PDEV Spectrometer User Guide

Jeff Mock
2030 Gough St.
San Francisco, CA

415 346-2829
jeff@mock.com

Summary

This is a technical user guide for the PDEV spectrometer. The first portion of the document is a user guide for operation of the spectrometer. The second section is technical information for extending operation of the spectrometer.

Terms

Here are a few terms used throughout the document, it's useful to know a few of these terms upfront.

PDEV	This refers to the entire spectrometer, the hardware and the software on both the spectrometer and the fileservers used to control it. The name doesn't mean anything. The name was chosen at the beginning of "development" and lives on as the name of the spectrometer. Sometimes it's uppercase, sometimes lowercase.
GX	GX refers to the signal processor FPGA in the spectrometer. There are two FPGAs in the spectrometer, usually referred to as GXA and GXB. GX is the entire FPGA, the internal design of the FPGA is broken into separate pieces with different names.
SP	The FPGA design is divided into two pieces. SP is the signal processing datapath in the FPGA. PDEV is designed so that SPs for other purposes can be dropped into GX and take advantage of the rest of the system without too much effort on the part of the developer. SP sometimes refers to the default signal processor used for EALFA/PAFLA processing, but I try to use SP1 to when referring to the default SP design.
SP1	The default signal processor. This is the signal processor design for the FPGA used for EALFA/PALFA processing. It does a dual polarity PFB with a programmable transform length from 16-8192 points. It does full stokes calculations and has an efficient method for dumping only the data required for the observation. In rev.2 it adds a complex mixer, a digital LO, and a decimating low pass filter for high resolution frequency bins. Rev.3 adds time domain dumping from the decimating low pass filter and more control for generating a calibration output signal synchronized to the polyphase filter.
SPEX	Simple example signal processor. This is a simple SP design, it doesn't do anything useful, but serves as a simple example for someone that wants to write a new SP.
PFB	Polyphase filter bank. A PFB is an improvement over the FFT that converts a signal to the frequency domain with much great channel to channel separation than is possible with an FFT.

FIR	Finite impulse response filter. In this application usually a low pass filter
DLPF	Decimating low pass filter. A low pass filter that reduces that bandwidth limits the input signal and reduces the output datarate of the filter by an integral multiple.
HR mode	In rev.2 of the FPGA, a decimating low pass filter can be inserted into the signal path to generate frequency bins as narrow as 12Hz for high spectral resolution processing. When the decimating LPF is enabled this is referred to as HR mode.
plinth	Plinth is everything in the FPGA besides the SP. The plinth code provides a register interface to the powerPC processor in the spectrometer, a DMA interface for dumping data, a large QDR SRAM FIFO for buffering realtime data from the SP to the non-realtime world. Plinth contains diagnostic code for testing the path from the ADCs, DMA, and FIFOs, measuring signal levels, and synchronizing observation starts across many PDEV boxes to PPS.
pnet	pnet is the program used to control a network of spectrometers. pnet is written in perl and is run from a server machine. It reads configuration files for details about the observation and communicates with many spectrometer boxes and the file servers used to store data. It coordinates a synchronized observation across many machines and checks the integrity of the network of machines to make sure that the spectrometers are running off the same ADC clock, running the same version of the software (and many other things).
prun	prun is the application programming running on the spectrometer CPU that manages the observation. Prun is started when pnet connects to a spectrometer. It writes the FPGA registers, starts the observation, sends data to the fileserver (psrv), and reports observation status back to pnet. One copy of prun runs for each FPGA on the spectrometer box, so there are typically two copies of prun running on each spectrometer box during an observation.
psrv	Psrv is the program running on the fileserver during an observation. Psrv is started by prun on a spectrometer. It receives data from prun on the spectrometers and stores data on the local filesystem in a special PDEV file format.
pmon	Pmon runs on the spectrometer and monitors the hardware status. It keeps track of temperature, voltage, and fan speeds. Pmon will power down the FPGA and keep pnet from running if something is out of spec. Pmon controls the tiny LCD display on the front of the pdev boxes.

The Spectrometer Box

The spectrometer box contains a powerPC processor running Linux. It is an 800MHz CPU, so the box probably has about 25% of the processing power of a typical 2006 PC. The processor manages data movement from two large FPGAs to the gigabit ethernet ports. Most of the signal processing in the box is done on the two large FPGAs.

Each FPGA is connected to an analog board with four 12-bit ADCs. The bottom row of SMA connectors on the front of he box connect to one ADC board, the top row of SMA connectors connect to the other ADC board. The two rows of SMA connectors are normally associated with two sub-bands of signal processing. A row of four SMA connectors is normally associated with two polarities of two complex signals.

There is an SMA connector on the back of the box dedicated to the sample clock for the ADCs. The clock signal should be a 0.5v sine wave from 100-200MHz. The box has an internal low jitter 156.25MHz oscillator that can be used as the ADC clock. In this case the sample clock is not referenced to an external source.

There are four SMA connectors labeled GP0 – GP3 for digital I/O on the back of the box. These are for PPS, cal, blanking, and a synchronized cal output. Each of the four connections are identical and software configuration lets you assign functions to the connectors. Each of these connectors can be a digital input or output controlled by the FPGA.

There are two independent gigabit ethernet ports labeled A and B. The B port is currently unused but it is simple to enable this port if it is needed. The MAC addresses of the ethernet ports are stored with the serial number of the box in a tiny EEPROM on the CPU board. The MAC address is related to the serial number as follows. If the serial number of the box is #0123, the MAC addresses are:

```
Port A:      aa:bb:cc:01:23:00
Port B:      aa:bb:cc:01:23:01
```

The MAC addresses are shown in the LCD display shortly after reset.

There are two DB-9 PC-style serial ports on the back of the box. These are 3-wire serial ports (Tx/Rx/Gnd), they do not have any RS-232 flow control signals. The first serial port is used by the boot ROM to print diagnostic messages. These serial ports are currently unused after booting but will likely be used to control mixers is the commissioned system.

The front of the box has a 128x64 LCD that is primarily used for displaying system status. In normal operation pmon uses the display to show fan speed and temperatures. The DNS name of the spectrometer box is displayed in the lower left corner of the display and if an observation is taking place the name of the observation is in the upper right corner.

The switch on the front panel is a reset switch. There is a mains power switch on the back of the box near the power plug.

Booting the Spectrometer

The spectrometer box can be booted a few different ways. It can boot linux from on-board flash, but this is an advanced topic. The primary method for booting the spectrometer is using DHCP and TFTP protocols over the A ethernet port.

The spectrometer box has an 8MB flash chip that is partitioned into three areas. 256kB of the flash contains a boot ROM. 768kB of the flash is set aside for a small file system to contain local system parameters (like a static IP address if needed), the remaining 7MB of flash is reserved for a system image file if the box needs to boot standalone. For this discussion we'll only consider the 256kB of boot ROM and network booting. The boot ROM is the first code executed by the powerPC after power-up or reset. This code uses the first serial port as a console, but this is rarely needed. For diagnostic purposes, you can connect a serial cable to a PC at 115 kbaud and see all of the boot messages, interrupt the boot process, and enter commands into the console.

Normally, the spectrometer will check to see if there is a valid boot image in its local flash. If this boot image is not valid, the spectrometer will use DHCP to get an IP address and the IP address of a TFTP server and the filename to boot. All of this information must be provided by a DHCP server. A typical DHCP server entry for the spectrometer might look like:

```
host pdev-101 {
        hardware ethernet   aa:bb:cc:01:01:00;
        fixed-address       pdev-101.mock.com;
        option host-name    "pdev-101.mock.com";
        filename            "pdev";
}
```

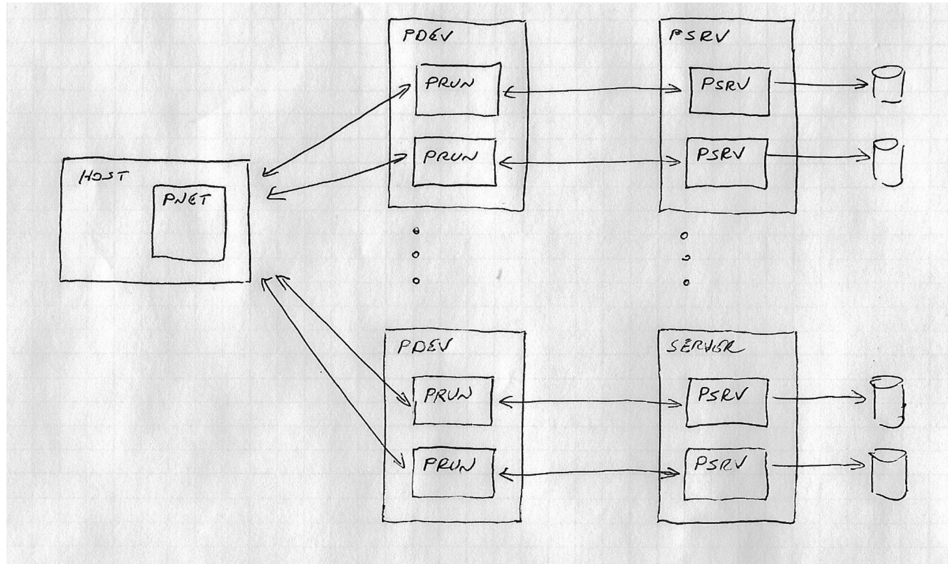
The spectrometer will next use TFTP to try and load the boot file from the TFTP server. The boot file is a compressed image of both the Linux kernel and the root filesystem. This file is typically about 4MB. Once the file is loaded and uncompressed, the Linux kernel is booted with a root filesystem in RAM and Linux takes over from this point.

The boot ROM puts a few progress messages on the LCD display, more messages are sent to the serial port. Once Linux boots, the LCD is controlled by a device driver in Linux kernel. After the Linux kernel is booted, startup scripts will configure the FPGAs, setup the network interface and start the hardware monitor application. Booting is complete when temperature and fan speed information are displayed in the LCD. It should take about 30-seconds to boot a large network of spectrometer boxes from the network.

Observation Overview

Each spectrometer box contains a powerPC processor running Linux and two FPGAs that each do signal processing on a 170MHz sub-band.

Figure 1 shows the relationship of pnet, prun, and psrv during an operation. pnet is typically run from the command line with an option that specifies the configuration file to be used for the observation.



Pnet makes a connection to the spectrometers and starts prun for each FPGA to be used in the observation. Typically there are two copies of prun running on each spectrometer for the two FPGAs. In a typical ALFA observation pnet will manage 14 network connections, 2 connections to each spectrometer box for each beam in ALFA.

Pnet is typically run from a server machine, but can probably be run from any Linux machine on the observatory local area network, the bandwidth requirements are low but it does need to

maintain several connections. During an observation, pnet will receive status information once per second from both prun and psrv (through prun). Depending on the debug level, pnet might print out a lot of this information.

Prun runs under Linux on the powerPC processor on the spectrometer box. It is started when pnet makes a network connection to the spectrometer. A copy of prun is started for each FPGA used during the observation. The start of the observation is synchronized to PPS, but the DMA and data storage for each FPGA take place independently.

Psrv runs on the fileserver and is started when prun makes a network connection to the fileserver. It is a simple program that stores the data dumped from the spectrometer in a PDEV specific file format. It reports status back to prun which in turn passes the status back to pnet to report to the user.

pnet.conf

Observations are primarily setup with the `pnet.conf` file. This file specifies the spectrometer boxes to use the for the observation and register settings for the observation. The default signal processor for EALFA/PALFA has a large number of registers and configuration options. The `pnet.conf` file can be reasonably complex. Typically, a `pnet.conf` file might be setup once for a given type of observation and used for many observations. Real example `pnet.conf` files can be found in `/usr/local/pdev/etc` on the machine where `pnet` is intended to run.

