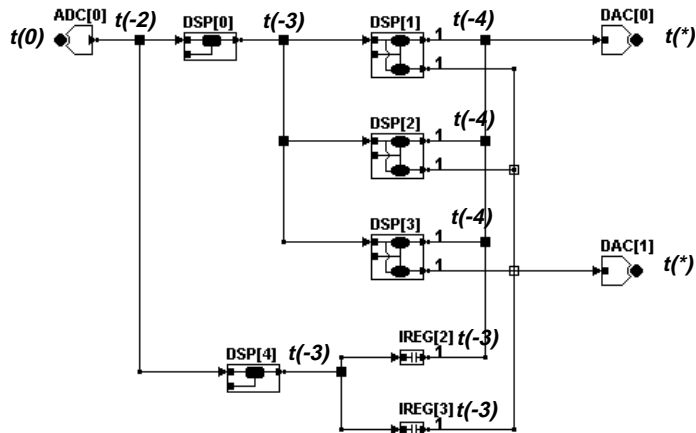
Signals routed to a DP2 input channel are stored in a revolving 32K sample memory bank. Specific time-delayed "taps" can then be made to appear at the DP2's outputs. The amount these taps are delayed is controlled by the TAP[0..31] inputs. These inputs can be preprogrammed using the **PD1setDEL** call or programmed dynamically using an inbound data stream.

While dynamically updating delays, the DP2 will drop or repeat samples as it adjusts for the newly programmed delay times. For example, consider what happens if a delay time changes from 10ms to 20ms. If the PD1 is sampling at 50KHz (20 μ s), 500 samples must be dropped, causing a discontinuity in the output waveform. This output distortion can be minimized by using the DP2's internal delay interpolator. When the delay interpolator is active, the DP2 will drop or repeat samples over a specified period of time until the needed delay time is reached. The interpolator is enabled using the **PD1interpDEL** call. The period of interpolation is controlled by a single parameter used to specify the number of sample clock ticks over which the delay should be updated.

*Refer to the [XBDRV Software Reference](#) for more information about **PD1interpDEL**.*

Inherent Resource Delays

Most PD1 resources have at least a one sample delay. For example, signals routed to a DSP's input port will not appear at its output port until one sample cycle later, even when the device is placed in bypass mode. The PD1 circuit shown below illustrates how resource delays will affect signals as they are routed through the PD1. Assume all DSPs are in bypass mode.



At time $t(0)$, the A/D device samples the waveform input. After a two sample delay, this value is written to the inputs of DSPs [0] and [4]. Another sample delay is introduced when the signal is output from these DSP, creating a total delay of 3 samples. The $t(-3)$ output from DSP[0] is then sent to three DSPs, where it is delayed another sample to $t(-4)$. The output from DSP[4] is sent to IREG[2] and IREG[3]. Because isolation registers introduce no delay, the values output from DSPs [1], [2], and [3] will have a one sample cycle discrepancy with the values output from IREGs [2] and [3] when these signals are mixed at multi-routes [A] and [B]. The resulting analog waveform $t(*)$ will delay the signal fed to its input by two more sample cycles, resulting in an overall circuit delay of 5 to 6 samples. At 50KHz sampling, this only amounts to about 100 μ s. However, in some situations, the cycle mismatch could be more troublesome. The problem becomes more convoluted when filters with different group delays are employed.

For many applications, this small amount of delay can be ignored. Often, PD1 circuits are symmetric, nullifying the effects of any delays for time-critical binaural processes. However, one should be aware of these delays and assess their potential impact on PD1 performance. A list of PD1 resources and their respective delays are shown in the table below:

Resource	Delay
ADC[*]	2 samples
DAC[*]	2 samples
IB[*]	2 samples
OB[*]	2 samples
DSP[*] (any mode)	1 sample + filter delay
DEL[*]	programmable
IREG[*]	0 samples

Chapter 2 Programming Overview

For more information about APOS and XBDRV function calls, see the appropriate software reference manuals.

Using the comprehensive software drivers provided with the PD1, it is possible to design a variety of signal processing applications written in 'C' or Pascal. Such applications will include standard APOS and XBDRV function calls, PD1 calls, and user code designed to implement any special processing you desire. This section explains only the use of the specialized PD1 calls. These calls are described in detail in the *XBDRV Software Reference Manual*.

Programming the PD1 is very similar to programming any System II device. However, because the PD1 supports real-time signal processing functions, the basic structure of user programs will change slightly to support this ability.

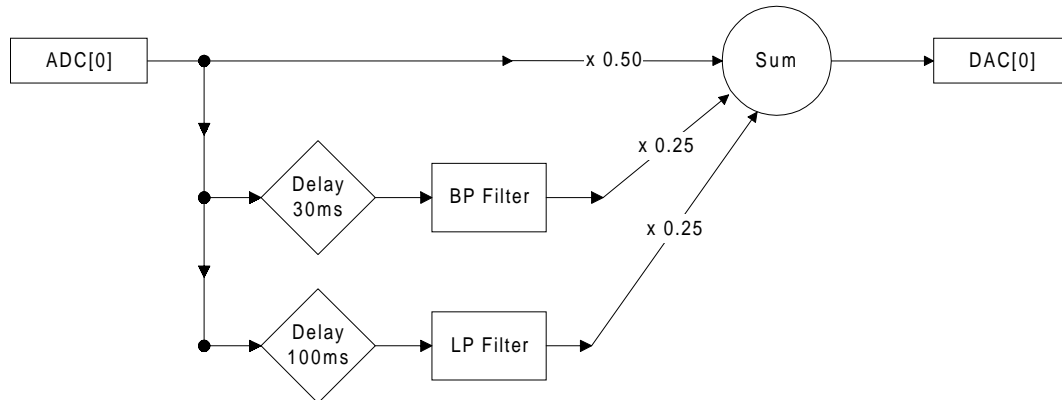
Like other TDT, D/A–A/D devices, the PD1 operates in two basic modes. When in IDLE mode, the PD1 can be programmed using any of the PD1 calls listed in the XBDRV document as well as those included in PD1_SUP. PD1 parameters should always be programmed while the PD1 is in IDLE mode. Once initialized, the PD1 can be placed in RUN mode. In this mode, the conversion clock is running and the PD1 can only be programmed via inbound (IB) data streams.

A typical PD1 program will include the following steps:

Step	When Required	PD1 Mode
1. Clear the PD1.	All applications	IDLE
2. Specify conversion parameters.	All applications	IDLE
3. Define the routing schedule.	All applications	IDLE
4. Setup the delay processor.	Applications using the delay processor	IDLE
5. Setup DSPs and/or preload with filter coefficients.	Applications using DSPs	IDLE
6. Start the PD1 RUNning using PD1arm/PD1go.	All applications	RUN
7. Manipulate PD1 resources using IB streams.	If dynamic updating required	RUN
8. (1) Stop the PD1 using PD1stop or (2) allow the PD1 to run for a specified number of samples (PD1npts) and loop back to reconfigure.	All applications	IDLE

Example 2.0: A First PD1 Application

The best way to explain each of these programming steps is via a simple example. As a first illustration, an example that does *not* use dynamic updating will be described. Suppose we wish to realize a simple circuit that delays, filters, and sums, such as the one shown below:



In this example, analog signals digitized by ADC[0] will be delayed, filtered, and added back to the original signal as shown. The resulting data will be converted back to an analog waveform via DAC[0].

The following steps develop a 'C' program for implementing this functionality on the PD1.

Step 1 Clearing the PD1

```
PD1clear(1);
```

Prior to calling any other PD1 function, it is necessary to execute **PD1clear**. This function call clears the PD1, resetting it to factory default settings and initializing all PD1 variables. This call also executes **PD1resetRTE**, **PD1clrIO**, and **PD1clr sched**, resetting the Real-Time Router, the analog interface, and the routing schedule.

Note: A set of software variables is used when programming the PD1. These variables are initialized by **PD1clear**. It is important that you call **PD1clear** before using any PD1 variables or making any other PD1 calls.

Step 2 Specifying Conversion Parameters

```
PD1nstrms(1, 0, 0);
PD1srate(1, 20.0);
PD1npts(1, -1);
```

There are three basic PD1 conversion parameters that must be initialized. The **PD1nstrms** call is used to specify the number of inbound (IB) and outbound (OB) streams needed. In this example, we will not be using any IB or OB data streams. IB and OB streams need only be used when you must dynamically update delay times and/or filter coefficients or when you will be using waveform data stored on the AP2. The sample rate and number of conversion points must also be specified using the **PD1srate** and **PD1npts**, respectively. For our application, we will convert at 50KHz (20 μ s) continuously, until we call **PD1stop**.

Step 3 Defining the Routing Schedule

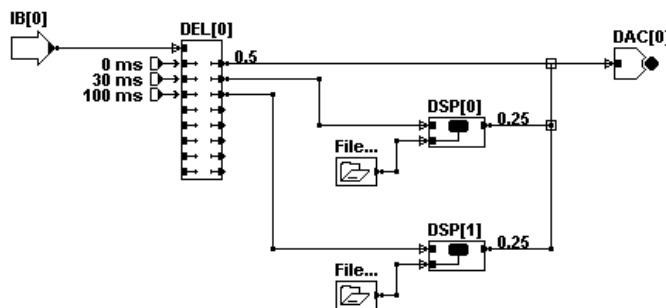
For more information about designing routing schedules see Chapter 4 of this document and the [AutoRoute User's Guide](#).

```
PD1clrshched(1);
PD1addsimp(1,
  ADC[0],DELin[0]);
PD1addsimp(1,
  DELout[1][0],DSPin[0]);
PD1addsimp(1,
  DELout[2][0],DSPin[1]);
float sf[3];
int src[3];
src[0]=DELout[0][0];
sf[0]=0.5;
src[1]=DSPout[0];
sf[1]=0.25;
src[2]=DSPout[1];
sf[2]=0.25;
PD1addmult(1,
  src, sf, 3, DAC[0]);
```

The routing schedule provides an instruction list to the Real-Time Router, effectively "wiring up" the various PD1 resources into a functional circuit. Each PD1 application must contain code to program the routing schedule.

You may code the routing schedule "by hand" or generate it through AutoRoute.

The PD1 circuit needed to implement our application is shown below:



The DP2 delay processor (channel-0) is used to generate the delays. The two delayed signals are then each routed through a DSP and summed using the desired scale factors. The resulting digital signal is fed to the DAC.

Note: The above diagram indicates that the filter coefficients for both DSPs will be loaded from disk. The DSPs will not actually be loaded and initialized until Step 5.

Types of Routes

The PD1 RTR supports four type of routes:

1. Simple routes
2. Multi-routes
3. Inbound data streams
4. Outbound data streams

Each of these types is supported by an XBDRV call.

Simple routes are used to connect a single resource output port to a single resource input port. In our example, simple routes are used for the following:

1. The connection from ADC[0] to the DEL[0] input
2. The input of DSP[0]
3. The input of DSP[1]

Multi-routes are actually summing junctions whereby multiple resource outputs can be weighted (scaled), summed, and sent to a resource input port. A multi-route is used in our example to sum the two delayed/filtered signals with the original signal. The result is sent to DAC[0].

For more information about dynamic versus static routing of data streams, see Chapter 4 "Using Inbound Data Streams."

Inbound (IB) and outbound (OB) streams *cannot* be the source or destination of either multi or simple routes. Instead their data ports are routed using two special calls: **PD1specIB** and **PD1specOB**. Inbound data streams can also be routed dynamically. Note that our example program does not utilize IB or OB data streams.

Step 4 Setting-up the Delay Processor

```
PD1clrDEL(1, 3, 0, 0, 0);
PD1setDEL(1, TAP[0][0],
(int)(0.0 * 50.0));
PD1setDEL(1, TAP[0][1],
(int)(30.0 * 50.0));
PD1setDEL(1, TAP[0][2],
(int)(100.0 * 50.0));
PD1latchDEL(1);
```

In applications using the delay processor, a certain amount of basic setup is required. The necessary code can be programmed "by hand" or generated through the use of AutoRoute.