The file format is a free-format text file. Lines beginning with # are comments, blank lines are ignored. The file is broken into sections with each section containing specific parameters. Consider the following simple file:

```
# Sample pdev.conf file for testing
#

[pdev]
# name   IP address  Beam    Subband sp  setup   file server
#
beam0x   pdev-103    0       0       0   gxa     fs1
beam0y   pdev-103    0       1       1   gxb     fs1
beam1x   pdev-104    1       0       0   gxa     fs2
beam1y   pdev-104    1       1       1   gxb     fs2
beam2x   pdev-105    2       0       0   gxa     fs3
beam2y   pdev-105    2       1       1   gxb     fs3
beam3x   pdev-106    3       0       0   gxa     fs4
beam3y   pdev-106    3       1       1   gxb     fs4
```

The `[pdev]` section specifies the SPs to be used for the observation. This example gives names to the eight SPs on four spectrometer boxes. The first column is a symbolic name for the SP. The second column is the host name of the spectrometer box containing the SP. The next two columns group the SPs into beams and sub-bands. As far as pnet is concerned, these groupings are arbitrary and intended to be used in post processing to group signals together. In this example there are four beams (0-3) with two sub-bands (0-1) for each beam.

This is close to ALFA where there are seven beams and two sub-bands for each beam. This format would allow specifying something like a single beam with many sub-bands for a single-pixel wide-band receiver (provided that the mixers in the analog IF path can support such an arrangement.)

The fifth column is the physical FPGA number containing the SP. PDEV has two FPGAs in each box numbered 0 and 1 corresponding to GXA and GXB respectively. The sixth column is the

name of the setup section to be used for configuring the SP registers. In this case there will be two different setup sections, GXA for sub-band 0 and GXB for sub-band 1. The last column is the name of the fileserver that prun will connect for dumping.

```
# The signal processor ID and version number expected in the
# FPGAs
#
[sp 01.03]
```

The `[sp]` section specifies the identification for the SP expected for the observation. Each SP provides the Plinth code a 16-bit number. The upper 8-bits are a unique number identifying the SP design, the lower 8-bits are the version number of the SP design. In this case 01 is the ID for the default EALFA/PALFA SP1 signal processor, expecting version 03 of the this SP design. Before an observation is started, pnet verifies that all of the spectrometers have the correct FPGA code loaded and are running the same version of the operating software.

```
# These are registers defs for the signal processor.  A
# different sp will probably have different registers.
#
# The default signal processor has a lot of registers
#
# Full defaults for these registers in the sample files
# in /usr/local/pdev/etc
#
[defs]
ARSEL           0       # Input select
AISEL           1
BRSEL           2
BISEL           3
ARNEG           4       # negate input
AINEG           5
BRNEG           6
BINEG           7
LEN             12      # transform length 16-8192
DIAG            13
PSHIFT          14
PFBBY           15
SHIFT           17
FCNT            19
DCNT            20
DSHIFT_S0       22
DSHIFT_S1       23
DSHIFT_S2       24
DSHIFT_S3       25
```

The `[defs]` section defines the register names and addresses for the SP. Each SP design will have a different `[defs]` section naming the various registers. This example is small portion of the SP1 register set. The Plinth code addresses 64k 16-bit registers. Each line in the `[defs]` section contains the name of the register and the address in the Plinth address space. In this case, the register ARSEL is a register that selects the source for the real portion of the A-polarity input to the SP1 signal processor.

```
# Header for dump file
#
[header]
FMTWID          # 0=8-bit components, 1=16-bit, 2=32-bit
FMTTYPE         # 0=power, 1=A/B power, 2=full stokes
LEN             # transform length
```

```
DUMPSTART       # First bin # dumped
DUMPSTOP        # Last bin # dumped
FCNT            # Number of transforms per integration
DCNT            # Number of transforms dropped per int
ARSEL
AISEL
BRSEL
```

The [header] section specifies fields to be added to the header of the PDEV dump file. Each of these parameters is a 16-bit value corresponding to the value written to an SP register. The header of the PDEV file already contains a number of Plinth items like the ADC clock frequency, the beam/sub-band of the SP, etc. This sections is for including information specific to the SP, like the LEN register that contains the transform length of the SP1 signal processor. The idea is that all SP1 dump files will have the same header, but other SP designs will have header values specific to the SP.

```
[dump]
name            x1234
filesize        1000000     # in 1kbyte blocks
byteswap        3           # 0-7
magic           0x2e83fb01  # Magic number for SP1 files
dma             65536       # DMA size from FPGA
gpoe            0x8         # Enable output on GPIO pins
ppssel          0           # GP input for PPS, default is 0
ppsedge         1           # set to one to use posedge
ppsforce        1           # Force a fake PPS to start obs

    # In the default sp these are all 4-bit values that see
    # the threshold for console error reporting of
    # overflow/saturation detection at
    # various points in the datapath.  In general, these are
    # 8 32-bit values that can be used to mask error reporting
    # in any signal processor design.
    #
err0_thresh     0           # ovf_adc overflow
err1_thresh     0           # ovf_pfb overflow
err2_thresh     16          # ovf_vshift saturation
err3_thresh     16          # ovf_acc_s2s3 saturation
err4_thresh     16          # ovf_acc_s0s1 saturation
err5_thresh     16          # ovf_ashift_s2s3 saturation
err6_thresh     16          # ovf_ashift_s0s1 saturation
err7_thresh     0           # not used

    # These are stated valued that are passed through to the
    # pdev file.  pnet assumes they are correct.  The pdev
    # boxes also calculate an estimated ADC clock, but it is
    # only accurate to about 0.1%.  This value, if provided
    # is treated as the actual stated ADC clock frequency.
    $
lo1mix          1625.0
lo2mixlow       174.0
lo2mixhigh      310.0
adcclk          170.0
```

The `[dump]` section is used for several miscellaneous items related to starting an observation. The `name` item sets the primary name used for the dump files. This name also appears in the LCD of the spectrometer during observations. The dump filename is composed of the `name` parameter, the start date of the the observation, the symbolic name for the SP in the `[pdev]`

section, and a sequence number. Filenames for a test observation done at the time of this writing:

```
(a)   (b)      (c)    (d)   (e)
x1234.20070109.beam0x.00000.pdev
x1234.20070109.beam0x.00001.pdev
...
x1234.20070109.beam0y.00000.pdev
x1234.20070109.beam0y.00001.pdev
...
x1234.20070109.beam1x.00000.pdev
x1234.20070109.beam1x.00001.pdev
...
x1234.20070109.beam1y.00000.pdev
x1234.20070109.beam1y.00001.pdev
...
```

(a) Name from the [dump] section
(b) Date observation started in YYYYMMDD
(c) SP identifier from [pdev] section
(d) File sequence number
(e) All files have a .pdev extension

`Filesize` sets the maximum size for each sequence of the dump file in 1k byte blocks. Setting to zero puts the observation in a single file.

The plinth code that handles DMA can be programmed to do byteswapping in a flexible way. The `byteswap` parameter controls this function. The LSB of the parameter does single byte swapping. The next bit swaps 16-bit shorts, the next bit swaps 32-bit words. Depending on the type of components being dumped by SP1, the byteswapping parameter needs to be set to particular values to accommodate the differences between powerPC byte ordering (big endian) and the X86 byte ordering on the file servers (little endian).

`Magic` sets a 32-bit magic number in the pdev dump file. This is used to distinguish an SP1 dump file from some a dump file for some other type of SP.

`dma` sets the size of individual DMAs from the FPGA to the hardware. Change this value at your peril.

`gpoe` turns on the output drive for the GP SMA connectors on the back of the box. Normally the connectors are TTL level inputs, setting the corresponding bit to 1 make the connector an output. For this to be useful, the SP must produce some interesting signal to drive out of the box. In the case of SP1, a GP pin set as an output is used for a synchronized winking cal output signal.

`ppssel` and `ppsedge` set the GP input and the edge of the signal to use for PPS synchronization of the beginning of the observation. `ppsforce` is used to cause a fake PPS signal to start an unsynchronized observation if no PPS signal is present.

`err0_thresh` through `err7_thresh` are used for setting thresholds for masking error reporting to the console during an observation. The parameters can be used in a user defined way for any SP design. The assignments for the default SP are shown in the example above.

```
[setup gxa]

# First thing in SP setup puts FPGA in diagnostic mode
DIAG            1
```

```
# Input selector A and B complex signals
#   0..3  Selects ADC0 .. ADC3
#   4       Internal test signal selected for input
#   5       Sets component to zero
#
# A and B typically represent the A and B polarities of the
# input signal. R and I typically represent the real (I) and
# imaginary (Q) portions of the signal.
#
ARSEL           0
AISEL           1
BRSEL           2
BISEL           3

# This will load 32k 16-bit values for PFB filter coefficients
# The file pfb.4096.hamming is generated by the pnet_mkpfb_coeff
# script
#
PFB0            file pfb.4096.hamming
```

The `[setup]` section is the list of the register operations to setup an SP for an observation. The operations are performed in the order specified. The last value written to any register is used in the `[header]` section. If the `file` parameter is used, values are read from a file and written to consecutive addresses on the SP. This is useful for writing things like filter coefficients in an SP.

Registers are written in the order specified in the [setup] section. Registers can be overwritten or written multiple times. For register values that are written in the header of the PDEV dump file, the last value written to the register is saved in the header.

pnet.conf conventions

The configuration file supports a recursive include directive to break the configuration into multiple pieces. Pnet first looks for the include file in the current directory, if it's not found in the current directory pnet looks for the file in `/usr/local/pdev/etc`. By convention, a configuration file is broken up as follows:

```
# Sample pdev.conf file for testing
#
include "boxes.conf"
include "cal.conf"
include "sp1def.conf"

[dump]
name            x1234
filesize        1000000     # in 1-kbyte blocks
byteswap        3           # 0-7
magic           0x2e83fb01  # Magic number for SP1 files
dma             65536       # DMA size from FPGA
gpoe            0x8         # Enable output on GPIO pins
ppssel          0           # GP input for PPS, default is 0
ppsedge         1           # set to one to use posedge of pps
ppsforce        1           # Force a fake PPS to start obs

[setup gxa]

# First thing in SP setup puts FPGA in diagnostic mode
DIAG            1
```

```
# Input selector A and B complex signals
#   0..3  Selects ADC0 .. ADC3
#   4       Internal test signal selected for input
#   5       Sets component to zero
#
ARSEL           0
AISEL           1
BRSEL           2
BISEL           3
...
```

`boxes.conf` contains the `[pdev]` section, `boxes.conf` is kept in `/usr/local/pdev/etc` and is common to all configuration files for the spectrometer. This way, if there is a change to the physical system configuration because of faulty hardware, the global `boxes.conf` file is modified and all of the configuration files will follow the change.

`cal.conf` contains calibration information for the ADCs. This is typically setup as a system wide file to compensate for DC offsets and gain mismatches in the ADCs or analog electronics prior to the spectrometer.

`sp1def.conf` contains the `[sp]`, `[defs]`, and `[header]` sections. This information is all specific to the SP design and should be provided by the SP designer. So, a typical configuration for an observation only needs to provide the proper `[dump]` and `[setup]` sections to setup parameters for the observation.