The DP2 contains a dedicated DSP for processing and optionally interpolating time delays. The number of delays the DP2 can process is limited and is inversely proportional to the sampling rate. When initializing the DP2 using **PD1clrDEL**, the number of delay taps needed for each channel is specified. This number must not exceed the device limit indicated on the PD1 data sheet. For our example, we need three delay taps.

DP2 delay times can be specified dynamically using IB streams or pre-programmed using the **PD1setDEL** command. In this example, we will use the **PD1setDEL** call to program the three delay times and latch them into the DP2 using **PD1latchDEL**.

Note: **PD1setDEL** expects delay times to be specified in terms of *number of samples*. To convert our delay times from milliseconds we must apply the following equation:

$$\begin{aligned} n_{\text{samples}} &= (\text{delay ms}) * 1000 / (\text{sample period } \mu\text{s}) \\ &= (\text{delay ms}) * 1000 / 20 \mu\text{s} \\ &= (\text{delay ms}) * 50 \end{aligned}$$

Note: The **PD1flushDEL** call is not included in the example program because it is called automatically by **PD1clrDEL**. The **PD1flushDEL** call should be used when you need to clear signal data from the DP2's memory, but don't want to clear delay time specifications and/or other device setups.

Note: The DP2's delay interpolator option was not enabled using the **PD1interpDEL** call. You will only need the delay interpolator if delay times will be updated dynamically via IB streams.

Step 5 Setting-up the DSPs (convolvers)

```
PD1idleDSP(1, 0x0FFFFFFF);  
  
PreLoadRaw(1, DSPid[0],  
           MONO, FILE_A, "BP.FIR",  
           "", 1.0, 1.0, 1);  
  
PreLoadRaw(1, DSPid[1],  
           MONO, FILE_A, "LP.FIR",  
           "", 1.0, 1.0, 1);
```

The PD1's DSPs can be run in a variety of modes and can be loaded with filter coefficients in several ways.

Modes

A number of XBDRV calls are used to program the PD1's DSPs. They include: **PD1bypassDSP**, **PD1idleDSP**, **PD1resetDSP**, **PD1interpDSP**, and **PD1lockDSP**. **PD1bypass** and **PD1idle** are used primarily in debugging, allowing the user to place a DSP in bypass or nopass mode. If a DSP is going to be utilized in dynamic update mode the **PD1resetDSP** call is made early in the program to ready the DSP for dynamic updating and control.

Loading Coefficients

Filter coefficients may be loaded two ways:

1. Static loading

In static loading, the DSPs are pre-loaded with filter coefficients and then 'locked'.

or

2. Dynamic updating

In dynamic updating, new filter coefficients are constantly fed to the DSP's coefficient input port via inbound data streams.

When static loading is being employed (as in our example), the filter coefficients can be loaded using a variety of techniques. The simplest way to get coefficients pre-loaded to DSPs is via the supplemental PD1 calls included in the PD1_SUP drivers software. Refer to Appendix A for more information on PD1 supplemental calls.

See [Digital Signal Processing Applications](#) for more information about generating coefficients.

In our example, the filter coefficients are being loaded from two ASCII data files one called "BP.FIR" and the other called "LP.FIR".

Note: In this example, the **PD1idleDSP** call is unnecessary. It is included to ensure that any unused DSPs will remain in IDLE mode.

Step 6 RUNning the PD1

```
PD1arm(1);
PD1go(1);
```

Once all required system initialization is complete, we can begin running the PD1 by issuing two XBDRV calls, **PD1arm** and **PD1go**. If external device triggering is required, the **PD1go** command should be omitted.

The PD1 will continue to RUN conversion cycles until the *npts* value is reached. This value was programmed earlier with **PD1npts**. In our example this is very large number and would take about 24 hours. We are, therefore, running in continuous mode.

Note: In cases where IB or OB streams will be used, it is necessary to issue an APOS *play* and/or *record* call, as is typical with other TDT A/D – D/A devices.. Because no IB or OB streams are being used in our example, there is no need to make these calls.

Step 7 Dynamically Controlling PD1 Resources

```
/* NOP */
```

In this example, dynamic updating is not used. Refer to Example 2.1 for a demonstration of this feature.

Step 8 Stopping the PD1

```
gets(dump);
PD1stop(1);
PD1clrIO(1);
```

By calling **PD1npts(1, -1)**, we placed the PD1 in an approximation of continuous convert mode. We must, therefore, have some means of stopping the PD1. This is usually done after some user-selected period of time. The code shown on the left stops the PD1 when the user presses the [RETURN] key.

Note: When **PD1stop** is issued, the PD1 output voltages hold the last converted voltage level. In order to zero the PD1 outputs, a call to **PD1clrIO** must be included .

Putting it All Together

So far, this example has illustrated the processes of programming the PD1 in a step-by-step manner. In this section, the complete program is presented. It is highly recommended that you compile and run this example program.

If your PD1 is not equipped with a DP2, simply omit that element from the circuit and skip Step 4. Once you have the program running, try manipulating various parameters and note the resulting device *operational* changes. The complete program listing is included below and provided on disk (c:\tdt\drivers\pd1sup\doce exams\exam20).

Note: The simple PD1 example described here illustrates a variety of points associated with programming the PD1. It does *not*, however, address all nuances associated with the implemented circuit. For example, no mention was made of the additional delay introduced by the use of band-pass and low-pass filtering. To account for this additional delay one might include a third DSP performing all-pass filtering that is phase matched to the band-pass and low-pass filters.

```

/*****
/* PD1 Users Guide Example 2.0
/*****
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#include "apos.h"
#include "xbdrv.h"
#include "pd1_sup.h"

void main()
{
    float sf[3];
    int src[3];
    char dump[10];

    if(!apinit(APa) && !apinit(APb))
    {
        printf("\n\nAP Init Error\n\n");
        exit(0);
    }

    if(!XB1init(USE_DOS))
    {
        printf("\n\nXBUS Init Error\n\n");
        exit(0);
    }
}

```

```

/* Step-1 */
PDlclear(1);
PDlfixbug(1); /* This fixes bug in PD1 */

/* Step-2 */
PDlstrms(1, 0, 0);
PDlstrate(1, 20.0);
PDlnts(1, -1);

/* Step-3 */
PDlclrsched(1);

PDladdsimp(1, ADC[0], DELin[0]);
PDladdsimp(1, DELout[1][0], DSPin[0]);
PDladdsimp(1, DELout[2][0], DSPin[1]);

src[0]=DELout[0][0]; sf[0]=0.50;
src[1]=DSPout[0];   sf[1]=0.25;
src[2]=DSPout[1];   sf[2]=0.25;
PDladdmult(1, src, sf, 3, DAC[0]);

/* Step-4 */
PDlclrDEL(1, 3, 0, 0, 0);
PDlsetDEL(1, TAP[0][0], 0);
PDlsetDEL(1, TAP[1][0], 30 * 50);
PDlsetDEL(1, TAP[2][0], 100 * 50);
PDllatchDEL(1);

/* Step-5 */
PDlidleDSP(1, 0xFFFFFFFF);
PreLoadRaw(1, DSPid[0], MONO, FILE_A,
            "BP.FIR", "", 1.0, 1.0, 1);
PreLoadRaw(1, DSPid[1], MONO, FILE_A,
            "LP.FIR", "", 1.0, 1.0, 1);

/* Step-6 */
PDlarm(1);
PDlgo(1);

/* The PD1 is now running */

printf("Example #1 is running\n");
printf(" Press [RETURN] to quit...\n");
gets(dump);

/* Step-8 */
PDlstop(1);
PDlclrIO(1);
}

```

Example 2.1: Using an IB to Carry Signal Data

Example 2.1 presents a modified version of Example 2.0. In Example 2.0, ADC[0] was the source of digital data used in our circuit. You may want to use digital waveforms stored on the AP2 or on a PC hard disk as a source of data. We can create a circuit that uses such digital source data by modifying Example 2.0 such that ADC[0] is replaced with the statically routed stream IB[0]. The required modifications to the program are illustrated below.

Steps 1 & 2 Clearing the PD1 and Specifying Conversion Parameters

Steps one and two from Example 2.0 will remain unchanged, with one exception: the PD1instrms call in Step 2 must be changed to:

```
PD1instrms(1, 1, 0);
```

Step 3 Defining the Routing Schedule

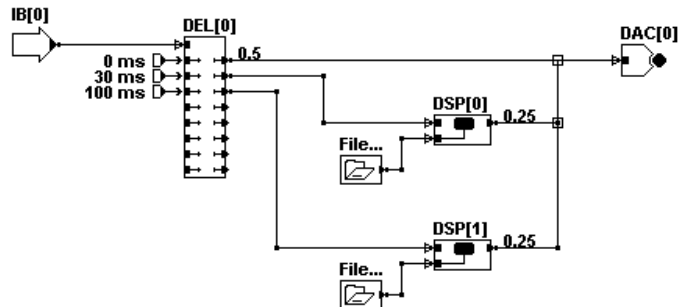
```
PD1clr sched(1);

/* Replace this call */
PD1addsimp(1,
  ADC[0], DELin[0]);

/* With... */
PD1specIB(1,
  IB[0], DELin[0]);

/*
All other calls are the
same as in example #1.
*/
```

The new circuit will be similar to the one shown in example 2.0, except for the replacement of ADC[0] with IB[0]:



Note: The simple route connecting ADC[0] to DELin[0] will be replaced with the statically routed inbound stream, IB[0]. All other routing calls remain unchanged.

Steps 4 & 5 Setting-up the Delay Processor and DSPs

Steps 4 and 5 will not need modification. It is important to remember that we will be passing only waveform data on IB[0]. Filter coefficient data must still be pre-loaded from the PC.

Step 6 RUNning the PD1

```
allot16(1, 100000);
dpush(100000);
gauss();
scale(5000.0);
qwind(10.0, 20.0);

qpop16(1);
play(1);
PD1arm(1);
PD1go(1);
```

Because the AP2 will now be generating the signal data used in our application, we need some additional code before we initiate conversion on the PD1. This additional code could be placed anywhere prior to the arm and go in Step 6. In our example it will be added to Step 6.

We will use noise as our signal. The noise is generated on the AP2 using the **gauss** call and windowed using the **qwind** call. The signal is then popped to a 16-bit DAMA buffer (ID=1). The single channel **play** call is issued before the **PD1arm** call. This tells the AP2 to send a single stream of data using DAMA buffer ID=1 as the source.

Note: Typically when working with signals of finite length, the PD1 would not be programmed to convert continuously. For simplicity we will leave our program as is.

Step 7 Dynamically Controlling PD1 Resources

Step 7 of Example 2.1 does not require any modification to the code presented in Step 7 of Example 2.0. However, if you wanted to "freshen" the noise, you would want to place the code in this section.

Note: Refreshing noise requires setting up a double buffer system. Refer to the document *Signal Processing Applications using TDT System II* for more information on double buffering.

Step 8 Stopping the PD1

Step 8 of Example 2.1 does not require any modification to the code presented in Step 8 of Example 2.0.

Putting it All Together

A complete listing of this program is available on disk (c:\tdt\drivers\pd1sup\docexams\exam21).

Example 2.2: Dynamic Updating via an IB Stream

As stated previously, IB streams can be used to carry not only waveform data but other information as well. For example, we can modify our program in Example 2.0 to include an IB stream carrying delay time information. This will allow us to modify delay tap times while the PD1 is RUNNING. The ability to dynamically alter/update system resources is what gives the PD1 its unique processing abilities.

The information sent via inbound data streams is encapsulated in data records called packets. A packet contains not only the values to be sent, but also information about the PD1 resource port to which the data should be sent. Because the destination information is imbedded in each packet, there is no need to statically route these IBs using **PD1specIB**. A typical IB packet is shown below. This record will setup and route a 1000 sample delay to DP2 channel-1, tap-4.