[cal] section

Plinth conditions the ADC inputs before sending the signal to the SP. The ADCs have an intrinsic DC offset, this is usually less than 10 ADC units, but is sometimes sufficiently large to cause a large DC spike in long transforms. The Plinth code can correct this offset before the signal is sent to the SP. In rev.2 of the FPGA, the plinth code has a scaler after the offset to adjust for any difference in gain prior to the spectrometer. The gain stage can adjust the gain of each ADC by a factor of [0..2) as a 1.15 fixed point number. The scaling is specified as a floating point value in the [cal] section as follows:

```
[cal beam0x]
adc0_offset     1
adc1_offset     3
adc2_offset     5
adc3_offset     -9
adc0_scale      1.000
adc1_scale      0.987
adc2_scale      0.999
adc3_scale      1.234

[cal beam0y]
adc0_offset     9
adc1_offset     7
adc2_offset     -3
adc3_offset     -1
adc0_scale      1.000
adc1_scale      0.987
adc2_scale      0.999
adc3_scale      1.234
```

There is a [cal] section for each SP line in [pdev] since each ADC will have its own individual characteristic. It can be somewhat tedious to generate this information manually seeing as there

are 56-ADCs in an ALFA system, so pnet can automatically generate the calibration information. The following pnet command will calculate the DC offsets based on an average of 8M samples from the ADCs:

```
% pnet --sigcal=cal.conf
```

The following command will generate `cal.conf` for all of the SPs in the configuration file. The ADCs seem to be pretty stable, so it's probably not necessary to generate a new calibration file very often. Some care should be taken with the telescope while generating calibration data to insure that the noise baseline is suitable for calibration and not overly affected by a source or RFI.

Generating scaling information for the cal.conf file is a little trickier. `--sigcal` does not generate scaling information by default. It might be desirable to automatically generate scaling information to level gains across all of the beam but there might be other mitigating factors that make generating scaling values problematic. So, to automatically generate scaling values for the [cal] section use the following command:

```
% pnet --sigcal=cal.conf --scale
```

The above command will set the scaling of the ADC with the lowest RMS signal level to 1.000. Other ADCs will be scaled by a value less than or equal to 1.000 such that all beams have the same RMS signal level. This may not be what you want...

Pnet can be used to test signal levels. The following command will repeated report the mean and RMS signal levels (with the calibration registers applied). This might be useful for testing signal levels prior to an observation. The values are in ADC units and show the four ADCs associated with each SP:

```
% pnet --sigstat

# Mean signal level
#
# beam0x:    0.46    0.50   -0.41    0.27
# beam0y:   -0.23   -0.04   -0.22    0.07
# beam1x:    0.15   -0.05   -0.50   -0.20
# beam1y:   -0.23   -0.37   -0.03    0.05
#
# RMS signal level
#
# beam0x:    1.04    1.07    1.08    0.99
# beam0y:    0.98    0.93    0.99    0.95
# beam1x:    0.95    1.03    1.09    0.99
# beam1y:    0.99    1.02    1.00    0.98
...
```

Pnet is a little particular about the `[cal]` section. If one SP has a `[cal]` section then all of the SPs must have a `[cal]` section. This can create a small problem if new hardware is added to the system. In this case, temporarily create an empty `cal.conf` file in the current directory until a new file can be created with `pnet –sigcal=cal.conf`.

Pnet Operation

Pnet has a lot of options, it can run a variety of diagnostics on the spectrometers and has a number of utility functions for managing a group of spectrometers, but only the main operating mode is discuss here. Run pnet with the `-?` option to see all of the options or refer to the technical section later in the document for more information.

Once the configuration file is set, observations are relatively simple. The –dump=n option to pnet says to start a synchronized observation across all of the SPs and dump n blocks from each SP:

```
% pnet --dump=1000
Connected to all
Connected to all psrv
First dump sequence number is 00000

118.84 integrations per second
Estimated dump time 8.4 s
Estimated total dump size 0.26 GB

Spectrometer box bandwidth estimates:
    pdev-105   15.58 MB/s
    pdev-111   15.58 MB/s

Fileserver bandwidth and dump size estimates:
    ao         31.16 MB/s, 0.26 GB

All spectrometers running...
0.00 MB/s 0.00 MB [0:0]/1000 blocks  (0.0%)
30.57 MB/s 15.60 MB [101:101]/1000 blocks  (10.1%)
29.84 MB/s 45.61 MB [219:219]/1000 blocks  (21.9%)
31.04 MB/s 77.99 MB [338:338]/1000 blocks  (33.8%)
31.09 MB/s 109.09 MB [457:457]/1000 blocks  (45.7%)
31.08 MB/s 140.25 MB [576:576]/1000 blocks  (57.6%)
31.11 MB/s 171.44 MB [669:695]/1000 blocks  (66.9%)
31.12 MB/s 202.54 MB [814:814]/1000 blocks  (81.4%)
31.12 MB/s 233.70 MB [932:932]/1000 blocks  (93.2%)
All spectrometers finished
%
```

When pnet starts up it will calculate a bandwidth estimate for each spectrometer box and a bandwidth estimate to each fileserver. It is the observer's responsibility to insure that these bandwidths are reasonable. The connections are gigabit ethernet that has a wire speed of about 115MB/s. The design goal is to support 50MB/s from each spectrometer box and 50MB/s to each fileserver. In practice the system performs much better. Long test observations have sustained bandwidths of more than 75MB/s per link. If the bandwidth limit is exceeded, the failure will usually occur as a FIFO overflow, possibly in the FPGA hardware or in the powerPC device driver ring buffers. Either way, a dump that fails because the non-realtime network and filesystems cannot keep up with the realtime requirements of the SPs will result in an error message for the first failure and pnet exiting with an error status code. It is important to setup observation configurations that do not write to the fileservers too fast.

Pnet will try to create dump files starting with a sequence number of 00000. If this file already exists, Pnet will look for the first available sequence number that is a multiple of 100. Pnet will not normally overwrite existing files. If you would like to overwrite possibly existing dump files, run pnet with the `–force` option. In this case the dump files will always begin with sequence number 00000.

You can use the `–obs=foo` option to pnet to add a special field to the filename. In this case the dump files will have the following names:

```
x1234.20070109.foo.beam0x.00000.pdev
x1234.20070109.foo.beam0x.00001.pdev
...
```

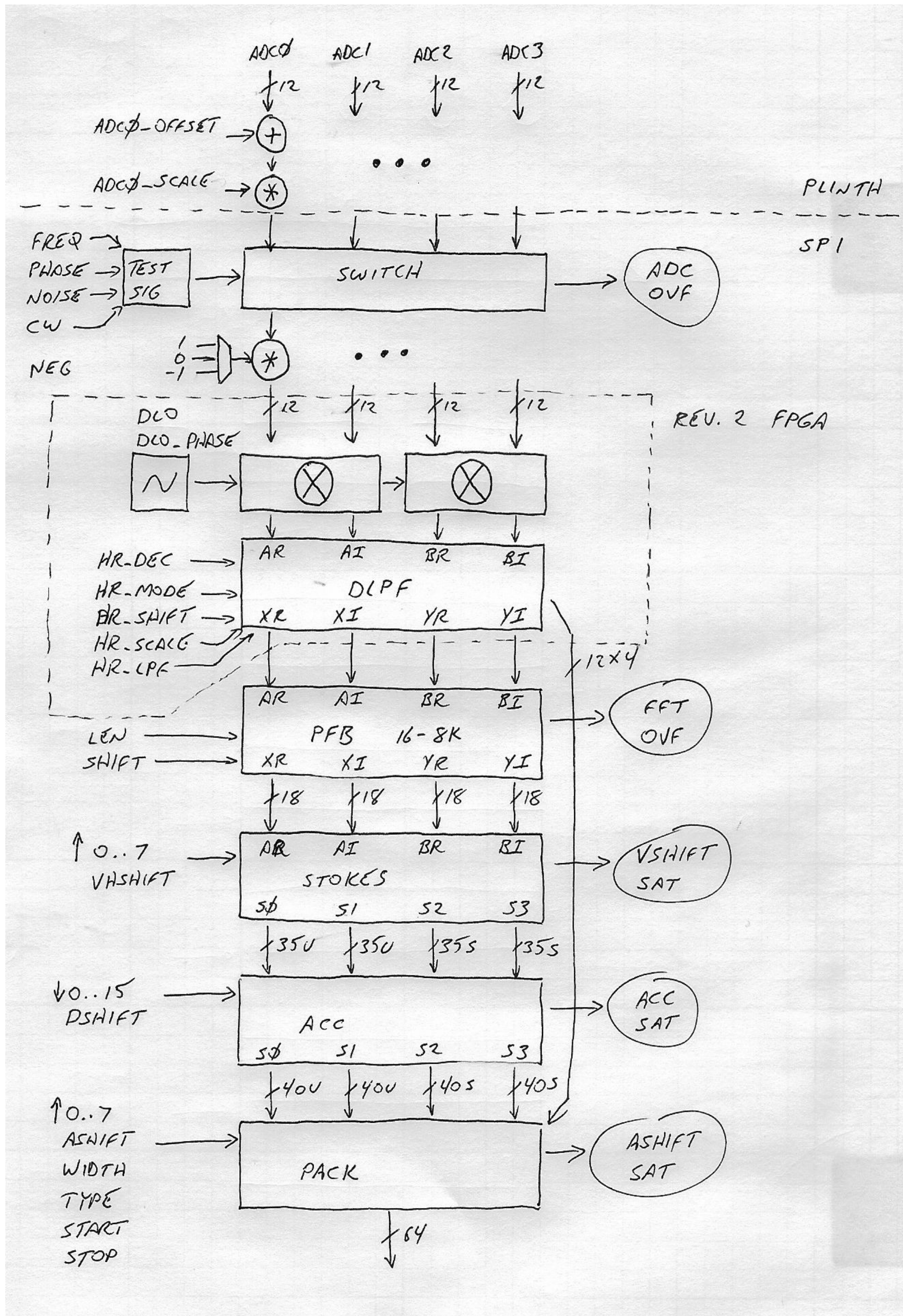
In case of trouble you can run pnet with the -d option to turn on debugging messages. Using the

-d option twice will turn on a lot of debugging messages.

A observation can be aborted with a ^C. This will result in files that have a dump header inconsistent with the length of the file, but otherwise self-consistent and usable. Because of the many FIFOs and buffers in the system, the files for different SPs might be slightly different lengths after an aborted observation.

SP1 Programming

SP1 is the default SP (signal processor) for pdev. This SP meets all of the requirements for PALFA and EALFA observations. It has a lot of registers, but hopefully programming the spectrometer is not too complex. The registers listed here are defined in `/usr/local/pdev/etc/sp1defs.conf` and should be included in any SP1 configuration file. Figure 2 show the datapath for SP1. This document is updated for rev.2 of the SP1 FPGA. Rev.2 includes a new complex mixer, a decimating low pass filter for high resolution frequency bins (HR mode), and a mechanism for time domain dumping of the output of the low pass filter.



Input, decimation, and PFB

Data from four ADCs passes through a crossbar switch and is assigned to form two complex

signals for the two polarities of a single beam. The ADC values can be independently negated or set to zero to adapt to different exponent conventions for complex signals in the analog mixers or to process real signals before the spectrometer.

The ADCs detect overflow if the analog signal is outside the limits of the converter. The ADC overflows are counted and recorded as status information with each integration packet.

There is also an internal test signal generator for diagnostics that generates a linear or elliptically polarized complex CW signal plus noise. This test signal is not used in normal operation, but is used for diagnostics and to verify simulation results against real hardware.

The next stage of processing (new to revision 2 of the FPGA) is a complex mixer. The LO for the complex mixer is an internally generated digital LO. There are two complex mixers, one for each polarization. Each mixer receives the same frequency from the digital LO, but the phase difference between the LO for the two mixers can be adjusted arbitrarily. The section on HR mode describes the mixer in more detail. If the DLO frequency and phase are set to zero, the mixer is a precise numerical pass-through of the input signal.

Following the complex mixers is an optional decimating low pass filter. The DLPF can be bypassed in normal operating modes. Enabling the DLPF turns on HR mode, a high resolution mode for calculating very narrow frequency bins across a portion of the input signal. The decimation of the low pass filter is variable to any integral value from 2x to 1024x. The filter is an 8x overlap filter. The filter is symmetrical around DC, so only 4 coefficient tables are needed, these are mirrored around DC in hardware for the 8x overlap. Filter coefficients are calculated with the script `pnet_mkdlpf_coeff` and are loaded as part of the `pnet.conf` file. With loadable filter coefficients the width of the filter, the windowing function for the filter, and other filter parameters are left as software options to the user preparing the configuration file for the observation. The output of the DLPF is sent to the PFB, but the output of the DLPF can be send to the packer for direct dumping of time domain information. The bandwidth of ADC input time domain signals is too great for time domain dumping, but if the DLPF decimated enough to reduce the bandwidth to a manageable level, this time domain information can be dump directly. This might be useful for alternative applications like RADAR signal processing. The DLPF is described in detail in the HR mode section.

The input to the PFB/FFT is four signed 12-bit signal components from either the DLPF or the complex mixers. The PFB FIR filter processes the 12-bit data using a 16-bit coefficient table accumulating an 18-bit result. The filter coefficients are chosen such that the peak gain of the FIR filter is less than one so there is no possibility of overflow in the FIR filter. The FIR filter can be bypassed under software control making the PFB block only perform an FFT if desired. The filter coefficients are loaded by the configuration process for the SP.

The FIR portion of the PFB is a 4x overlap filter that supports a maximum transform length of 8192. The filter length can be programmed by application software for a transform length of any power of two from 16 to 8192 points. The filter coefficients for the FIR filter can be read and written by application software making it easy to use different filter characteristics without changing the FPGA.

The FFT can also be programmed by application software for any power of two transform length from 16 to 8192 points. The FFT takes a shift mask that controls downshifting of the FFT on a stage by stage basis. If there is overflow in the FFT this is detected and recorded as status for the result packet. The FFT datapath is 18-bits throughout producing 18-bit results for the components.

Stokes calculations

Stokes parameters are calculated according to the following (as told to me by Aaron Parsons):

```
I = AA* + BB*
Q = AA* - BB*
U = AB* + BA*
V = j(AB* - BA*)
```

The hardware actually calculates and integrates four slightly different scalar values:

```
s0 = 2AA*           (unsigned value)
s1 = 2BB*           (unsigned value)
s2 = 2Re(BA*)
s3 = 2Im(BA*)
```

An exercise for the reader to check my work and verify that:

```
I = ½ (s0 + s1)
Q = ½ (s0 – s1)
U = s2
V = s3
```

Calculating s0,s1,s2,s3 have a couple of advantages. s0 and s1 are calculated as unsigned values (no sign bit). This provides one extra bit of precision for I and Q. For example, if dumping 8-bits per component, s0 and s1 are 0..255 (really [0,1)). Calculating I Q from s0 and s1 gives 9 real bits of precision for these parameters where you would only get 8-bits of precision if I and Q were dumped directly. Keeping s0 and s1 independent makes overflow errors easier to analyze since s0 and s1 correspond to the power of the A and B polarities respectively.

Prior to the stokes calculation,, the PFB values are upshifted (with saturation) 0..7 bits. Any saturation at this stage (vshift) is counted and stored as status with the accumulation. s0, s1, s2, s3 are calculated to 32-bits of precision and leave the stokes block at 32-bits.

Accumulation

The accumulators integrate s0,s1,s2,s3 for a specified number of iterations and then dumps the result to the pack unit for formatting for the host computers. Prior to the accumulators, the s-parameters are downshifted 0..15 bits to provide sufficient headroom for the number of accumulation iterations. The s0 and s1 accumulators are unsigned and saturate at their maximum values. The s2 and s3 accumulators are signed and saturate at both the maximum and minimum integers. All four accumulators are 40-bits.

Packing

The packer takes the four 40-bit stokes accumulations for each frequency bin and trims this down to the data required for the current observation. The data trim is controlled by four different registers that operate independently in a reasonably flexible way.

Bytes dumped per frequency bin (bpb)			
Components	**Bits per component**		
	8	**16**	**32**
I	1	2	4
s0, s1	2	4	8
s0, s1, s2, s3	4	8	16

The register FMTWID sets the number of bits to dump for each component, this can be 8-bits, 16-bits, or 32-bits per component. FMTTYPE sets the type of data to dump. This can be only stokes-I (total power), s0/s1, or s0/s1/s2/s3 (full stokes). There are also two registers to control the range of frequency bins to dump. A contiguous range of bins from DUMPSTRT to DUMPSTOP are dumped.

A 64-bit status word is sent with each dump, this word contains a 16-bit sequence number for the dump and 32-bits of overflow and saturation information for the datapath. The 64-bit status word at the end of an integration packet is formatted as follows:

```
[15:0]      Packet sequence number, wraps at 65536
[31:16]     Actual number of integrated transforms (0..65535)
[32]        Cal input set at some point during integration
[35:33]     0
[39:36]     ADC overflow count
[43:40]     PFB overflow count
[47:44]     VSHIFT saturation count
[51:48]     ACC_S2S3 saturation count
[55:52]     ACC_S0S1 saturation count
[59:56]     ASHIFT_S2S3 saturation count
[63:60]     ASHIFT_S0S1 saturation count
```

Bits `[31:16]` change function in `HR_MODE` with `DLO_DWELL` is setup to a non-zero value. In this special case, bits `[31:16]` contain the value of the `DLO` during the integration period. This is a special case for a operation mode that sweeps the `DLO`.

When an observation is started, operation will begin at the first rising edge of PPS after the command is received. The first dump will have a sequence number of zero and increment for each following dump. Timekeeping on the dumps is done by keeping track of the sequence number and having precise knowledge of the time interval between dumps as an integral number of ADC clocks. The precise number of ADC clocks per integration can be calculated using the following information.

The following table shows the number of bytes dumped for each frequency bin (**bpb**). The number of bytes dumped per integration (**bpi**) is as follows:

```
bpi = 8 + round-up-to-next-multiple-of-8 ( bpb *
         (DUMPSTOP - DUMPSTRT + 1))
```

The time interval between dumps (**dti**) is a function of the clock period, transform length and integration parameters as follows:

```
period = 1 / 170MHz = ~5.882ns
dti = period * LEN * ( FCNT + DCNT )
```

In HR mode, **dti** impacted by the decimation of the lowpass filter (`DEC`) as follows. See the section

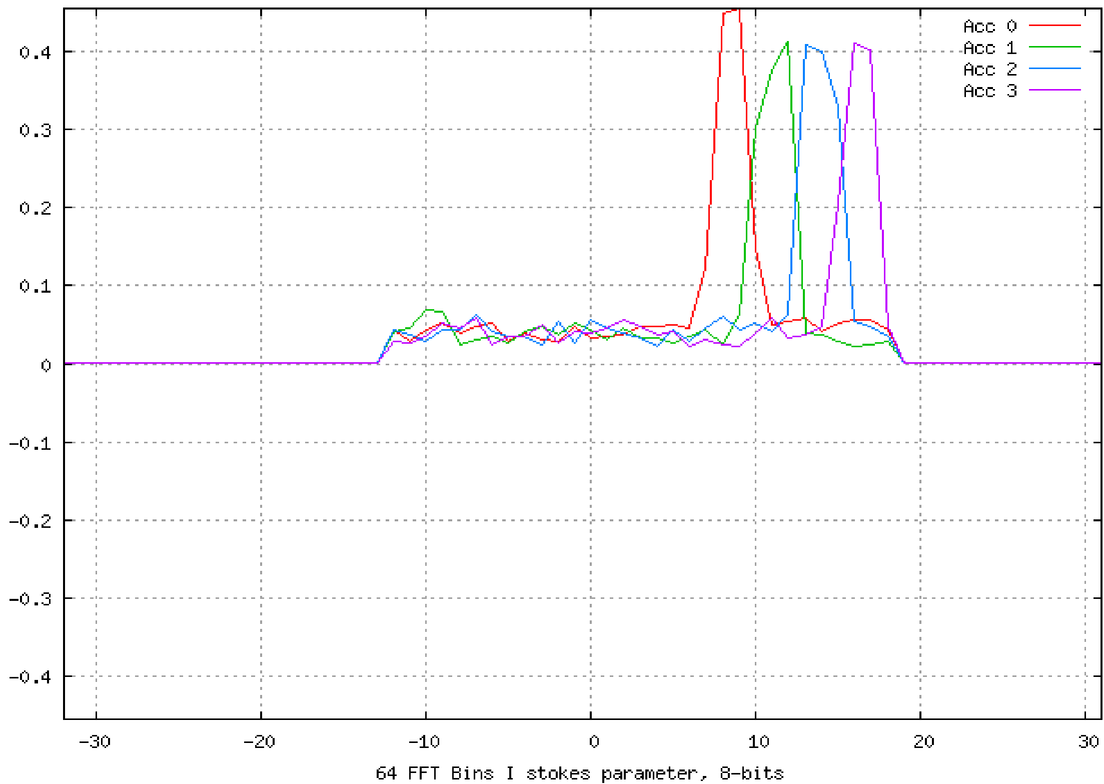
on HR mode programming for more details:

```
period = 1 / 170MHz = ~5.882ns
dti = period * LEN * DEC * ( FCNT + DCNT )
```

LEN is the length of the transform, this can be any power of two from 16 to 8192. FCNT is the number of transforms to integrate, DCNT is the number of transforms to drop between integrations. DCNT is normally zero and 100% of the input signal is integrated. The exception is when **bpb** is 16, in this case DCNT needs to be set to one and one transform is discarded per accumulation.

It follows that the total dump datarate (**tdd**) for each signal processor is:

```
tdd = bpi / dti
```



The dump parameters should be used to limit the dump datarate to the maximum that can be sustained by the filesystem of the host computers. Remember there are two signal processors in each box each independently processing a 170MHz sub-band. Here is an example plot from simulation dumping 8-bits of the stokes-I parameter across a range of bins for a simulated swept CW signal.

Overflow/Saturation

Overflow and saturation are measured at five points in the signal path. At some of these points overflow is measured for multiple parameters, a total of 7 overflow/saturations are measured. For each measurement, a count of the number of overflow/saturations is maintained. This count is distilled into a 4-bit number stored in the status word for the accumulation that provides a general indication of the number of errors. The following 7 4-bit parameters are stored in the status word:

`ovf_adc`
Overflows in the ADCs of A and B polarity inputs.

`ovf_pfb`
Overflows in the PFB

`ovf_vshift`
Saturations in the upshift after the PFB

`ovf_acc_s0s1`
`ovf_acc_s2s3`
Saturations in the s0s1 or s2s3 accumulator

`ovf_ashift_s0s1`
`ovf_ashift_s2s3`
Saturations in the upshift just prior to data packing.

For each measurement, 4-bits are kept that give an indication of the number of overflows that occurred during the integration period as follows. This convention is used for all of the 4-bit overflow/saturation fields in the dump header.

```
0      0 errors
1      1 error
2      2-3 errors
3      4-7 errors
4      8-15 errors
5      16-31 errors
6      32-63 errors
7      64-127 errors
8      128-255 errors
9      256-511 errors
10     512-1023 errors
11     1024-2047 errors
12     2048-4095 errors
13     >= 4096 errors
```

Overflows are imprecise. Due to pipelining in the signal path, errors might be recorded for a packet, but the actual packet screwed up by the overflows might be the following packet. The overflows provide a general indication that something is wrong at a certain stage of processor and should not be used for fine grain signal filtering.

During an observation, the server checks the error terms and reports errors back to the pnet program. Sometimes it's not desirable to see all of the error messages. The `err_thresh` parameter in the `[dump]` section can be used to set the threshold for each error parameter before messages are reported to the console.