TAP[4][1]	_START	1000	_STOP
-----------	--------	------	-------

Typically, packets include a destination address followed by the **_START** marker, data, and the **_STOP** marker. Basically, the **_START** and **_STOP** markers act as a valve. **_START** switches on the flow of data to the designated resource port. **_STOP** acts as if it switches off the flow of data. In reality, **_STOP** never actually turns off the data flow; it simply directs the data to a DUMP location.

To illustrate the use of IB streams for controlling PD1 resources, let's modify Example 2.0 to allow for dynamic updating of DP2 delay times. In our new example, the program will allow the user to scroll through delay times using the [1], [2], [9] and [0] keys. The delay time for the bandpass-filtered signal will be controlled with the [1] and [2] keys while the delay time for the lowpass-filtered signal will be decreased and increased using the [9] and [0] keys, respectively.

Steps 1 & 2 Clearing the PD1 and Specifying Conversion Parameters

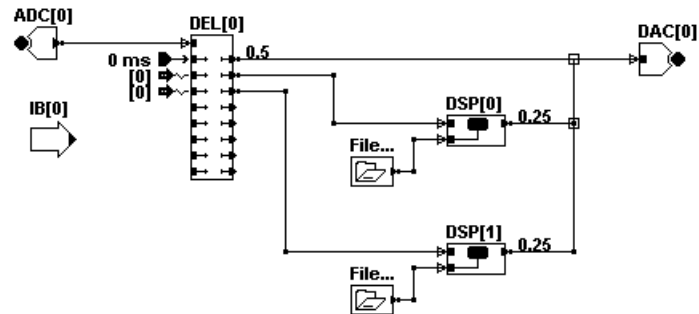
As in Example 2.1, we only need to change the **PD1nstrms** call in Step 2. The call should be programmed as follows:

```
PD1nstrms(1, 1, 0);
```

Step 1 will remain unchanged.

Step 3 Defining the Routing Schedule

The routing schedule programmed in Example 2.0, Step 3, will not require changes. The resulting circuit, however, is illustrated in a slightly different manner. In the diagram below, an IB stream icon is used to indicate that the resource ports are to be programmed by the IB stream.



Note: IB[0] is not connected to any resource port, instead it simply appears on the diagram along with two marks showing the two ports to which it will eventually need to send data.

Steps 4 & 5 Setting-up the Delay Processor and DSPs

Step 4 can be eliminated, with the exception of the call, **PD1clrDEL**. Step 5 will remain unchanged.

Step 6 Setting-up to RUN

In this example, we will program the DP2 via IB[0]. In order to accomplish this, we must modify Step 6. Specifically, we must initialize the DAMA buffer and then build the packet.

```
int del1, del2;

del1 = 30 * 50;
del2 = 100 * 50;

allot16(1, 20);

dpush(20);
value(0.0);

make(0, TAP[1][0]);
make(1, _START);
make(2, del1);
make(3, _STOP);

make(10, TAP[2][0]);
make(11, _START);
make(12, del2);
make(13, _STOP);

make(15, GSYNC);

qpop16(1);

play(1);
PD1arm(1);
PD1go(1);
```

Building a Packet

There are a variety of ways to build the data packets needed for resource control. One of the most efficient ways is to setup the record format in DAMA and simply update the data portion of the record using **makedama16** and/or **poppart16**.

In this example we employ this method. Note all "glue" associated with both delay specification records is setup along with initial delay values.

Modifying a Packet

Later in Step 7, we will simply modify the 2nd and 12th element of DAMA buffer ID=1 to change delay tap times.

Setting the DAMA Buffer Length

Once running, the AP2 will loop continuously through the buffer, sending the same data over and over again. Because of this continuous looping, you can think of buffer duration as the frame rate for your updating. In this example, the buffer is $20 * 50\mu\text{s} = 1\text{ms}$ long. This means the fastest we can change delay times is 1000 times per second. If this loop time was too long, the DAMA buffer could be shortened to as few as 10 points.

Note: In this example, the 20-element DAMA buffer is longer than necessary. It is made longer than is required to help clarify programming.

Latching the Data

The 15th element of the DAMA buffer is loaded with the GSYNC marker. This marker will cause the previously programmed delay times to be latched into the DP2. All dynamically programmable PD1 resources can be loaded asynchronously and then latched using a GSYNC or LSYNC marker.

Step 7 Controlling Delay Times

```

printf("Example #1b is
      running\n");
printf("  Press:
      [1],[2],[9],[0], or
      [x]
      to quit...\n");
do
{
  c = _getch();
  switch(c)
  {
    case '1':
      dell -= 10*50;
      break;
    case '2':
      dell += 10*50;
      break;
    case '9':
      del2 -= 10*50;
      break;
    case '0':
      del2 += 10*50;
      break;
  }

  makedama16(1, 2, dell);
  makedama16(1, 12,
del2);

  printf("\n Delay-1:
        %3.1f      Delay-2:
        %3.1f      ",
        0.02*dell,
0.02*del2);
}while(c!='x');

```

While the PD1 is RUNNING, the PC must update delay times based on user input. The code shown on the left will alter the 2nd and 12th value of the controlling DAMA buffer based on user input.

Note: No range checking is done, allowing the user to enter negative delay times. In your applications, you will want to ensure that delay values are in the range 0 to 32760.

Note: In this example, we alter a DAMA buffer while it is being sent to the target resource. Under most circumstances this can create a potential problem. However, because we are altering only one value, the potential for conflict is nil.

Putting it All Together

A complete listing of this program is available on disk (c:\tdt\drivers\pd1sup\docexams\exam22).

Chapter 3 PD1 Data Handling

The PD1 can process data sent from the AP2 in much the same way as other TDT D/A–A/D devices. You simply match the number of inbound (IB) and outbound (OB) data streams programmed on the PD1 to an appropriate AP2 processing setup. For example, if you want the PD1 to function as a two channel D/A with single channel A/D, you would call **PDInstrms**(1, 2, 1) along with **dplay**(x, y) and **record**(z). The first call instructs the PD1 to use two inbound streams and one outbound stream, while the latter calls tell the AP2 to process two play buffers and one record buffer. This matching of processing setup is essential to proper device operation.

Raw Data

Raw waveform data may be sent and received from the PD1 via data streams. Such data is sent in 16-bit integer format. For each inbound and outbound data stream, source and destination DAMA buffers must be allocated. The function calls **allot16** or **_allot16** are used to allocate these DAMA buffers.

Note: Raw waveform data must be limited to the integer range from +32760 to -32768. Failure to limit waveforms being played from the AP2 will result in unpredictable device operation. The upper limit for integer values is not 32767 as with other TDT devices. Instead this value has been lowered to reserve seven values for packet encoding (see below).

Resource Control Data (Packets)

In addition to handling raw waveform data, the PD1's IB streams can be used to receive "resource control data" or *packets* from the AP2. It is this ability to program the PD1's resources in "real time" that give the device its powerful signal processing capabilities.

These packets contain not only the data to be sent, but also the destination resource port and marker values. These *marker* values flag the PD1 that a data packet is on the way. A typical data packet has the following format:

<i>Port ID</i>	<i>_START</i>	<i>Data</i>	<i>_STOP</i>
----------------	---------------	-------------	--------------

Port ID Refers to the input port of the destination resource. For example, the Port ID for inbound filter coefficient data sent to DSP[0] would be COEF[0]. The current legal Port IDs are COEF[0..27] and TAP[0..31][0..3].

_START A marker used to indicate the beginning of the data transmission.

Data DSP filter coefficients, or DP2 delay times.

_STOP Marker used to terminate the packet transmission.

The PD1 processes IBs by constantly searching for marker values. When a *_START* value is detected, the IB's destination port is switched to the previously received address. For this reason, the *Port ID* must be sent just before the *_START* marker. All subsequent data will be sent to this new address for processing by the destination resource. When the *_STOP* marker is detected the data flow is switched to a DUMP location.

Note: Values contained in the data portion of the packet must be limited to the integer range from +32760 to -32768. Failure to limit data will result in unpredictable device operation. The upper limit for integer values is not 32767 as with other TDT devices. The upper seven values are reserved for packet encoding.

Packet Formats

The PD1 currently supports three data packet formats:

- Mono Coefficient Packets
- Stereo Coefficient Packets
- Delay Tap Packets.

Each of these packet formats conforms to the basic packet structure described in the previous section.

Mono Coefficient Data Packet

The data packet structure used to update filter coefficients for a DSP to be operated in MONO mode is as follows:

Port ID	_START	<i>MONO</i>	<i>nTaps</i>	<i>SF</i>	C_0	C_1	...	$C_{nTaps-1}$	_STOP
---------	--------	-------------	--------------	-----------	-------	-------	-----	---------------	-------

MONO A defined constant used to indicate a single channel filter.

nTaps Specifies the total number of filter coefficients.

SF Specifies the scale factor applied to the filtered waveform. Because all packets are held in 16-bit integer buffers the SF value must be coded in 1.15 format. The realizable scale factors range from 1.0 to -0.9997559. The corresponding SF value coded into the packet is calculated as follows:

$$SF = -32768.0 \cdot \text{scalefactor}$$

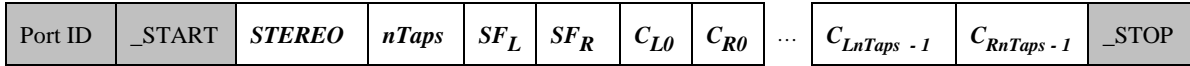
For example, to scale the signal down by 6dB (scale factor = 0.5), the scale factor is calculated as follows:

$$SF = -32768 * 0.5 = -16384$$

C_n Specifies the value of filter coefficient n .

Stereo Coefficient Data Packet

The data structure used when sending filter coefficients to a DSP operating in STEREO mode is similar to that used for one channel. Filter coefficients for left and right channels are interleaved as illustrated below.



STEREO Indicates a two-channel filter. This value can also be MONSTER, telling the DSP to operate in mono-in/stereo-out mode.

nTaps Specifies the total number of filter coefficients *per channel*.

SF_L Specifies the scale factor applied to the left channel. See SF discussion under Mono Coefficient Data Packet.

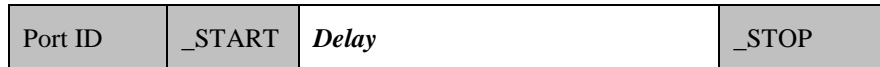
SF_R Specifies the scale factor applied to the right channel. See SF discussion under Mono Coefficient Data Packet.

C_{Ln} Specifies the value of filter coefficient *n*, left channel.

C_{Rn} Specifies the value of filter coefficient *n*, right channel.

Delay Time Data Packet

The data structure required to send delay time information using an inbound data stream is illustrated below:



Delay Delay is specified in terms of samples. Legal delay values range from 0 to 32760.

Latching Data

When a resource receives a data packet, it does not begin using the new data until it is *latched*. The PD1 supports two types of latching: global and local. As the name implies, a global latch will cause all "unlocked" resources to latch the last received data packet. This allows you to load more than one resource with new coefficient and/or delay time information and then latch them synchronously. Local latches, on the other hand, latch data to a specific resource only.

When the PD1 is IDLE, pre-loaded information can be latched using the XBDRV calls, **PD1latchDEL** and **PD1syncall**. Under most circumstances, however, resources will be latched in RUN mode by placing GSYNC and LSYNC markers on IB data streams.

Local Latching

A local latch is typically sent immediately following a data packet. This is not a requirement, however. To invoke a local latch, the following data series should be placed on an IB stream:

Typically some data packet	Port ID	LSYNC	_STOP
----------------------------	---------	-------	-------

Global Latching

It is often desirable to latch all resources synchronously. This is termed a global latch. A global latch may be accomplished by specifying the constant GSYNC at any point within an IB stream. When using multiple IB streams to carry data packets, it is important that only one GSYNC be issued per data packet sent. If GSYNC is specified within each inbound data stream, multiple latching will occur.

Organizing Data Packets on IB Streams

Data packet arrangement and IB stream organization can be programmed in a variety of ways. The PD1 can handle up to eight IB streams. The number used for a particular application depends on the amount of packeted information that needs to be sent and the update rate (frame rate) needed.

Typically, an application is designed so that all needed packets are sent to the required PD1 resources. A global trigger is then issued to actuate all resource changes simultaneously. The minimum time required to move all required data from the AP2 to the PD1 can be calculated as follows:

$$Time_{load} = \frac{NPTS_{total} \cdot sample_period}{N_{IB_Streams}}$$

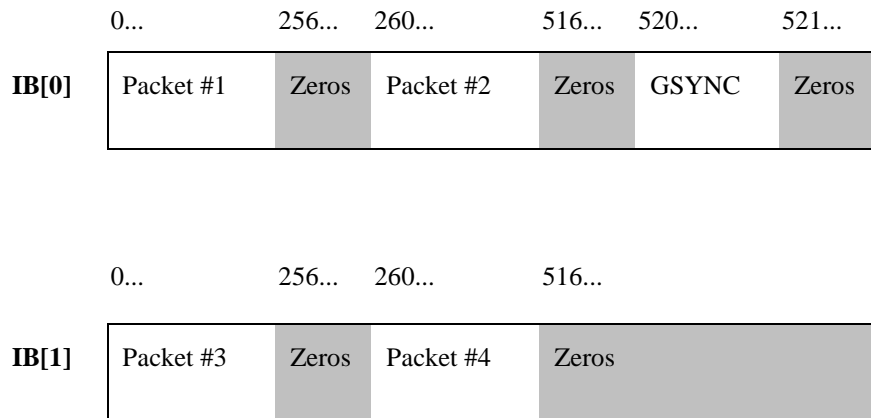
For example, if we needed to send four 250-tap filters using two inbound streams with a 50KHz (20 μ s) sample rate, the calculations would appear as follows:

$$NPTS_{total} = 4 \cdot (250 + 10) = 1040$$

$$Time_{Load} = \frac{1040 \cdot 0.02ms}{2} = 10.4ms$$

$$Update_Rate = \frac{1000}{10.4ms} = 96.15Hz$$

Note: The length needed for each filter packet is calculated as $250 + 10 = 260$. The extra ten points provide room for the needed packet "glue" and final GSYNC. A diagram showing packet arrangement is shown below:



Note: Only IB[0] contains the GSYNC marker. Including multiple GSYNCs will result in improper device operation.

The following APOS code could be used to build the buffers shown above:

```

dpush(540);          /* get 540 point buffer          */
value(0);           /* clear all points to zero          */

/* Build packet glue for filter #1                    */
make(0, COEF[0]);  /* Destination for packet #1        */
make(1, _START);  /* Start marker                      */
make(2, MONO);    /* Running DSP in MONO mode         */
make(3, 250);     /* Expect 250 taps                  */
make(4, -32768);  /* Set scale factor to 1.0          */
/* Later filter coefs will be overlaid at index 5     */
make(255, _STOP); /* Stop data flow                   */

/* Build packet glue for filter #2                    */
make(260, COEF[1]); /* Destination for packet #2        */
make(261, _START); /* Start marker                      */
make(262, MONO);  /* Running DSP in MONO mode         */
make(263, 250);  /* Expect 250 taps                  */
make(264, -32768); /* Set scale factor to 1.0          */
/* Later filter coefs will be overlaid at index 265   */
make(515, _STOP); /* Stop data flow                   */

make(520, GSYNC); /* GSYNC marker on IB[0] only      */

qpop16(DAMA1);

/* Identical code used to generate second DAMA,      */
/* however, the GSYNC specification is omitted.      */
/* Assume this filter is popped to DAMA2.            */
*/

```

The packet arrangement illustrated in this example offers just one possible solution. In reality, the options are nearly infinite. For example, all of the filter packets could have been sent on a single IB, or each one could have been sent on its own stream. The only limitation is that you do not exceed the transfer rate of the optical fiber. When running at 50KHz, this is not possible, even using eight IBs. However, if the sample rate is increased to 100KHz, no more than four IB streams can be used. Refer to the AP2/APOS documentation for specifications on the limits associated with the XBUS high speed data link (optical fiber).

The zeros left between filter packets in the above arrangement are inert; they simply allow filter packets to begin on the calculated boundaries. If update rate becomes critical, these zeros could be eliminated.

Controlling DAMA Data Flow from the AP2

In the above example, the "glue" needed for each data packet is pre-programmed into the two DAMA buffers being used, DAMA1 and DAMA2. Note that filter coefficients have not yet been loaded. Instead these will be overlaid once the PD1 is RUNning using the **qpoppart16** APOS call. This will allow the coefficients to be updated without respecifying the surrounding packet "glue".

For example, suppose the filters should be updated each time the user presses a key. The following code might be used:

```
printf("Press [SPACE] bar or [x] to exit");

do
{
  c = getch();
  if(c==' ')
  {
    /* build or load coefs for filter #1 on stack */
    qpoppart16(DAMA1, 5);

    /* build or load coefs for filter #2 on stack */
    qpoppart16(DAMA1, 265);

    /* build or load coefs for filter #3 on stack */
    qpoppart16(DAMA2, 5);

    /* build or load coefs for filter #4 on stack */
    qpoppart16(DAMA2, 265);
  }
}while(c!='x');
```

Using Play-Pause

Although the code shown above will work, it does not prevent data packets from being sent to the PD1 while they are also being updated. For example, the **poppart16(DAMA1, 5)** call could be loading filter data from the AP2 stack to DAMA1 while the interrupt driven software on the AP2 is shipping filter data from DAMA1 to the PD1 via IB[0]. This will result in the PD1 resource receiving a mix of old and new filter data. Of course, the next time the DAMA buffer loops around, the entire new filter will be loaded and latched. At worst, the irregular filter will be used for about 10ms. Often this brief irregularity is not an issue. In cases where it is, however, the irregularity can be eliminated using the new pause feature supported by APOS **seqplay**.

Refer to the [APOS Software Reference](#) for more information on using play-pausing.

Using play-pause, you can program the AP2 to send one or more DAMA buffers only when instructed. While a channel is paused, the AP2 will send an inert zero instead of sequential values from the DAMA buffer. Because the sending of packets and the corresponding updating of filters can be synchronized, it is possible to guarantee that only complete filters are received by PD1 DSPs.

A modified version of play-pause code is given below:

```
dplay(-DAMA1, -DAMA2);
/* Note, dplay also supports play-pause */
PDlarm(1);
PDlgo(1);

printf("Press [SPACE] bar or [x] to exit");

do
{
  c = getch();
  if(c==' ')
  {
    /* build or load coefs for filter #1 on stack */
    qpoppart16(DAMA1, 5);

    /* build or load coefs for filter #2 on stack */
    qpoppart16(DAMA1, 265);

    /* build or load coefs for filter #3 on stack */
    qpoppart16(DAMA2, 5);

    /* build or load coefs for filter #4 on stack */
    qpoppart16(DAMA2, 265);

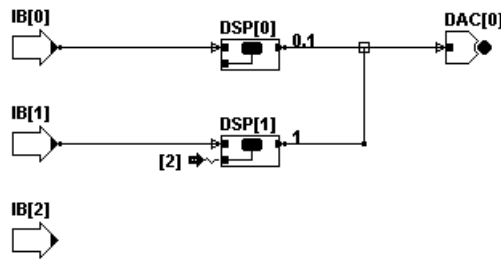
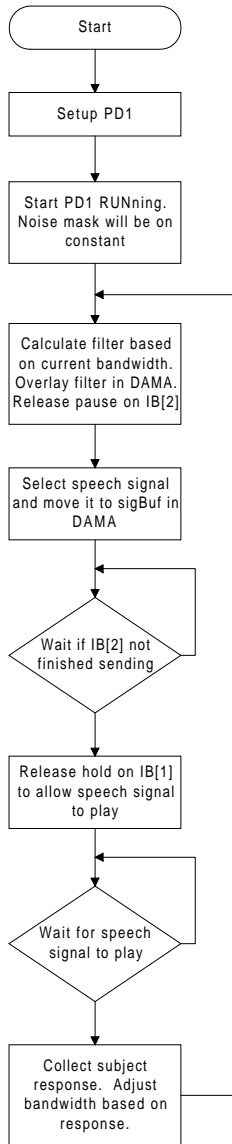
    /* Release pauses and allow packets to be sent */
    pfireall();

    /* Wait here until is paused again */
    do{}while(!ppausestat(1));
  }
}while(c!='x');
```

Example 3.0: Signal and Data Packet Control

To further clarify issues associated with packet arrangement and signal control, the following example program is provided. In this program, the PD1 will be programmed to conduct a speech intelligibility test. During the test, one of four words ("snap", "clap", "flap", "trap") will be presented and the listener will be asked to identify them. The speech signals will be bandpass filtered and presented with a continuous noise masker. The bandpass filter's width will be adjusted based on the subjects response. If a correct response is given the bandwidth will be narrowed; if an incorrect response is given it will be widened. At all bandwidths, the overall level of the signals will be held constant.

The following PD1 circuit will be used to realize this application:



Three inbound streams will be used. IB[0] and IB[1] will carry the noise masker and speech signals, respectively. DSP[0] will be pre-loaded with a bandpass filter that will band limit the noise masker between 100 and 5000Hz. Because the protocol requires a constant noise masker, the PD1 must be RUN continuously, meaning DSP[1] will be programmed dynamically. IB[2] will be used to send filter coefficient packets to DSP[1] which will perform the filtering on the speech signal.

This example demonstrates a fairly complex timing protocol. IB[0] will play continuously. IB[1] and IB[2] will be paused by default. IB[2] will be released to send a new filter to DSP[1] and IB[1] will be released to play the speech signal. A flow chart for the program is shown to the left.

Remember that AP2 data "channels" are aligned in the play process with PD1 IB streams as follows: Chan[1] ==> IB[0], Chan[2] ==> IB[1], and Chan[3] ==> IB[2]. When the two data streams are paused and release in the program, remember that **pfireone(2)** will actually release the signal corresponding to IB[1].

The following paragraphs will highlight portions of the program. A full listing of the program can be found in the file EXAM30.C distributed on the PD1 examples diskette.

Step 1 Clearing the PD1

```
PD1clear(1);
```

Attempting to use PD1 calls and PD1 reference variables before calling **PD1clear** is one of the most common programming errors.

Note: In this example, **PD1clear** is called *before* initialization of the DAMA buffer used to hold the filter coefficient packet. This is because the initialization uses PD1 reference variables (such as COEF[1]) and must follow the call to PD1clear.

Step 2 Specifying Conversion Parameters

```
PD1nstrms(1, 3, 0);
```

```
PD1srate(1, 20.0);
```

```
PD1npts(1, -1);
```

Because the coefficient packet being sent by IB[2] and the speech signal carried on IB[1] are never playing at the same time, one might think this application could be implemented using two IBs. IB[0] would carry the noise signal and IB[1] would carry both the filter packet and speech signal. This will not work because raw signal data *cannot* be sent in a data packet. IBs carrying raw signal data *must* be routed statically using **PD1specIB**.

Step 3 Defining the Routing Schedule

```
PD1clrsched(1);
```

```
PD1specIB(1,  
  IB[0], DSPin[0]);
```

```
PD1specIB(1,  
  IB[1], DSPin[1]);
```

```
src[0] = DSPout[0];  
sf[0] = 0.1;  
src[1] = DSPout[1];  
sf[1] = 1.0;  
PD1addmult(1,  
  src, sf, 2, DAC[0]);
```

The routing schedule for this application is very simple, consisting of two IB specifications and a single multi-route. The PD1 circuit used is shown above.

Note: The noise masker is scaled down by 20dB in the multi-route mix. Without this scale factor the speech signal can not be heard in the noise.

Step 4 Initializing the Delay Processor

The DP2 is not used in this application.

Note: When an application does not use the DP2, do not attempt any DELay processor calls. Doing so will result in an error message.