SP1 register details

The following sections detail all of the registers in the SP1 signal processor. Next to each register is a field definition that is common for hardware-types, but is evidently mysterious to astronomy types. The notation [x:y] refers to a bit field that begins with LSB bit 'y' and ends with MSB bit 'x'. In this design, all registers are right justified (the LSB for the field is always at bit 0), so bit fields are noted here as [x:0] which means an x+1 wide register where the LSB is stored in bit-0. A field described as [2:0] can contain values from 0 to 7.

ADC

```
ARSEL                                 [2:0]
AISEL                                 [2:0]
BRSEL                                 [2:0]
BISEL                                 [2:0]
```

Select the source for each component of the signal processing path. AR corresponds to the real portion of polarization A, AI is the imaginary part of polarization A, etc. Cross bar selection allows versatile physical wiring and the ability to bypass an errant polarity in the ALFA receiver. The encodings are:

```
0      ADC0
1      ADC1
2      ADC2
3      ADC3
4      test signal
5      zero
```

```
ARNEG                                 [0]
AINEG                                 [0]
BRNEG                                 [0]
BINEG                                 [0]
```

Control bit to negate (2's complement) each signal component. This can be used to correct for ADC format or change exponent convention for complex signals.

Test signal

The test signal block generates a complex polarized CW signal plus noise. This is used in place of the ADC inputs if selected by the ADC selection registers. These test signal registers are not used in normal operation, they are just for diagnostic and test purposes.

```
TS_FREQ_H                             [15:0]
TS_FREQ_L                             [15:0]
```

32-bit phase increment for NCO. 32-bit unsigned number added to phase accumulator each cycle. Phase increment is 12.20 fixed point number from 0 to 2pi. FREQ_H must be written first, FREQ_L written second, phase change causes no discontinuity so software can generate a swept CW signal by rapidly changing these registers.

```
TS_PHASE                              [15:0]
```

12.4 signed fixed point number phase offset from polarity A to polarity B for test signal. Value represents [-pi..pi) phase difference for simulating elliptically polarized signal.

```
TS_CW_A                               [15:0]
TS_CW_B                               [15:0]
```

Levels for CW signal for pol A and pol B. Value is a 0.15 unsigned number [0..1) used to set level. Set differently to simulate linearly polarized signal.

```
TS_NOISE_A                            [15:0]
TS_NOISE_B                            [15:0]
```

Levels for complex noise source to pol A and pol B. Value is a 0.15 unsigned fixed point number [0..1) for setting level. Noise generator is a poor quality LFSR feedback with weird looking spectral characteristics, but it's cheap and okay for hardware diagnostics.

CMIX

`DLO` [10:0]

The frequency of the digital local oscillator (DLO) for the complex mixer. This is an 11-bit signed number. A value of 0 is a frequency of 0 and no mixing is done. A value of `0x3ff` corresponds to a mix frequency of Fs*1023/2048 (or almost Fs/2). A value of 0x400 corresponds to a mix frequency of -Fs/2. The mix frequency is the frequency in the input signal that is shifted to DC. The rest of the input signal is rotated the same amount around the unit circle.

`DLO_PHASE` [10:0]

The phase difference between the DLO sent to the complex mixer for the two pols. This is an 11-bit signed value, using the same convention for frequency as `DLO`. The phase of the B polarity is adjusted by this value. A value of 0 is no phase shift between the two pols. This value can be used to adjust the phase angle between the two polarities to correct for some phase error in the system prior to the spectrometer. This correction can be used even when `DLO` is set to zero. A complex mixer multiplies the input signal by a sequence of complex values around the unit circle. If DLO is set to zero, the A polarity is always multiplied by the unity constant (1,0) and the B polarity is multiplied by a constant value corresponding to the phase angle set by DLO_PHASE. This allow a phase rotation of the B polarity. This is really useful for simulating an arbitrarily phase difference between polarities for testing when one can't be found naturally.

`DLO_DWELL` [10:0]

This register is used for sweeping the DLO in a controlled manner during an observation. This register is only used in HR_MODE when DLO_INC is set to a non-zero value. This register sets the number integrations periods where the DLO should be held at a constant value. After DLO_DWELL integrations the DLO incremented.

`DLO_INC` [10:0]

When this register is set to a non-zero value and HR_MODE is enabled, the DLO is swept and a portion of per integration status word is changed. Every DLO_DWELL integrations, the DLO is incremented by DLO_INC.

In this mode, the status word for the integration contains the value of the DLO so post processing software knows the value of the DLO for each integration period.

DLPF

`HR_MODE` [0]

Single bit that enabled HR mode. In HR mode the input of the PFB comes from the output of the DLPF. When HR_MODE is set to zero (default), the input of the PFB comes from the complex mixer. The other DLPF registers are ignored unless HR_MODE is set to 1.

`HR_DEC` [9:0]

Variable decimation of the DLPF. Legal values are 2-1024. The datarate (and bandwidth) of the input signal is reduced by this integer value. The filter coefficients loaded into the DLPF must correspond to this decimation or bad things will happen. The polyphase filter runs at this decimated rate.

`HR_SHIFT` [4:0]

The output of the LPF is upshifted by the number of bits specified. The input signal is 12-bits, this is multiplied by 16-bit coefficients to produce 16-bit intermediate values in the filter. The final accumulation in the filter is 26-bits, 10 extra bits are retained for the maximum decimation of 1024x without overflow. The upper 12-bits of the accumulation are kept for the output of the LPF. This

26-bit accumulation is upshifted by HR_SHIFT bits. Roughly speaking, HR_SHIFT should be set to 10-log2(HR_DEC) to accommodate the decimation gain of the DLPF. The shift range is expanded to 5-bits in version-4 of the SP1 design.

`HR_OFFSET` [15:0]

After the upshift the result is offset by `HR_OFFSET` and truncated to 12-bits for the PFB. This offset is a 1.15 signed fixed point number. By properly setting `HR_OFFSET`, a DC offset in the DLPF output can be corrected. Suggested values for HR_OFFSET to facilitate rounding and removed DC offset during time domain dumping:

```
4-bit          0x0800
8-bit          0x0080
12-bit         0x0008
```

`HR_LPF` [2:0]

Normally this register is zero. Setting to a non-zero value enables time domain dumping of the output of the DLPF in one of 6 modes. Complex numbers are dumped for each component, in 4-bit mode a 4-bit real value and 4-bit complex value are dumped. The register can be programmed to dump just the A polarity or both the A & B polarities, data can be dumped with 4, 8, or 16-bits per component. In 16-bit mode, only 12-bits are actually dumped and the 16-bits is padded with zeros.

```
                                           datarate
0      Normal HR mode
1      Pol-A 4-bits per sample             fs/dec
2      Pol-A 8-bits per sample             fs*2/dec
3      Pol-A 16-bits per sample            fs*4/dec
4      not used
5      Pol-A&B 4-bits per sample           fs*2/dec
6      Pol-A&B 8-bits per sample           fs*4/dec
7      Pol-A&B 16-bits per sample          fs*8/dec
```

Setting HR_LPF to a non-zero value puts SP1 in a special mode that bypasses the polyphase filter, stokes calculations and the data packer for frequency domain data. In time domain dumping mode, SP1 just dumps time domain samples, there are no sequence numbers, error checking, or block structure to the dumped data. Data is just a stream of time domain samples.

It is important to set the `byteswap` parameter in the `[dump]` section correctly for the time domain dump mode. When 16-bit data is being dumped (modes 3 and 7), `byteswap` should be set to 1. In the other modes `byteswap` should be set to 0.

`LPF_C0` [15:0]
`LPF_C1` [15:0]
`LPF_C2` [15:0]
`LPF_C3` [15:0]

These are the base addresses of four 1024 register areas for filter coefficients of the DLPF. The filter is has an 8x overlap, but the filter is always symmetrical, so there are only 4 filter coefficient memories that are mirrored about DC in hardware. The coefficients for the DLPF are generated with the scripts `pnet_mkdlpf_coeff`.

Main

These are the main operating parameter for the signal path. In general, the datapath is put into diagnostic mode by written DIAG, other registers are written to setup operating parameters, and DIAG is written back to zero as the last step of setting up SP1.

`DIAG` [0]

Put datapath in or out of diagnostic mode so other operating parameters can be altered.

`LEN` [13:0]

Length of transform. Must be a power of two from 16 to 8192.

`PFBBY` [0]

Bypass FIR filter in PFB using only an FFT transform when set. PFB coefficient tables are ignored.

`PSHIFT` [12:0]

Downshift mask for each stage in FFT. Bit [0] is the last stage of the FFT (butterfly size 2), Bit [12] is the first stage of the FFT (butterfly size 4096) of the FFT. If the transform length is less than 8192, unused bits are ignored. At each stage in the FFT, the result is optionally downshifted 1-bit with rounding. Controlling downshift at each stage lets user find optimal balance between dynamic range, quantization error and overflow.

`SHIFT` [2:0]

After the FFT, the result is upshifted with saturation 0..7 bits.

FIR coefficients

`PFB0` [15:0]
`PFB1` [15:0]
`PFB2` [15:0]
`PFB3` [15:0]

Each of these is the base address for 8K 16-bit values for the FIR coefficients to the PFB. The values are signed 1.15 fixed point numbers [-1..1). Programming these values is a little tricky. A perl program `pnet_mkpfb_coeff` does the dirty work of creating values for these tables for different transform lengths. This program is described later in this document.

Accumulation

The accumulators take the result of the transforms and integrate each frequency bin of the transform.

`DSHIFT_S0` [3:0]
`DSHIFT_S1` [3:0]
`DSHIFT_S2` [3:0]
`DSHIFT_S3` [3:0]

Prior to integration, the individual Stokes precursor values are downshifted 0..15 bits. This downshift is to leave enough room in the upper bits to avoid overflow for the specified number of accumulations. For example, if the signal is to be integrated for 16 transforms, DSHIFT might be conservatively set to 4, downshifting 4-bits to avoid any overflow (actually saturation) during accumulation.

`FCNT` [15:0]

This is the number of signal integrations to perform for each accumulation. The minimum value is 4, the maximum is 65536.

`DCNT` [3:0]

This is the number of transforms to drop between integrations. This is normally set to 0 with no lost transforms and 100% of the input signal is integrated. The exception is when all four stokes parameters are dumped and 32-bits of accumulation is dumped for each stokes parameter. In this case, a hardware

restriction requires that this be set to 1 with 1 lost frame every integration.

SCNT [3:0]

The number of transforms to drop after synchronization at the beginning of data collection. This insures that signal integration begins with signal present at the PPS signal. The is primarily a hardware constraint. For FFT transforms this register can be set to 2, for PFB transforms this register should be set to 5. It is safe to always leave this register as 5. This insures that the pipelines are flushed after a PPS starts the observation and no pre-PPS signal is used in the first integration.

Pack

After each accumulation, data is dumped from the accumulator and packed according to these registers for use by the host system.

ASHIFT_S0 [2:0]
ASHIFT_S1 [2:0]
ASHIFT_S2 [2:0]
ASHIFT_S3 [2:0]
ASHIFT_SI [2:0]

The output of the accumulators is upshift with saturation 0..7 bits. The four stokes precursors are upshifted independently. The stokes I-parameter is calculated and shifted independently when only stokes-I dumping is used. S0,S1, and SI are unsigned values. S2 and S3 are signed values in the shift and appropriate saturation is applied to each type.