Step 5 Setting up the DSPs

```
MakeFilt( NTAPS,  
          N_LFREQ, N_HFREQ);  
  
PreLoadRaw(1, DSPid[0],  
           MONO, STACK, "", "",  
           1.0, 1.0, 1);  
  
PD1resetDSP(1, 0x2);
```

In this application, we will pre-load DSP[0] with a fixed bandpass filter. DSP[1] will be programmed dynamically using IB[2].

The program makes use of a simple coefficient generation procedure called **MakeFilt()**. This procedure generates the filter coefficients necessary to realize an arbitrary frequency response using a technique known as *frequency sampling*. In this application, the frequency sampling is used to generate simple bandpass filters.

This same call is used later to generate the filter coefficients needed to filter the speech signal.

After DSP[0] is pre-loaded, DSP[1] is reset in preparation for programming via an IB.

Note: A bit mask is used in this call to indicate which DSP(s) are to be affected by the call. Refer to the XBDRV reference for more information on bit masks used to select DSPs for programming.

Step 6 Running the PD1

```
seqplay(pspecBuf);  
PD1arm(1);  
PD1go(1);
```

Sequence play is used to send three channels (streams) of data to the PD1.

Note: Earlier in the program, the DAMA buffers used to carry the speech signal and filter coefficients were both specified in their corresponding sequence lists with a negative sign. This will activate play-pause on these buffers.

Step 7 Run-time Control of the PD1

Refer to code listing

Once PD1go is issued, the noise masker on Channel-1 (IB[0]) will play continuously, being filtered by DSP[0]. Channels 2 and 3 (IB[1] and IB[2]) will immediately go into pause mode.

Comparing the flow-chart shown above and the source code in EXAM30.c, we see that after some simple frequency calculations are done and a filter is calculated, Channel 3 (IB[2]) is loaded and released to allow the newly designed filter to load to DSP[1]. Next, a speech signal is selected and moved to the sigBuf DAMA buffer for playing. Once Channel-3 has finished sending, the speech waveform on Channel 2 (IB[1]) is released for playout.

After waiting for Channel 2 (IB[1]) to complete playout, the program prompts the user for a response and then adjusts the filter bandwidth accordingly.

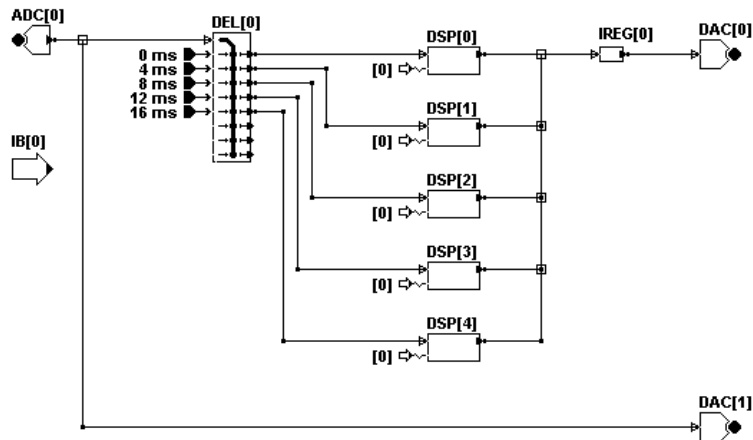
Putting it All Together

A complete listing of this program is available on disk (c:\tdt\drivers\pd1sup\docexams\exam30).

Example 3.1: Using Multiple DSPs to Implement a Long Filter

Each DSP on the PD1 is limited in the length of the filter that can be loaded (for statically loaded filters the limit is 480 taps; for dynamically loaded filters the limit is 255 taps). However, using the delay processor and multiple DSPs, one can implement a long filter. Using all 28 DSPs on a fully loaded system gives you the ability to create a 13,440 tap filter.

This example shows you how to implement a filter using multiple DSPs. The filter is blocked on the stack into 200 tap blocks, and each block is loaded into a different DSP. The signal to be filtered is sent to the delay processor where it is split and sent to multiple DSPs. The signal is delayed by the offset of its coefficient set. For example, the signal is delayed 4 ms ($20 \mu\text{s} \times 200$ taps) going into the second DSP, and 8 ms ($20 \mu\text{s} \times 400$ taps) going into the third DSP. The outputs of the DSPs are combined in a multiroute and then sent to a DAC.



AutoRoute was used to generate the routing of the signal (LongFilt.ART). One inbound data packet (IB[0]) is used to carry the filter coefficients. A typical hearing curve (THRTRACK.RES) is used as a filter in this program. The hearing curve is generated and manipulated in the frequency domain. The filter is converted to the time domain through an inverse FFT, and 1000 taps are blocked off for loading to the DSPs. The user can modify the filter while the program is running.

The filtered signal is played out DAC[0]. The unfiltered signal is played out DAC[1].

Pertinent sections of the program are duplicated below for an extended explanation of how these filters are loaded and the signal routed. You will need to look at the source code in its entirety to see how the program works. This example can be found on disk as source code and a compiled executable (c:\tdt\drivers\pd1sup\docexams\exam32\).

Step 1 Load Long Filter File from disk

```
pushdiska("thrtrack.res");
ntaps = topsize();
thrtrack=_allotf(ntaps);
curtrack=_allotf(ntaps);
coefstream=_allot16(250*5);
```

This typical hearing curve filter is in the frequency domain. DAMA Buffers are defined as follows:
thrtrack is the default filter
curtrack is the modified filter
coefstream will hold data stream being fed

Step 2 Build Filter Glue

```
for(i=0; i<5; i++){
    dpush(250);
    value(0.0);
    make(0, COEF[i]);
    make(1, _START);
    make(2, MONO);
    make(3, 200);
    make(4, -32768.0);
    make(210, _STOP);}
make(220, GSYNC);
catn(5);
qpop16(coefstream);
```

The filter will be run in five segments so we will prebuild 5 loading segments in the DAMA buffer (coefstream). The five packets are built separately and then concatenated into one filter packet.

Step 3 Define Routing Schedule

```
RouteSched(1);
DelaySet(1);
```

This calls the routing schedule that was setup in AutoRoute.

Step 4 Calculate and Load Long Filter

```
{
qpushf(curtrack);
scale(0.05);
alogten();
dpush(ntaps);
fill(0.0, 3.14159);
rect();
rift();
```

This section converts the frequency domain filter into a time domain filter. The block command is used to block off 1000 taps for the filter.

```
block(ntaps - 500, ntaps +  
500 - 1);  
reduce();  
hann();  
scale(32767.0/ntaps);  
for(i=0; i<5; i++)  
{  
block(i*200, (i+1)*200-1);  
extract();  
qpoppart16(coefstream,  
i*250 + 5);  
}
```

The signal is then scaled and blocked and popped into the inbound data packet.

Chapter 4 Designing Routing Schedules

The examples presented in this section assume that the user is creating routing schedules through TDT's circuit design application, AutoRoute. In most cases, the actual 'C' code generated by AutoRoute is also presented. This code can be compiled and tested using most 'C' compilers.

Using the Real-Time Router

See *The AutoRoute User's Guide* for more information about designing routing schedules.



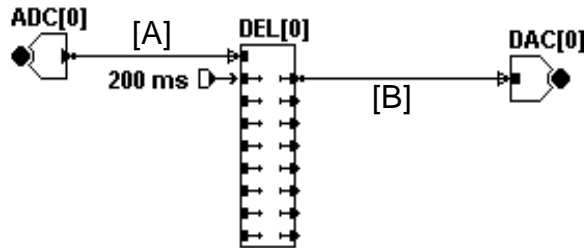
PD1 routing schedules can be generated by hand using five XBDRV calls: **PD1clrsched**, **PD1addsimp**, **PD1addmult**, **PD1specIB** and **PD1specOB**. Using these five calls, one can effectively "wire up" any conceivable PD1 circuit. For simple circuits, specifying the routing schedule can be very straightforward. As circuits become more complex, however, generating the routing schedule by hand can become tedious. To aid in this operation, a Windows compatible program called AutoRoute is provided with the PD1. Using this program, you can easily prototype PD1 circuits graphically and generate the source code necessary to create the desired routing schedule.

The discussion included here makes extensive use of the AutoRoute program. Although not required, use of this program is highly recommended.

What is a route?

All signals "handled" by the PD1 are in digital format. Sixteen-bit numbers are transferred between PD1 resources on logical wires called *routes*. The collection of routes needed to realize a complete circuit on the PD1 is called a *routing schedule*.

Typically, PD1 circuits are shown graphically using the iconic diagrams generated by the AutoRoute program. For example, a simple delay circuit can be implemented using the following:



In this circuit, analog waveforms connected to ADC[0] are converted to digital waveforms, delayed 200 ms by DEL[0], and converted back to an analog signal by DAC[0]. This circuit contains two routes, shown above as [A] and [B]. They are both *simple routes*, having a single source and destination, and can be specified using the following PD1 XBDRV calls:

```
PD1clrsched(1)
[A] PD1addsimp(1, ADC[0], DELin[0]);
[B] PD1addsimp(1, DELout[0][0], DAC[0]);
```

Note: The **PD1clrsched** call must be made just before routing calls are issued. Do not make any other PD1 calls between the **PD1clrsched** call and PD1 routing calls.

Route Types

The PD1 supports three route types:

- Simple routes
- Multi-routes
- Inbound and outbound stream routes.

Simple Routes Simple routes can be used for any connection from a single resource output port to a resource input port. On each "tick" of the sample clock, a simple route will cause the RTR to read a 16-bit word from the source port and write it to the destination port. IB and OB streams cannot be used with simple routes. Simple routes are added to the routing schedule using the **PD1addsimp** XBDRV call.

Multi-Routes Multiple resource output ports can be combined and sent to a resource input port using multi-routes. Using a multi-route, up to 15 resource outputs can be weighted (scaled) and summed. The resulting value is then sent to a specified resource port. On each conversion cycle, the RTR will read and scale all source ports, add them together, and write the result to the specified resource input port. IB and OB streams cannot be used in multi-routes. Multi-routes are added to the routing schedule using the **PD1addmult** XBDRV call.

Routing IB and OB Streams IB and OB streams can only be routed using special XBDRV calls. These calls specify a simple route between an IB or OB stream and a PD1 resource port. To route an IB stream, use **PD1specIB**. To route an OB stream, **PD1specOB** should be used.

Note: An IB stream cannot be connected directly to an OB stream.

Routing Rules

When specifying routing schedules using the Real-Time Router (RTR), a few rules must be addressed:

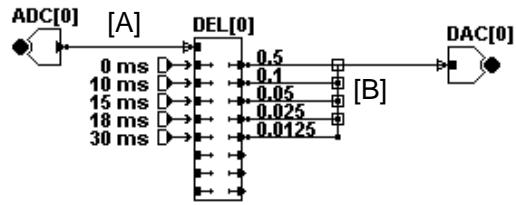
- The RTR processes the three route types in a fixed order. This is known as the Basic Ordering Rule.
- PD1 DSPs should always be read before they are written. This is known as the Read-Before-Write rule. Violation of this rule is known as a *write-before-read* error.

Basic Ordering Rule

The PD1 processes routes based on their type in the following order:

1. Multi-routes
2. Simple routes
3. Inbound and Outbound data streams

This Basic Ordering Rule is implemented by the *hardware*. Routes are always processed in this order, regardless of how they appear in the software code.

Example: Routing Schedule with multiple types**Routing Schedule:**

The AutoRoute program will generate the following 'C' code corresponding to the circuit shown above.

```

/*****
/* Call this procedure to auto route the PD1.
/*****
int RouteSched(int din)
{
    int src[32];
    float sf[32];

    PD1clr sched(din);

    src[ 0] = DELout[0][0];    sf[ 0] = 0.5;
    src[ 1] = DELout[1][0];    sf[ 1] = 0.1;
    src[ 2] = DELout[2][0];    sf[ 2] = 0.05;
    src[ 3] = DELout[3][0];    sf[ 3] = 0.025;
    src[ 4] = DELout[4][0];    sf[ 4] = 0.0125;
    PD1addmult(din, src, sf, 5, DAC[0]);

    PD1addsimp(din, ADC[0], DELin[0]);
    return(1);
}

```

Multi-Route [A]

Simple Route [B]

Applying the Basic Ordering Rule we see that the multi-route will be processed before the simple route.

Read-Before-Write Rule

In addition to the Basic Ordering Rule, there is a Read-Before-Write rule that must be adhered to when routing to and from DSPs on the PD1. You must read from a DSP *before* you write to it. Note that this rule only applies to DSPs. Other PD1 resources can be written to and/or read from in any order. In some cases, application of the Basic Ordering Rule *forces* a write-before-read. In these cases, it should be possible to use the isolation register to avoid the write-before-read problem.

Avoiding a Write-Before-Read Problem

The Real-Time Router allows the specification of an infinite variety of routing schedules. Write-before-read problems could arise in many of these schedules. Each routing schedule should be examined to see if a write-before-read problem exists. The examples on the following pages illustrate write-before-read problems and how to avoid them.

When examining a routing schedule to determine whether or not write-before-read problems exists for any given DSP, consider the following questions:

1. For any given DSP, will the application of the Basic Ordering Rule force a write-before-read?

If yes,

- Use an isolation register to avoid a write-before-read problem.

2. Will a write-before-read result if routes are coded in the order of signal flow?

If yes,

- Reverse the routing order.

Example 1: Simple Signal Processing

Signal Flow: The digital signal will be sent from ADC[0] to DSP[0]. Output from DSP[0] will be sent to DAC[0].

Problem: Linking the routes in the order of signal flow (left to right) will result in write-before-read on DSP[0].

Solution: Reverse the routing order (right to left).



Defining routes in the order of signal flow results in the following. Note the write-before-read on DSP[0].

Clears all routes

```
PD1clrshed(din);
```

*Reads from ADC[0],
Writes to DSP[0]*

```
1) PD1addsimp(din, ADC[0], DSPin[0]);
```

*Reads from DSP[0],
Writes to DAC[0]*

```
2) PD1addsimp(din, DSPout[0], DAC[0]);
```

Reversing the routing order solves the write-before read problem on DSP[0].

```
PD1clrshed(din);
```

*Reads from DSP[0],
Writes to DAC[0]*

```
1) PD1addsimp(din, DSPout[0], DAC[0]);
```

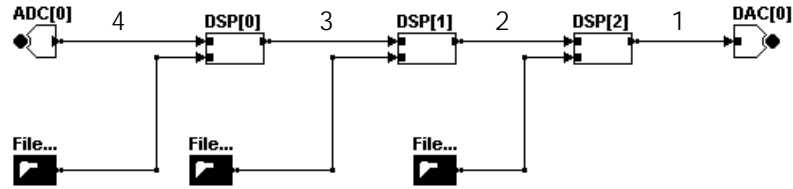
*Reads from ADC[0],
Writes to DSP[0]*

```
2) PD1addsimp(din, ADC[0], DSPin[0]);
```

Note: When defining routes graphically through use of the AutoRoute Windows Application, many write-before-read errors are corrected automatically by the software. However, such an automatic correction would not occur in the above example. Always draw routes in an order that avoids write-before-read errors. In the above example, the route between DSP[0] and DAC[0] would be drawn before the route between ADC[0] and DSP[0].

Example 2: Multiple DSPs Routed in Series

Signal Flow:	The digital output from ADC[0] will be sent through a series of 3 DSPs, DSP[0], DSP[1], and DSP[2]. The resulting output will be sent to DAC[0].
Problem:	Linking the routes in the order of signal flow (left to right) will result in write-before-read errors on DSP[0], DSP[1], and DSP[2].
Solution:	Reverse the routing order (right to left).



The code below specifies routes in the order of signal flow. Note the write-before-read errors on DSP[0], DSP[1], and DSP[2].

*Reads from ADC[0],
Writes to DSP[0]*

*Reads from DSP[0],
Writes to DSP[1]*

*Reads from DSP[1],
Writes to DSP[2]*

*Reads from DSP[2],
Writes to DAC[0]*

```
PD1clr sched(din);
```

- 1) PD1addsimp(din, ADC[0], DSPin[0]);
- 2) PD1addsimp(din, DSPout[0], DSPin[1]);
- 3) PD1addsimp(din, DSPout[1], DSPin[2]);
- 4) PD1addsimp(din, DSPout[0], DAC[0]);

By specifying all routes in reverse order, write-before-read errors may be avoided.

```
PD1clr sched(din);
```

*Reads from DSP[1],
Writes to DSP[2]*

*Reads from DSP[0],
Writes to DSP[1]*

- 1) PD1addsimp(din, DSPout[2], DAC[0]);
- 2) PD1addsimp(din, DSPout[1], DSPin[2]);
- 3) PD1addsimp(din, DSPout[0], DSPin[1]);
- 4) PD1addsimp(din, ADC[0], DSPin[0]);

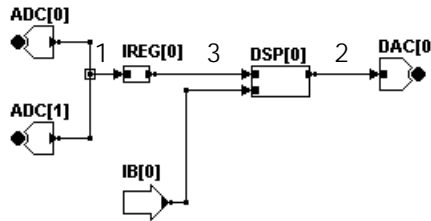
Example 3: Multi-route In, Single Route Out

This example is one in which the Basic Ordering Rule causes a write-before-read error.

Signal Flow: The digital data from ADC[0] and ADC[1] are to be mixed via a multi-route and sent to the input of DSP[0]. The output of DSP[0] will be sent to DAC[0] via a simple route.

Problem: The Basic Ordering Rule will cause a write-before-read error on DSP[0].

Solution: Include an intermediary isolation register, IREG[0]. The digital data from ADC[0] and ADC[1] are mixed via a multi-route and sent to the input of IREG[0]. The output of IREG[0] is then sent to the input port of DSP[0]. Use a reverse routing order (right to left) for all simple routes.



According to the Basic Ordering Rule, routes will be processed by the hardware in the following order, regardless of how they are specified in the software code:

1. Multi-routes
2. Simple routes
3. Inbound data streams

Without the addition of an isolation register, the Basic Ordering Rule creates a write-before-read error on DSP[0]. The routes would be processed as follows:

1. Multi-route from ADC[0] and ADC[1] to DSP[0]
2. Simple route from DSP[0] to DAC[0]

This write-before-read error can be avoided by including an intermediary isolation register.

1. Multi-route from ADC[0] and ADC[1] to IREG[0]
2. Simple route from DSP[0] to DAC[0]
3. Simple route from IREG[0] to DSP[0]

*Reads from ADC[0] and ADC[1],
Writes to DSP[0]*

*Reads from DSP[0],
Writes to DAC[0]*

*Reads from ADC[0] and ADC[1],
Writes to IREG[0]*

*Reads from DSP[0],
Writes to DAC[0]*

*Reads from IREG[0],
Writes to DSP[0]*

IB and OB Data Stream Routing

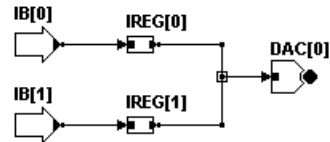
In some cases, it may be desirable to mix multiple data streams. However, inbound and outbound data streams may not be mixed via a multi-route. In most of these cases, the data may be mixed by adding intermediary isolation registers.

Example: Mixing Data Streams

Signal Flow: Two Inbound data streams, IB[0] and IB[1] are to be mixed and sent to DAC[0].

Problem: Inbound data streams may not be connected via a multi-route.

Solution: Add intermediary isolation registers.



Above is an example illustrating how signal data from two separate data streams may be mixed and output to a DAC through the use of intermediary isolation registers.

The data from IB[0] is sent unprocessed through IREG[0], while the data from IB[1] is sent unprocessed through IREG[1]. The output from IREG[0] and IREG[1] is mixed and sent to DAC[0].

```
PD1clrsched(din);
src[ 0] = IREG[0];   sf[ 0] = 1;
src[ 1] = IREG[1];   sf[ 1] = 1;
PD1laddmult(din, src, sf, 2, DAC[0]);
PD1specIB(din, IB[0], IREG[0]);
PD1specIB(din, IB[1], IREG[1]);
```

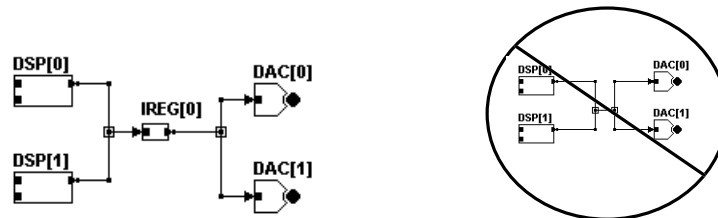
Solving Routing Problems with Isolation Registers

The isolation register is a temporary holding register. Isolation registers have many important uses. Several of these have been illustrated in the previous two sections. Additionally, two important uses of isolation registers are as follows:

- Multi-routes having multiple destinations
- Control of signal flow

Multi-Route with Multiple Destinations

You may wish to combine multiple signals with a multi-route and then send the summed signal as input to multiple resources. While it is possible to design such a route, it is inefficient. The multi-route must be processed for each destination resource. A better approach is to send the output of the multi-route to an isolation register. The output of the isolation register is then sent to multiple resources. By using this technique, the multi-route is processed only once.

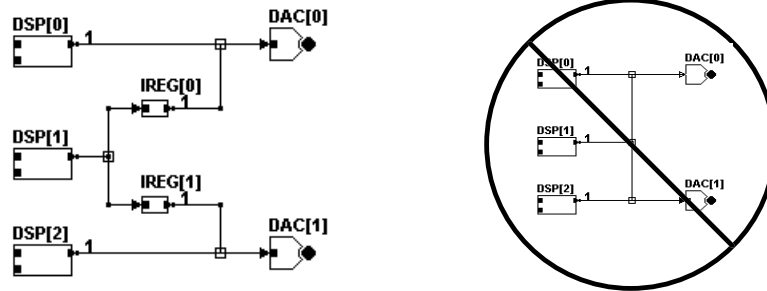


Controlling Signal Flow

Complex signal processing models often result in ambiguous signal paths. In such cases, isolation registers may be used to clarify and define signal flow in much the same way that a diode is used in circuit design.

Example 1: Controlling Signal Flow

Signal Flow:	The digital data from DSP[0] and DSP[1] are to be mixed via a multi-route and sent to the input of DAC[0]. The digital data from DSP[1] and DSP[2] are to be mixed via a multi-route and sent to the input of DAC[1].
Problem:	Without the use of isolation registers, data from DSP[0], DSP[1], and DSP[2] will be summed via two multi-routes, one sending data to DAC[0] and the other sending data to DAC[1].
Solution:	Add intermediary isolation registers.



Above is an example illustrating how isolation registers may be used to control the direction of signal flow.

Data from DSP[1] is sent through IREG[0] to be combined with the data from DSP[0] and sent to DAC[0]. Additionally, data from DSP[1] is sent through IREG[1] to be combined with the data from DSP[2] and sent to DAC[1].

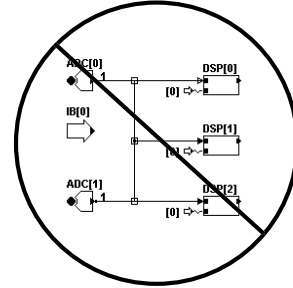
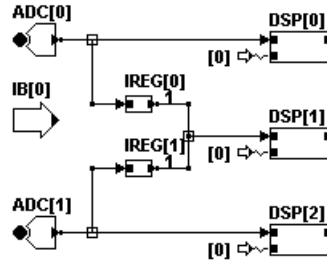
```

PD1clrsched(din);
src[ 0] = DSPout[0];   sf[ 0] = 1;
src[ 1] = IREG[0];    sf[ 1] = 1;
PD1laddmult(din, src, sf, 2, DAC[0]);
src[ 0] = DSPout[2];   sf[ 0] = 1;
src[ 1] = IREG[1];    sf[ 1] = 1;
PD1laddmult(din, src, sf, 2, DAC[1]);
PD1laddsimp(din, DSPout[1], IREG[1]);
PD1laddsimp(din, DSPout[1], IREG[0]);

```

Example 2: Controlling Signal Flow

Signal Flow:	DSP[0] is to receive data directly from ADC[0]. DSP[1] is to receive summed data from ADC[0] and ADC[1]. DSP[2] is to receive data directly from ADC[1].
Problem:	Without the use of isolation registers, all three DSPs will receive summed data from ADC[0] and ADC[1].
Solution:	Add intermediary isolation registers.