FMTWID [1:0]

The width of data to dump for each parameter. The encodings are:

```
0      8-bit values
1      16-bit values
2      32-bit values
```

The powerPC processor has big-endian byte ordering. The X86 server machines have little-endian byte ordering. The plinth code can byte swap the data from the SP so that it is stored with the correct byte order on the fileserver. The `byteswap` directive in `[dump]` must be set as follows:

FMTWID	BYTESWAP
0	0
1	1
2	3

FMTTYPE [1:0]

The type of data to dump. The encodings are:

```
0      Stokes-I            (sum of power of pol-A and pol-B)
1      s0 and s1           (Stokes I&Q)
2      s0, s1, s2 and s3   (full stokes I, Q, U, V)
```

DUMPSTRT [12:0]

Frequency bins are numbered 0..len-1. Bin-0 is the most negative frequency, bin len-1 is the maximum positive frequency and the DC bin is len/2. This parameter specifies the first bin to dump. This allows saving bandwidth by trimming unused frequency bins.

DUMPSTOP [12:0]

The last frequency bin to dump. If all bins are to be dumped, DUMPSTRT is set

to 0 and DUMPSTOP is set to len-1.

Blanking control

SP1 has two mechanisms to control signal blanking. In both cases, the blanking hardware generates a signal to the PFB to cause blanking. If the blank signal is asserted at any time during a transform, the entire transform is dropped and not accumulated. SP1 keeps track of the number of transforms actually integrated each period, this is stored in the header with each integration. If blanking is disabled the number of actual integrations is always FCNT, the number of intended integrations. With blanking enabled, the number of actual integrations can be less the specified, or even 0. The two blanking mechanisms are independent and can be used simultaneously if desired. One mechanism uses an external signal from a blanking generator, the other mechanism uses overflow information from the ADCs to generate a blanking signal.

`BLANKSEL` `[3:0]`

Select the GP input on the spectrometer box to use for the external blanking input. This GP connector should be programmed as an input with the `gpoe` parameter in `[dump]. If BLANKSEL is 0xf` (default), no external blanking is done.

`BLANKPER` `[15:0]`

When an external blanking signal is selected with BLANKSEL, BLANKPER determines how the blanking signal is handled. If BLANKPER is 0 (default), the external blanking signal is used directly to determine when blanking is active. If BLANKPER is non-zero, BLANKPER is the number of ticks to blank the incoming signal after the rising edge of the external blanking signal. The thinking is that AO has a radar blanking system that produces a rising edge prior to a periodic radar RFI source. The duration of the pulse cannot be precisely controlled, so BLANKPER allows the duration of the radar blanking to be precisely controlled to balance between sufficient blanking and signal integration time. A tick is 16 ADC clock periods (about 94ns at 170MHz).

`OVFADC_THRESH` `[15:0]`

OVFADC_THRESH is used to support blanking based on ADC overflows. OVFADC_THRESH is the number of ADC overflows that must occur during a transform before blanking is asserted. By default this is set to a large value (`0xffff`) so overflow blanking never occurs.

`OVFADC_DWELL` `[15:0]`

OVFADC_DWELL is used with OVFADC_THRESH to set the duration of the blanking interval. The duration is measured in ticks (16 ADC clocks)

Calibration Input

`CALSEL` `[3:0]`

Selects the GP input to use for calibration input. By default `0xf` is selected which means that no input is selected for cal input. A single bit is recorded in the header for each integration. If cal is asserted at any point during the integration then the cal bit is set in the header, otherwise the cal input is zero.

Calibration Output

SP1 can be programmed to generate a calibration control output signal. If this is used, it should be sent to the receiver from one of the spectrometers and looped back as an input to all of the spectrometers. When SP1 generates a cal output it is synchronized to the polyphase filter. The cal output can be programmed so that the output transitions on any FFT boundary during the

integration period.

`CALCTL` [1:0]

Bit-0 is a manual control for the cal output. If it is set then the cal output is asserted. Bit-1 enables the winking cal output. When the spectrometer is running the cal output winks according to CALON and CALOFF. One of the GP connections needs to be enabled for output with the `gpoe` directive in the `[dump]` section to drive the signal out of the spectrometer box. The cal output actually comes from GXA, but there is no harm in programming both GXA and GXB to generate cal.

`CALON` [15:0]

Number of integrations for cal to be asserted.

`CALOFF` [15:0]

Number of integrations for cal to be deasserted.

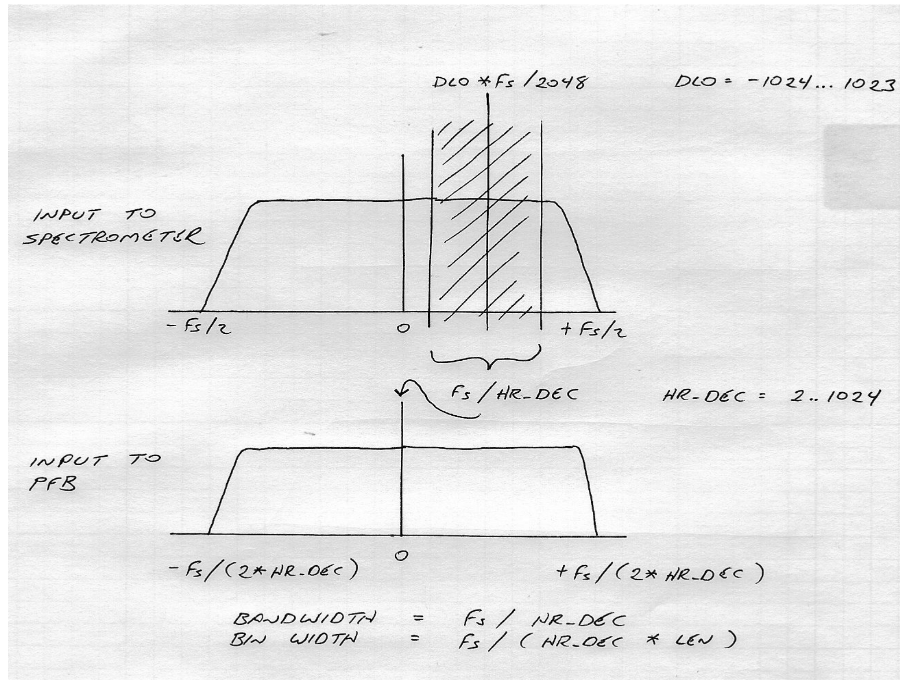
`CALPHASE` [15:0]

Sets the position during the integration when the cal output transitions. If set to 0, the cal output will switch at the beginning of the integration. Due to pipelining delays, the actual output signal of the spectrometer will transition about 40ns after the integration begins. `CALPHASE` can be as large as `FCNT-1`, in this case the cal output will transition as the last FFT of the integration period begins. Normally, `CALPHASE` should be set to `FCNT/2` so that the cal output transitions in the middle of the integration period.

If `CALPHASE` is set to 0, `CALON` and `CALOFF` are set to 1, the cal output will switch at the beginning of every integration. If this cal signal is externally looped back to the cal input of the spectrometer, the pdev dump file will show cal being active in every integration. This is because the cal output signal transitions slight after the beginning of the integration and possibly even later counting cable delays from the cal output to the receivers and back to the cal input. The cal input recording will record a 1 if the cal input is asserted any any time during the integration. Regardless of the `CALPHASE` setting, if `CALON` is active for n integrations, the looped back cal input signal will show active for n+1 because of the misalignment between cal output and the actual integration.

HR mode programming

Rev.2 of the FPGA adds a new feature for integrating very narrow frequency bins in a high resolution mode. The polyphase filter has a maximum length of 8192 points. When HR mode is enabled, the DLPF is used to decimate the input rate by some integer ratio and the polyphase filter is run over a narrowband signal as shown in the following figure. The DLPF is not used in non-HR modes.



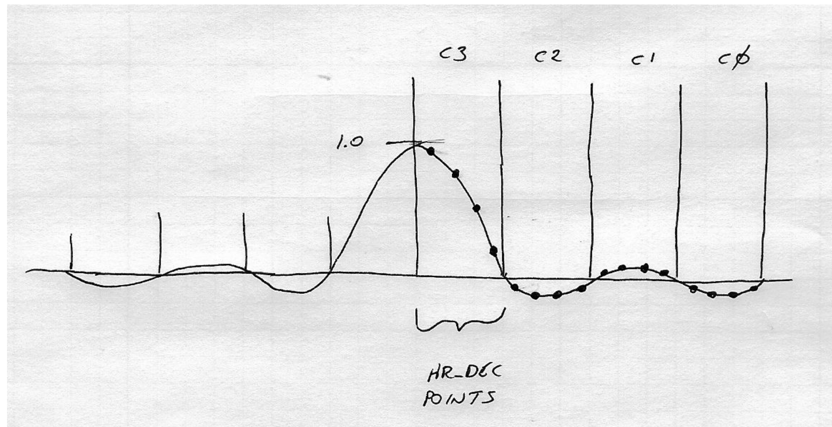
The first step in setting up HR mode is to program the complex mixer to select a center frequency for the spectrum. This is controlled with the DLO register. The mix frequency (the DC frequency for the polyphase filter) is `DLO*Fs/2048`. `DLO` is an integer from -1024 to 1023.

HR mode is controlled in the `pnet.conf` file like other SP1 parameters. HR mode is enabled by setting `HR_MODE` to 1. The decimation for the DLPF is set with `HR_DEC`. `HR_DEC` be set to any value from 2 to 1024. The bandwidth of the input signal is reduced by a factor of `HR_DEC`.

HR_SHIFT and HR_SCALE scale the out levels of the filter as described in the register section above. The following example sets up for 20x decimation. With a 170 MHz ADC clock, the PFB does a transform across 170/20 = 8.5 Mhz. With an 8192-pt transform the bins are 1.03 kHz. The transform time in HR mode is reduced by the decimation rate. The transform time for this example is (20*8192)/170 = 963 us.

```
HR_MODE     1
HR_DEC      20   # 2..1024
```

The DLPF must be loaded with filter coefficients to match the decimation rate. The filter has an 8x overlap. The filter is symmetrical so only four coefficient tables are needed. The filter coefficient registers are organized as follows:



The filter coefficients are generated with the pnet_mkdlpf_coeff program as follows:

```
% pnet_mkdlpf_coeff --dec=20 --fn=filter --window=hamming
% ls -l
-rw-r--r-- 1 jeff users 100 2007-03-01 22:06 filter.0020.0
-rw-r--r-- 1 jeff users 100 2007-03-01 22:06 filter.0020.1
-rw-r--r-- 1 jeff users 100 2007-03-01 22:06 filter.0020.2
-rw-r--r-- 1 jeff users 100 2007-03-01 22:06 filter.0020.3
% cat filter.0020.3
7fdb
7eb8
7c76
791f
74c3
6f75
694c
6264
5ada
52ce
4a63
41b9
38f5
3037
27a2
1f53
1766
0ff5
0917
02dc
```

The filter coefficients are 1.15 signed fixed point numbers with a range of [-1..1), they are loaded in the pnet.conf file as follows:

```
LPF_C0       file filter.0020.0
LPF_C1       file filter.0020.1
LPF_C2       file filter.0020.2
LPF_C3       file filter.0020.3
```

Generating PFB FIR coeffcients

`pnet_mkpfb_coeff` generates coefficients for the PFB FIR filter. As part of the pnet installation, tables for a typical filter tables for various transform lengths are stored in `/usr/local/pdev/etc as follows:`

```
pfb.16.hamming
pfb.32.hamming
pfb.64.hamming
pfb.128.hamming
pfb.256.hamming
pfb.512.hamming
pfb.1024.hamming
pfb.2048.hamming
pfb.4096.hamming
pfb.8192.hamming
```

These files should be suitable for most applications, but if you are picky (and you know who you are), you can generate custom coefficient tables using `pnet_mkpfb_coeff` or modify this script to generate optimized filter coefficients.

... add drawing of filter ...

-- add instructions for running pnet_mkpfb_coeff ...