Data from ADC[0] is sent directly to DSP[0]. Data from ADC[1] is sent directly to DSP[2]. Additionally, data from ADC[0] is sent to IREG[0], while data from ADC[1] is sent to IREG[1]. The output of these two isolation registers is summed and sent to DSP[1].

```

PD1clrshed(din);
src[ 0] = IREG[1];   sf[ 0] = 1;
src[ 1] = IREG[0];   sf[ 1] = 1;
PD1addmult(din, src, sf, 2, DSPin[1]);
PD1addsimp(din, ADC[0], IREG[0]);
PD1addsimp(din, ADC[1], IREG[1]);
PD1addsimp(din, ADC[0], DSPin[0]);
PD1addsimp(din, ADC[1], DSPin[2]);

```

Chapter 5 The PD1 and 3D Audio

Before reading this chapter, you should be familiar with PD1 and its basic functionality. It is very important that you read preceding chapters of this document before proceeding. The discussions contained in this chapter also assume you are familiar with the methods and concepts associated with sound localization and the rendering of 3D auditory displays.

The PD1's high performance architecture is ideal for a variety of signal processing applications. In particular, its highly parallel architecture, along with its ability to update convolving filters dynamically, make it an ideal platform for the development of high performance 3D auditory displays, including virtual displays.

The PD1 localizes sound sources by convolving the sound source with a filter for the left and right ears. Beyond this basic construct, all other aspects, including filter size, filter type, and other associated model components, are controlled by the user.

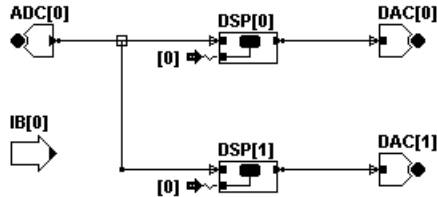
Head Related Transfer Function

For more information about HRTF Files, see the section "HRTF Files" below.

The head related transfer function (HRTF) determines the filter coefficients necessary to model 3-D audio. For any given source position, the HRTF will produce a specific set of filter coefficients. HRTFs may be specified by defining source position relative to the listener in terms of degrees of azimuth and elevation. HRTF information and associated filter coefficients are stored in files known as *HRTF files*.

Example 5.0: A 3-D Audio Application

In this example, a simple 3-D audio application is presented. The application models a 3D auditory display with a single sound source. ADC[0] functions as the signal input. Once digitized, the signal is routed to two DSPs where it is filtered based on the left and right ear transfer functions. The PD1 circuit being used is shown below:



While the PD1 will handle all signal routing, conversion, and real-time filtering, the program is responsible for determining the sound source position relative to the listener and for telling the AP2 to ship the correct HRTF sets to the PD1's convolvers.

In this example, sets of filter coefficients for various combinations of azimuth and elevation values will be loaded into AP2 memory from an HRTF file. The azimuth is incremented automatically during the running of the program (a useful exercise would be to modify the program to adjust azimuth and elevation based on user input). The azimuth and elevation values are used to select the correct HRTF filter set from AP2 memory. The filters will be streamed to the PD1 on IB[0].

This example can be found on disk as full source code and as a compiled executable (c:\tdt\drivers\pd1sup\docexams\exam50).

Refer to Appendix A for more information on PD1 supplemental calls.

A set of supplemental PD1 calls are provided to aid in loading and accessing HRTF filters from standard TDT HRTF files. Using these calls, the user can load HRTF files into AP2 memory using the **LoadHRTFFile** call and easily access them using the **PushHRTF** call.

Step 1 Clearing the PD1

```
PD1clear(1);
```

As in previous examples, it is necessary to execute **PD1clear** prior to executing any other PD1 function or referencing any PD1 variables.

```
/* from above      */
/* int hrtfdbn;    */
/* HRTFhead H;    */

hrtfdbn =
    LoadHRTFFile(&H, sss);
```

In this example, we must load the HRTF data from our HRTF file prior to performing Step 2, Specifying Conversion Parameters. This is necessary because we will be using information stored in the HRTF header to determine the sampling period specified in **PD1srate**.

The supplemental call, **LoadHRTFFile**, loads the HRTF file specified by *sss* from disk. It allocates the necessary DAMA buffer and returns the dama buffer number, *hrtfdbn*. It also loads the file header into the HRTF header, *H*. The default file used in the example on disk is *sos127_5.hrt*, but other HRTF files could be loaded by providing the file name.

Step 2 Specifying Conversion Parameters

```
PD1nstrms(1, 1, 0);
PD1srate(1, H.speriod);
PD1npts(1, -1);
```

As can be seen, we are using **PD1nstrms** to define only one IB data stream and no OB data streams. Examination of the call **PD1srate** shows that we will convert the data at a rate determined by the sampling period defined in the HRTF header, *H*. Based on the call **PD1npts(1, -1)**, we will convert the data continuously, until we call **PD1stop**.

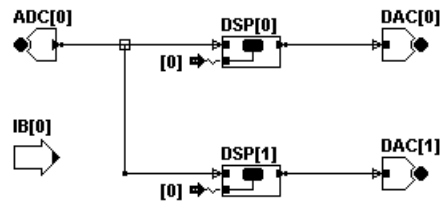
Step 3 Defining the Routing Schedule

For more information about designing routing schedules see Chapter 4 of this document and the [AutoRoute User's Guide](#).

The routing schedule used in this example is relatively simple, consisting of four simple routes.

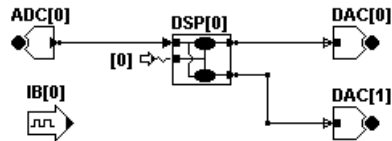
The PD1 circuit needed to implement our application is shown below.

```
PD1clrsched(din);
PD1addsimp(din,
  DSPout[0], DAC[0]);
PD1addsimp(din,
  DSPout[1], DAC[1]);
PD1addsimp(din, ADC[0],
  DSPin[0]);
PD1addsimp(din, ADC[0],
  DSPin[1]);
```



Note: In this example, we are processing left and right coefficients for a single source. To do so, we are using two different DSP's, each receiving data from separate MONO coefficient packets. Using this configuration, the maximum filter length we can use (assuming 50KHz sampling) is 255 taps per ear.

If HRTF filter lengths are restricted to 127 taps or less this example could be rendered using the following circuit. Note that only a single DSP is required.



Step 4 Initializing the Delay Processor

The DP2 is not used in this application.

Note: When an application does not use the DP2, do not attempt any DELAY processor calls. Doing so will result in an error message.

Step 5 Setting-up the DSPs (convolvers)

```
PD1resetDSP(1, 0x3);
PD1interpDSP(1, 500,
             0xf);
```

We will be dynamically updating left and right filter coefficients. The call **PD1reset** is issued to ensure that both DSPs are operating in convolution mode, enabling us to dynamically update coefficients. We have also optionally enabled filter coefficient interpolation by calling **PD1interpDSP**. Refer to the XBDRV reference manual for information on this call and its parameters.

Step 6 Setting-up to RUN

In this example, we will program DSP[0] and DSP[1] via IB[0]. In order to accomplish this, we must initialize a DAMA buffer and then build a record consisting of the packet for DSP[0] followed by the packet for DSP[1].

```
pspecbuf = _allot16(10);
pseqbuf = _allot16(10);
coef = _allot16(600);
```

```
dpush(10);
make(0, pseqbuf);
make(1, 0);
qpop16(pspecbuf);
```

```
dpush(10);
make(0, -coef);
make(1, 1);
make(2, 0);
qpop16(pseqbuf);
```

Sequenced Play

Sequenced play is used to allow use of the play-pause feature. The play specification list (DAMA buffer *pspecbuf*) includes only one DAMA buffer, *pseqbuf*. *Pseqbuf* includes the DAMA buffer that will hold the filter coefficients (*coef*).

Note that in the sequence buffer list, the DAMA buffer *coef* is preceded by a negative sign. This enables the pause feature, allowing us to ensure that filter coefficients will be updated synchronously.

Step 6 Setting-up to RUN (continued)

```

dpush(600);
value(0.0);

make(0,COEF[0]);
make(1,_START);
make(2,MONO);
make(3,H.ntaps);
make(4,-32768.0);
make(10+H.ntaps,_STOP);

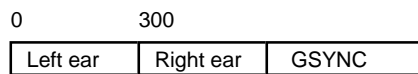
make(300,COEF[1]);
make(301,_START);
make(302,MONO);
make(303,H.ntaps);
make(304,-32768.0);
make(310+H.ntaps,_STOP);
make(312+H.ntaps,GSYNC);
qpop16(coef);
azstep = 1.0/H.resAZ;
elstep = 1.0/H.resEL;
seqplay(pspectbuf);
PDlarm(1);
PDlgo(1);

```

Building the Filter Packets

We will be updating our filter coefficients dynamically while the PD1 is RUNning. Prior to RUNning, we need to build a record that contains all the packet "glue." The actual filter coefficients will be inserted into the packets during RUNning through the use of **qpoppart16**.

The record shell containing the left and right ear packets is illustrated below.



Modifying the Packets

Later in Step 7, we will include coefficient data by modifying the 5th and 305th elements of the DAMA buffer *coef*.

Latching the Data

The element at position $312 + H.ntaps$ of the DAMA buffer is loaded with the GSYNC marker. This marker will cause the previously programmed delay times to be latched into the DP2. All dynamically programmable PD1 resources can be loaded asynchronously and then latched using a GSYNC or LSYNC marker.

Initializing Azimuth and Elevation Positions

In this application, the the azimuth value is incremented automatically. Azimuth and elevation resolution values are obtained from the HRTF header, *H*. Later, during RUNning, changes in azimuth and elevation values will be used to obtain filter coefficients from the HRTF buffer, *hrtfdbn*.

Step 7 Dynamically Controlling Filter Coefficients

```
do
{
  PushHRTF(&H, az1, ell,
  CT_LEFT, hrtfdbn);
  PushHRTF(&H, az1, ell,
  CT_RIGHT, hrtfdbn);

  do{
  }while(!ppausestat(1));

  az1 = az1 + 1.0;
  if (az1>H.maxAZ)
  az1 = H.minAZ;

  qpoppart16(coef, 305);
  qpoppart16(coef, 5);

  pfireall();

  //Code trapping any
  changes the user has
  made to sound source
  elevation and azimuth
  could be placed here.//
}while(!(kbhit()));
```

In this example, sound source position is incremented while RUNning. The new azimuth and elevation values are stored in the variables *az1* and *ell*, respectively.

Filter coefficients are updated based on the current azimuth and elevation values. The process is as follows:

1. Coefficients for the left and right channels are pushed onto the stack through the use of the function call, **PushHRTF**.
2. The application is halted until playing is paused.

This is accomplished through use of a do/while loop that examines the pause status by calling **ppausestat**. This pause loop ensures that we do not update the *coef*DAMA buffer while it is in the process of playing out coefficients.

3. *az1* is incremented by one on each loop. If the value of *az1* exceeds the maximum azimuth in the HRTF file (H.maxAZ), it is set to the minimum azimuth (H.minAZ).
4. Once play is paused, the *coef*DAMA buffer is updated through the use of **qpoppart16**.
5. Play is re-initiated by calling **pfireall**.
6. The process is repeated until the user quits the application.