Pdev dump file format

Pdev creates dumps files in the same format regardless of the SP being used. The file format is simple. A 1024-byte header is followed by the data as it is dumped from the SPs. The first eight 32-bit words of the dump are as follows:

```
0xdeadbeef          Pdev magic number
magic               from [dump] section of pnet.conf
adcf                ADC clock freq in Hz (+/- 0.01%)
byteswap            from [dump] section
blksiz              size of each block (integration)
n                   number of blocks dumped
beam                from [pdev] section
subband             from [pdev] section
```

In version 2 of the pdev dump file, the initial header is increased from 8 32-words to 32 32-bit words. The first 8 words are the same as the version 1 header except the magic number is changed to indicated a version 2 file. The initial 32 32-bits are as follows in a version 2 file:

```
0xfeffbeef          Pdev magic number
magic               from [dump] section of pnet.conf
adcf                ADC clock freq in Hz (+/- 0.01%)
byteswap            from [dump] section
blksiz              size of each block (integration)
n                   number of blocks dumped
beam                from [pdev] section
subband             from [pdev] section
lo1mix              from [dump]
lo2mixlow           from [dump]
lo2mixhigh          from [dump]
adcclk              from [dump]
time                Start time in UTC seconds
resv1               set to 0, reserved for Phil
resv2               set to 0, reserved for Phil
if1                 from [dump]
# remaining 16 values are currently undefined
```

The remainder of the 1024-byte header contains 16-bit values corresponding to the `[header]`

section of the configuration file. This is the SP specific header information. The maximum file size is specified in the `[dump]` section. A new file with an incrementing sequence number is opened when the maximum file size is reached. The offset to these SP header values is different between version-1 and version-2 pdev files. In version-1 files, the user header offset is 32-bytes from the beginning of the file. In version-2 files, the user header offset is 128-bytes from the beginning of the file.

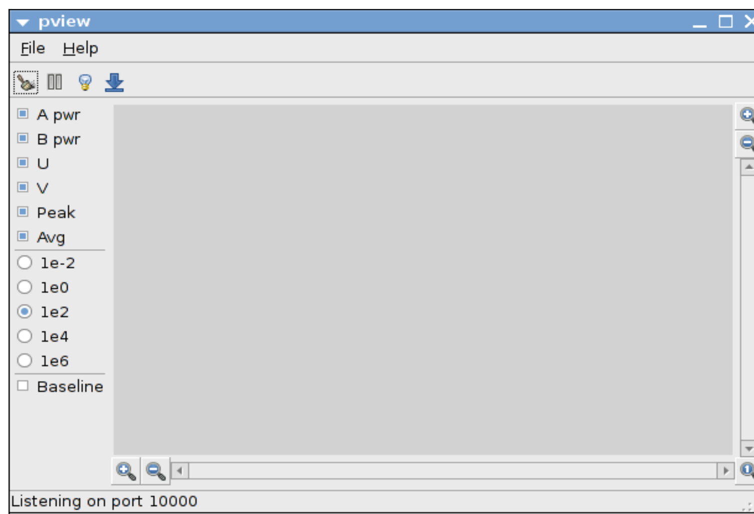
pview

Pview is an engineering diagnostic program for viewing the output of the spectrometer in real-time. Pview acts as a replacement for psrv, the server-side program that stores the output of the spectrometer. Instead of storing the output of the spectrometer to a file it does a real-time display.

pview should be run on a computer with a high performance graphics display and a gigabit network connection to the spectrometers under test. Pview replaces psrv, so it sees the same network bandwidth as a server machine. Pview can be run as a networked X11 application with good results, but best results with a local graphics display.

Pview is a multi-threaded application. One thread reads the spectrometer data over the network connection and processes the data for display. Another thread updates the display with an animated display of the spectral data. Both of these threads can consume quite a lot of CPU resources, so pview will likely benefit from being run on a multi-processor machine.

Pview should be started by the user from the command line. The program is still a bit rough and some diagnostic messages will appear on the terminal where the application is started. Start pview from the command prompt and you should get a boring looking window something like this:



Pview works by listening for a network connection from a spectrometer. The status bar should show that pview is listening for a connection on port 10000 of the local machine. pview will try port 10000 first and and try higher numbered ports until it finds an unused port. If pview is aborted unexpectedly the operating system may keep the listening port active for a few minutes, if pview if re-launched it will use a higher numbered port like 10001. Note this port number.

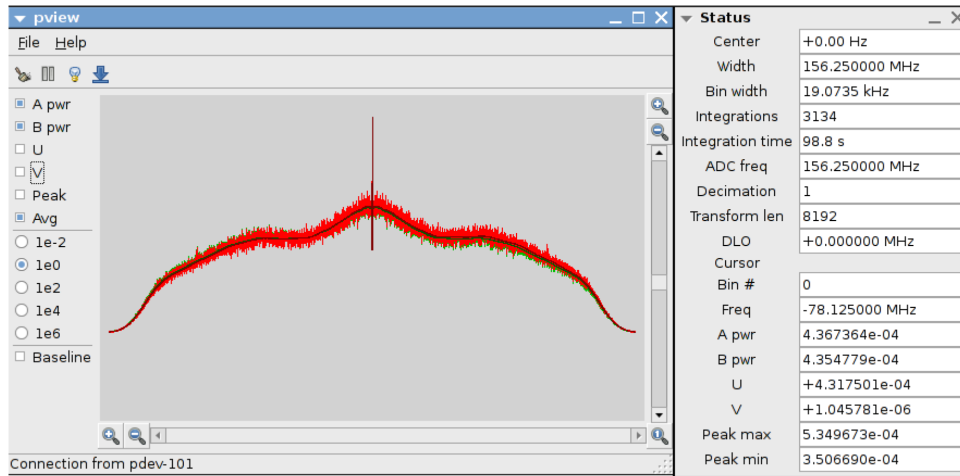
To use pview, the `pnet.conf` file for an observation needs to be modified slightly. The first change is that the `[pdev]` section needs to be modified to point to pview as the file server machine for the observation. The `[pdev]` section is normally in the file `boxes.conf`. You can

create a local `boxes.conf` file or not include boxes.conf in your `pnet.conf` file and add the `[pdev]` section directly. The modified `[pdev]` section should look something like this:

```
[pdev]
# name   IP address   Beam    Subband sp   setup    file server
#
b0sp0    pdev-101     0       0       0    gxa      laptop:10000
```

The change is to specify the file server as `machine:port` where port is the port number pview is listening on.

From a new terminal window, use pnet to start the observation as normal. Pnet will connect to the spectrometer and the spectrometer will connect the the file server which is really the pview application running on a workstation. Spectra are dumped as normally, but are now displayed in the application. Once pnet is started you should see an animated display like this:



The horizontal and vertical zoom buttons and scroll bars can be used to zoom in on interesting parts of the signal. The check boxes on the left side enable signals for display. The tooltips should provide some information if you hold the mouse over the buttons. In this case I changed the default Y scaling to 1e0 and zoomed and scrolled a bit on the Y-axis to get a nice animated image. The Status window is displayed (using the light bulb button on the toolbar) to show additional technical information and the last location of the cursor in the display window.

The radio buttons on the left side scale the Y axis. With 32-bit dumps in particular, the signal can have a wide dynamic range that is difficult to control with the vertical zoom buttons. By setting the scaling with the radio button it is easier to display the interesting portion of the signal.

There is a status window that can be display with a button on the toolbar. When the status window is display, clicking the mouse in the display window will show detailed frequency and level information for the signal.

A few notes about performance using pview. For best results, the spectrometer should be configured to output at least a few integrations per second. Some aspects of the user interface are updated at the integration rate and the application can feel sluggish if the spectrometer outputs data too slowly. The spectrometer should be programmed to output data probably no faster than about 100 integrations per second. The workstation might not be able to keep up with drawing data at faster frame rates. It is probably desirable to use a CPU/network monitor on the workstation to keep track of CPU and network usage. These can change dramatically depending

on the number of signals being displayed and the size of the windows. Pview will attempt to drop integration frames gracefully if the internal FIFOs start to fill, but your mileage may vary if system resources are exceeded. Another good indicator about performance problems is the number of integrations in the status window. The networking code maintains a FIFO of about 50MB of data from the spectrometer. When the FIFO is more than half full, pview will start dropping integration packets to try and keep the FIFO less than half full. If this starts happening, the status window will start showing the number of packets actually integrated along with the number of packets dropped. This is an indicator that the workstation doesn't have enough processing power to keep up with the spectrometer. A typical solution is to increase the integration time in the spectrometer to slow the datarate to pview. Clearing the integration buffer (the broom icon in the toolbar) will set all of these counters back to zero. As a reference point, using my circa 2005 dual-core laptop (two 1.667GHz cores) with gigabit ethernet, I find that I can keep up with dump rates of about 25 MB/s from the spectrometer. A faster machine can probably do quite a bit better and likely keep up with the maximum specified 50MB/s dump rate of the spectrometer.

During normal operation with no dropped integrations, the network buffer will stay nearly empty. The FIFO depth can be monitored by running pnet with the -d option. The FS-buf column is the size of the network fifo on pview. When performance is close to the edge and packets are being dropped, the FIFO will linger near half full (25MB). The causes some delay in the user interface. If the dump rate is 25MB/s, the display will lag the realtime signal by about 1s, so a change of input signal is delay by about 1s in the display. In normal operation with no dropped packets the delay is not noticable.

In the `[dump]` section of `pnet.conf` is a parameter called `dma`. This is normally left at 65536. When using pview for interactive display it might be desirable to change this value. The dma parameter sets the size of dma done by the powerPC processor in the spectrometer. In normal operation a larger value makes the spectrometer run more efficiently. With pview, it is desirable to make this value smaller. For example, if the dump size of each spectrum is small, say 10k bytes and the integration rate is low, say 1 integration per second, the powerPC will process a full DMA buffer before sending data to pview. This will cause a jerky display where the pview will sit idle for about 6-seconds and then quickly display the six integrations that fit intput a 64k DMA. By changing the DMA parameter from 65536 to 1024, the individual DMAs are smaller and pview operation is much smoother.

```
[dump]
dma          1024          # Normally 65536, 1024 for pview
```

Pview can do some basic baseline removal. After integrating an off-source signal, push the down-arrow button in the toolbar. This stores the current signal as a baseline reference. When the baseline check box on the left is checked, the baseline is removed from the current signal. After switching to baseline removal, the display may look quite strange. Change the Y-axis scaling to 1e-2 and the veritcal zoom buttons to see the signal with appropriate scaling. The baseline removal is (a-b)/b. The baseline is only valid for the current observation. It pnet aborts or finishes normally and the observation is restarted, the baseline is lost and a new baseline needs to be stored.

Pview is designed to display the output of a single SP on a single spectrometer. If you have sufficient computing resources, you can run multiple copies of pview at the same time. Each copy of pview listens on a different port. By setting up the pnet.conf file properly, two different SP's can be run at the same time and send to different pview windows. Be really careful that you don't run out of network or CPU resources.

Pview can save the current spectrum as a text file from the menu with File->Save. This will save the current integrated spectrum along with the baseline signal and bin number. This should make is easy to important data from an engeering test into a spreadsheet or some other program for

further analysis.

Pview is intended to be pretty reliable. You should be able to start pview and leave it running for a long time. If there is no network connection from a spectrometer it consumes very little resources. If pnet/prun aborts unexpectedly, pview should recover gracefully and listen for a new connection. A typical session should consist of starting pview once and possibly trying many different observation parameters in the pnet.conf file, starting and aborting pnet many times to test different parameters. Pview should be very stable in this situtation.

spcvt

spcvt is a server program for manipulating pdev dump files. It can do a few useful things to any pdev file, and a few cool things with SP1 specific pdev files. It can serve as an example for writing custom programs to process pdev files.

```
Usage:  spcvt [options] file ...

   Operations on any pdev file
   [--info]        print info about file
   [--get=n]       get block n and write it to stdout
   [--cmp]         Compare two observations

   SP1 specific commands
   [--csv=n]       Dump n blocks in text format for spreadsheets
   [--plot=n]      Plot n blocks using gnuplot
   [--blk=n]       Starting block number for csv command
   [--avg=n]       Number of blocks to average for csv command
   [--norm]        Normalize values in CSV
   [--log]         Normalize and output values in log -dBc
   [--start=n]     Starting bin for plot
   [--stop=n]      Last bin for plot
   [--check]       Check headers for seq numbers and overflows

   [--fits]        Convert PDEV file to FITS file
   [--maxrows=n]   Maximum number rows (blocks) in each FITS file
   [--hdr]         Print headers for each SP1 block
```

When spcvt opens a pdev file it verifies that the length of the file(s) matches the header and the file has correct magic numbers.

[--info] prints out interesting information about the file. If spcvt has specific information about the SP type, it will interpret. Sample output for an SP1 file:

```
[jeff@ao dump]$ spcvt --info x1234.20070111.b0a.00000.pdev
Info:
    Initial sequence    00000
    Base filename       x1234.20070111.b0a
    ADC freq            156.25 MHz
    Number of files     1
    Filesize            6555424
    Total size          6.56 MB
    Byteswap            3
    Number of blocks    100
    Block size          65544 bytes
    SP magic            0x2e83fb01
    Beam                0
    Subband             0
```

```
    User header:
        0002 0002 1000 0000 0fff 0140 0001 0000
        0001 0002 0003 0000 0000 0000 0000 0000
        1555 0000 0002 0002 0002 0002 0002 0002
        0002 0002 0002 0005 2000 2423 0000 0010
        0008 0040 0040

SP1 Info:
    Transform length    4096
    Start bin           0
    Stop bin            4095
    Component width     2 (32-bit)
    Dump type           2 (full stokes)
    Frames integrated   320
    Frames dropped      1
    PFB bypass          0
    PSHIFT              0x1555
    SHIFT               0
    DSHIFT_S0           2
    DSHIFT_S1           2
    DSHIFT_S2           2
    DSHIFT_S3           2
    ASHIFT_S0           2
    ASHIFT_S1           2
    ASHIFT_S2           2
    ASHIFT_S3           2
    ASHIFT_SI           2

    Pol A input         (ADC0, ADC1)
    Pol B input         (ADC2, ADC3)
    Integration time    8.41 ms
    Bin width           38.15 kHz
    File time           0.84 s
```

[--cmp] compares pdev files that might are supposed to be identical aside from the header. [--get=n] extracts a specified block from the file and writes it to standard out.

spcvt has a number of other options for processing SP1 specific files. The most useful is the [--fits] option to convert a pdev file to a FITS file. This is an example conversion intended to serve as an example for a more sophisticated FITS conversion suitable for science applications.

Creating a new SP

The SP is a signal processing datapath. It takes input from the ADCs, processes it in someway, and produces output at a decimated datarate. There is a simple interface to read/write 16-bit registers and a few control signals to start and stop observations. The intention is that an engineer (or inspired astronomer) can create a new signal processor module, a corresponding `pnet.conf` configuration file, and do observations and data collection without writing any software or dealing with any system issues. Proficiency with verilog is required to write a new SP.

Here is the top level verilog interface to an SP:

```
module sp (
    ck,
    reset,
    master,
    obs_start,
    obs_stop,
```

```
    ctl_we,
    ctl_addr,
    ctl_data,
    ctl_rd_data,

    gp,
    gp_out,

    adc0_dat,
    adc0_ovl,
    adc1_dat,
    adc1_ovl,
    adc2_dat,
    adc2_ovl,
    adc3_dat,
    adc3_ovl,

    pack_dat,
    pack_vld,

    sp_id,

    extra_cc,
    extra_hdr
);

    // The ADC clock, nominally 170MHz, all signals
    // are synchronous to this clock.
    //
    input                       ck;

    // Reset for the signal processor, the powerpc can
    // assert this with a register write.  External reset
    // also causes this to be asserted.
    //
    input                       reset;

    // This is a DC signal connected to a strapping option on
    // the board.  One of the FPGAs is indicated as master and
    // the other slave.  This is used for selecting the FPGA
    // the does the sampling of the external GPIO pins.
    //
    input                       master;

    // Strobes to start and stop an observation.  SP should
    // begin processing signals when obs_start is asserted for
    // one cycle.  sp should stop dumping data a short time after
    // obs_stop is asserted for a cycle.  Typically obs_start
    // is asserted a deterministic time after PPS when the
    // hardware is armed in the plinth code.  For testing,
    // obs_start can be asserted manually in software, multiple
    // fpgas on the same board are synchronized on a manual
    // start.
    //
    // obs_stop is not necessarily synced across multiple
    // fpgas or boxes.
    //
    input                       obs_start;
    input                       obs_stop;
```

```
// Diagnostic bus interface.  Writes are mapped into the
// powerpc interface.  A register write is a single cycle
// pulse on ctl_wr with ctl_addr and ctl_data valid
// during the cycle.
//
// Registers reads (if used) are a little goofy. The address
// is written to ctl_addr (with no strobe to indicate this),
// the signal processor places the read data on ctl_rd_data.
// The powerpc reads an indirect register to collect the data.
// The timing for reads is not critical since the address
// and data phases of the read are separated by software on
// the ppc.
//
input                       ctl_we;
input   [15:0]              ctl_addr;
input   [15:0]              ctl_data;
output  [15:0]              ctl_rd_data;

// These are the conditioned general purpose inputs to the
// box.  These are used for pps, blanking, cal, etc. This
// signals are already resampled to the ADC clock and
// synchronized across the two FPGAs.
//
input   [`GP_IN-1:0]        gp;

// This is the output signal for the GP pins.  The pins
// are normally terminated inputs.  If a bit is set
// in the pi register PREG_GPOE the pin becomes an output
// and this signal drives it.  Only the FPGA with master
// asserted can actually drive the GP outputs of the box.
//
output  [`GP_IN-1:0]        gp_out;

// The four 12-bit ADC inputs and overflow for the ADCs.
//
input   [`N_ADC-1:0]        adc0_dat;
input                       adc0_ovl;
input   [`N_ADC-1:0]        adc1_dat;
input                       adc1_ovl;
input   [`N_ADC-1:0]        adc2_dat;
input                       adc2_ovl;
input   [`N_ADC-1:0]        adc3_dat;
input                       adc3_ovl;

// This is the primary output of the signal processor, a
// 64-bit bus and a data valid flag on the same cycle
// the databus is valid.
//
output  [63:0]              pack_dat;
output                      pack_vld;

// This is likely a constant value that indicates the
// design and revision number.  By convention, the upper
// 8-bits is the design and the lower 8-bits is the
// revision number.  Software in the ppc can read this
// to query the signal processor in the fpgas.
//
output  [15:0]              sp_id;
```

```
    // These are directly connected to bonding pads for extra
    // signals.  extra_cc is connected to the other FPGA and
    // extra_hdr are connected to headers on the board
    // for debug or other crazy functions.
    //
    inout   [35:0]              extra_cc;
    inout   [35:0]              extra_hdr;
endmodule
```

The signal processor runs off a single clock, `ck`. This clock is the ADC clock. GX also runs off a second clock synchronous to the powerPC I/O bus, but this is hidden from the SP. Everything in SP runs off `ck`. The clock signal has been conditioned, all inputs an outputs from the SP are registered to the clock signal.

The `reset` signal should set the SP to a known quiescent state. The SP should not make any assumptions about FPGA initialization. The `reset` signal is asserted by a system reset or by a plinth register write. The SP should not generate any output after a reset until `obs_start` is asserted.

`master` is a DC signal connected to an external pullup/pulldown resistor. The first GX chip, GXA, has a pullup resistor connected to this pin, all other FPGAs have a pull down resistor. The master FPGA is used to synchronize external GP signals and distribute them to the other FPGAs. For the most part, the SP does not need to know if it is running on the master FPGA.

`obs_start` and `obs_stop` are used to start and stop observations. `obs_start` is asserted for a single clock cycle. In normal operation, the clock pulse is synchronized to PPS and all FPGAs are synchronized to get `obs_start` on the same clock cycle. `obs_stop` is asserted for a single cycle to stop the observation. `obs_stop` is not synchronized across the FPGAs. The SP should stop generating output after `obs_start` is asserted, this requirement is a little soft, the SP should stop sending data within a new milliseconds.

The register interface with the `ctl_*` signals is quite simple. A register write is a single cycle operation. `ctl_we` is asserted for a single cycle with `ctl_addr` and `ctl_data` valid. The address should be the same value used in the `[defs]` section of the configuration file. All registers are 16-bits and there can be up to 65536 registers. All of the configuration for the SP should be done through this register interface. It is not necessary for all of the registers to be initialized by a reset, but enough state should be initialized to insure that he SP does not produce any output until `obs_start` is asserted.

The `gp` inputs can be used for any purpose needed by the SP like calibration, blanking, etc.

The `gp_out` signals can be driven with any output signal the SP needs to generate. When the SP generates an output, the plinth code needs to be program the GP connector as an output with the `gpoe` parameter in the `[dump]` section.

The ADC signals are pretty self explanatory. The ADC produces an overflow the SP can use if desired.

The SP should provide an 16-bit ID. This is read by the plinth code and the pnet application software to insure the correct SP is loaded. The SP also has access to extra signals with no defined function. 36-bits are connected to a header next to the FPGA, and 36-bits are connected to the other FPGA on the PDEV spectrometer.

The most important output signals are `pack_dat[63:0]` and `pack_vld`. This 64-bit bus is how processed data is sent from the SP to fileservers for storage. After an observation is started with

obs_start, the SP can send 64-bits of information on any given cycle by asserting pack_vld.

Plinth code manages byteswapping of the 64-bit data bus, there is no flow control. The assumption is that the SP is processing a real-time signal and flow control is not possible. The plinth code provides a FIFO between the real-time data from the SP and the DMA controller moving data to main memory of the powerPC. This FIFO is 2^20 64-bit words deep. So, the SP can conceivably dump a 64-bit word every cycle for 2^20 cycles before possibly filling the FIFO, so it's possible for an SP to do relatively large data bursts.

Software on the powerPC and fileservers keep track of the plinth FIFO as well as other FIFOs implemented in software. Plinth calculates parity on the 64-bit bus, stores parity with the data in the FIFO, and sends the data with parity over the I/O bus of the powerPC and the powerPC DMA controller checks parity when the data is stored in main memory. Pnet insures that all of the data sent by the SP arrives at the fileserver reliably. If a FIFO overflow or a parity error occurs, the errors are report to the user by pnet.

The powerPC IO bus has a maximum sustained bandwidth of about 110MB/s. This bus is shared by two FPGAs. The network connection from the spectrometer to the fileserver is gigabit ethernet, the maximum bandwidth of this connection is something less than 100MB/s. The SP should insure that the two SPs in the spectrometer keep their average bandwidth below these limits.

Building an SP

To build an SP you need a source tree for the PDEV system. To get the current source tree use subversion (svn) on a Linux machine:

```
% svn checkout https://www.mock.com/svn/pdev/trunk pdev
```

Read the subversion documentation to take advantage of the feature of the source code control system.
To use the simulation environment, get a copy of the GPL programs cver and gtkwave. These programs are a verilog simulator and waveform viewer.

The GX design files are in pdev/gx. The easiest way to start an SP design is to make a copy of SPEX, the small example SP, change the name of the project in Makefile.defs, and verify that you can build the SP:

```
% cp -r pdev/gx/spex pdev/gx/spnew
% vi pdev/gx/spnew/src/Makefile.defs
% cd pdev/gx/spnew/build
% make
```

The example signal processor contains a simulation test bench and a few examples for programming the new SP and generating test signals.