Step 8 Stopping the PD1

```
PD1stop(1);
```

The PD1 will be stopped when the user presses any key via the PD1stop call.

Putting it All Together

A complete listing of this program is available on disk as full source code, and as a compiled executable (c:\tdt\drivers\pd1sup\docexams\exam50).

HRTF Files

When using the PD1 in acoustic localization and virtual reality work, filter files called HRTF files are used to store the Head Related Transfer Functions (HRTFs). The HRTF header specifies the number, location, and other general information about each filter. Actual filter coefficients are stored as HRTF records. Each record consists of either a mono filter or a pair of stereo filters recorded for a particular spatial position.

HRTF files are binary integer files. These files are used by TDT's acoustic modeling application, Sound Stage. To ensure ease of use, HRTF files should be saved with the default extension, *.hrt*.

HRTF File Formats

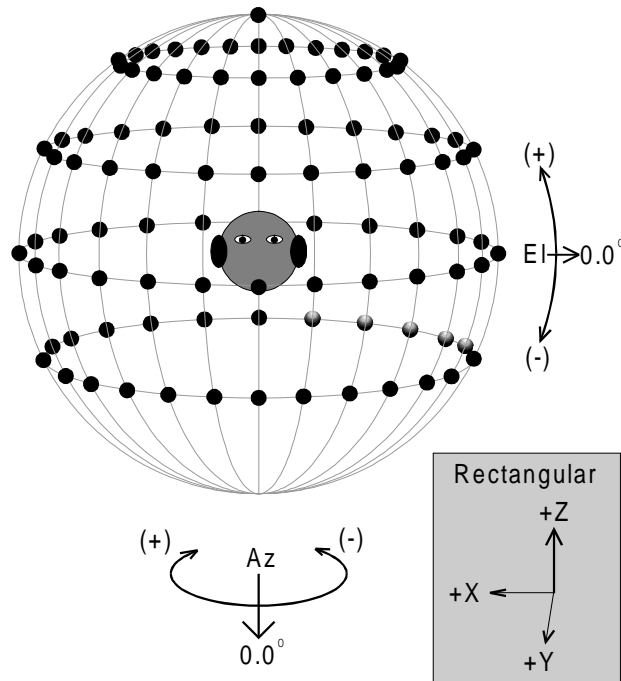
When programming the PD1, you can store your HRTF filters in any format you like. However, in an effort to simplify and standardize some of this record keeping, TDT has specified a versatile and expandable file format.

The TDT HRTF file format consists of a file header followed by a number of HRTF filters. The first 16-bit word within the file header specifies the HRTF format code, allowing for future upgrading and changing of the file format.

Type 1 File Format

Type 1 file format supports HRTF filters at uniform azimuth and elevation locations over all or part of a sphere. Using a Type 1 file format, HRTF filters are represented geometrically as shown below.

HRTF filters are located at the intersection of defined elevations and azimuths. The defined elevations range from -30 to +90 degrees with a 30 degree elevation resolution. Defined azimuths range from -180 to +180 degrees with a 15 degree azimuth resolution. Although this is not a typical HRTF arrangement it will serve well as an example.



Header Structure

The TDT HRTF header structure is shown below:

```
typedef struct
{
    int formcode;          /* Format code          */
    int ID;                /* User ID             */
    int npos;              /* Total Positions     */
    int recszie;           /* Size of each record */
    int ntaps;             /* Number of taps per ear */
    float minAZ;          /* Minimum AZ angle    */
    float maxAZ;          /* Max AZ angle        */
    float resAZ;          /* One over AZ angle size */
    int nAZ;              /* Number of AZs      */

    float minEL;          /* Minimum EL angle    */
    float maxEL;          /* Max EL angle        */
    float resEL;          /* One over EL angle size */
    int nEL;              /* Number of ELs      */

    float speriod;        /* Sample Per (usec)   */

    int dummy[1003];      /* For later use       */

}HRTFhead;
```

Some of the values are redundant for the current Type-1 file format. However, these redundant values will be needed for future planned arrangements.

HRTF Header Fields

The HRTF header fields are explained below. An appropriate value is also provided, based on the example shown above:

formcode = 1

Indicates file format code. To date, the only specified file type is Type 1. Type 1 file format is indicated by setting *formcode* = 1.

ID = any value

This integer is set aside for user use.

npos = 97

Use this value to indicate the total number of HRTF filter positions recorded in the HRTF file. This value is calculated as follows:

$$\text{npos} = (\text{Number of azimuths} * \text{Number of elevations}) + 1.$$

The term +1 is necessary so that the HRTF filter positioned at 90 degrees elevation is included in the total. In cases where there will be no filter at 90 degrees elevation, it is still necessary to include a "dummy" filter at 90 degrees. In our example, $\text{npos} = (24 * 4) + 1 = 97$.

Note: A dummy filter at 90 degrees elevation can be excluded from filter coefficient calculation by specifying a maximum elevation less than 90 degrees. See *maxEl* below.

recsze = 254

An HRTF record consists of a stereo pair of HRTF filters recorded for a particular location. In Type 1 file format, this record simply consists of the left ear filter coefficients followed by the right ear filter coefficients. The *recsze* variable can be calculated as follows:

$$\text{recsze} = 2 * \text{ntaps}.$$

In this example, there are 127 filter taps per ear. Therefore, $\text{recsze} = 2 * 127 = 254$.

ntaps = 127

Number of filter taps per ear.

minAZ = -165

This value indicates the minimum azimuth at which filters exist in the file. In our example, filters are located at azimuths distributed throughout the entire sphere, therefore the *minAZ* is $-(180 - 15) = -165$.

maxAZ = 180

This value indicates the maximum azimuth at which filters exist in the file. In our example, filters are located at azimuths distributed throughout the entire sphere, therefore the *maxAZ* is 180.

resAZ = 0.0666666

The variable *resAZ* specifies the inverse of the azimuth resolution, and therefore should be set to one over the azimuth angle between adjacent HRTFs. In our example, the azimuth resolution is 15 degrees. The value of *resAZ* is calculated as follows: $\text{resAZ} = 1.0/15 = 0.0666666$.

nAZ = 24

Number of azimuths at each elevation.

minEL = -30

This value indicates the minimum elevation at which filters exist in the file. In our example, the minimum elevation is -30 degrees.

maxEL = 90

This value indicates the maximum elevation at which filters exist. In this example, the maximum elevation is 90 degrees.

Note: In cases where the maximum elevation will be less than 90 degrees, a dummy filter at 90 degrees elevation must be specified. (See *npos* and *nEL*.) This dummy filter may be excluded from filter coefficient calculation by setting *maxEL* equal to the actual maximum elevation.

resEL = 0.0333333

The variable *resEL* specifies the inverse of the elevation resolution, and therefore should be set to one over the elevation angle between adjacent HRTFs. In our example, this value is calculated as $1.0/30 = 0.0333333$.

nEL = 5

This variable specifies the number of elevations at each azimuth.

$nEL = \text{Number of elevations at each azimuth} + 1$.

The term +1 is necessary so that the HRTF filter positioned at 90 degrees elevation is included in the total. In cases where there will be no filter at 90 degrees elevation, it is still necessary to include a "dummy" filter at 90 degrees. In our example, $nEL = 4 + 1 = 5$.

Note: A dummy filter at 90 degrees elevation can be excluded from filter coefficient calculation by specifying a maximum elevation less than 90 degrees. See *maxEl* above.

speriod = 20.0

HRTF sampling period in μ seconds.

Specifying HRTF Filters

HRTF filters are specified as the intersection of defined elevations and azimuths. These filters are defined in order of descending elevation. Within each elevation, filters are specified in order of descending azimuth.

Note: In cases where there will be no filter coefficients generated at 90 degrees elevation, it is still necessary to specify a "dummy" filter at this elevation.

In the previous example, the maximum elevation is 90 degrees, the elevation resolution is 30 degrees, and the minimum elevation is -30 degrees. The azimuth positions range from -165 degrees to +180 degrees with a resolution of 15 degrees. The HRTF Filter file would appear as follows:

Header
EL = 90
EL = 60, AZ = 180
EL = 60, AZ = 165
EL = 60, AZ = 150
.
.
.
EL = 60, AZ = -165
EL = 30, AZ = 180
EL = 30, AZ = 165
.
.
.
EL = 30, AZ = -165
EL = 0, AZ = 180
EL = 0, AZ = 165
.
.
.
EL = 0, AZ = -165
EL = -30, AZ = 180
EL = -30, AZ = 165
.
.
.
EL = -30, AZ = -165

Appendix A PD1 Supplemental Calls

Function calls that simplify the process of loading filter coefficients to PD1 DSPs are provided with the PD1. These functions provide a means for loading raw coefficients and coefficients obtained from HRTF files.

The calls described in this section are distributed in a 'C' source file. The source and header files containing these function calls are named *pd1_sup.c* and *pd1_sup.h*, respectively. Pascal source code can be found in the file PD1_SUP.PAS.

Constants

Parameter values for various functions in this section are specified through the use of several sets of constants.

Single or dual channel operation mode is indicated through the use of the following constants:

MONO	Single channel
STEREO	Dual channel
MONSTER	Single channel in/dual channel out

Raw filter coefficient source location is defined as follows:

FILE_16	16-bit integer file
FILE_F	Floating point file
FILE_A	ASCII file (floating point values)
DAMA_16	16-bit AP2 DAMA buffer
DAMA_F	Floating point AP2 DAMA buffer
STACK	Top buffer on the AP2 stack

HRTF coefficient record type is specified by the following:

CT_LEFT	Left channel
CT_RIGHT	Right channel
CT_STEREO	Stereo
CT_MONSTER	Single channel in/dual channel out

Function Calls

PreLoadRaw(*din*, . . .)

Prototype	int PreLoadRaw(int <i>din</i> , int <i>dspn</i> , int <i>opmode</i> , int <i>stype</i> , char * <i>src_lm</i> , char * <i>src_r</i> , float <i>sf_lm</i> , float <i>sf_r</i> , int <i>lock</i>)
Description	This call pre-loads the specified DSP with single or dual channel filter coefficients. These coefficients may be obtained from raw filter files or from the AP2 stack or DAMA buffers.
Arguments	<p><i>din</i> PD1 Device index number.</p> <p><i>dspn</i> DSP ID number (0..27). Use DSP[n].</p> <p><i>opmode</i> Operation mode. Use MONO, STEREO, MONSTER.</p> <p><i>stype</i> Source type. Use FILE_16, FILE_F, FILE_A, DAMA_16, DAMA_F, STACK.</p> <p><i>src_lm</i> Pointer to left channel or single channel data. Points to a character string when the source is a file. Points to an integer when the source is a DAMA buffer. Not used when the source is the stack.</p> <p><i>src_r</i> Pointer to right channel data. (See <i>scr_lm</i> above.)</p> <p><i>sf_lm</i> Left channel or single channel scale factor. Scale factor must not be less than -0.99976 and must not exceed 1.</p> <p><i>sf_r</i> Right channel scale factor. Scale factor must not be less than -0.99976 and must not exceed 1.</p> <p><i>lock</i> Non-zero value locks the DSP with the loaded filter coefficients.</p>
Example	<pre>PreLoadRaw(1, DSPid[0], STEREO, FILE_16, "left.fir", "right.fir", 1, 1, 1);</pre> <p>This call will pre-load DSPid[0] with dual channel filter coefficients obtained from two separate 16-bit integer files, <i>left.fir</i> and <i>right.fir</i>. A scale factor of 1 is applied to both channels. Coefficients are locked in.</p>

Appendix B Running a Separate A/D Module

With a special factory modification to the PD1, it is possible to run both a PD1 and a separate A/D module within one XBUS system. It requires (1) the A/D module be installed in its own XB1 device caddie, and (2) that both caddies have their own OI1 for data communication. Optical fibers from the AP2 card are split between the two OI1s. Show below is a diagram displaying the proper connection of optical fibers to the two OI1 interfaces. Pay close attention to connector color coding and be sure to insert optical connectors all the way into their receptacles.

