ATA SNAP Firmware Manual

Release 2.1.0

Jack Hickish

Oct 18, 2021

CONTENTS:

CHAPTER

ONE

SNAP FIRMWARE SPECIFICATIONS

1.1 Requirements

1.1.1 ADC Sample Rate

The F-Engine firmware shall pass timing analysis for a maximum ADC sample rate of 2048 Msps.

1.1.2 Frequency Channels

The F-Engine firmware shall internally generate 4096 complex-valued frequency channels over the digitized Nyquist band. For an ADC sample rate of 2048 Msps, this represents a channel bandwidth of 250 kHz. Channels shall be generated using an 8-tap PFB frontend.

1.1.3 Input Coarse Delay

The F-engine design shall provision for a runtime programmable coarse (1 ADC sample precision) delay, individually set for each analog input. The maximum depth of this delay shall be at least 8192 ADC samples (4096 ns at 2048 Msps sample rate)

1.1.4 Output Bandwidth

The F-engine design shall be capable of outputting 4096 frequency channels at 4+4 bit complex resolution per sample, or at least 2048 frequency channels of bandwidth with 8+8 bit complex resolution. At 2048 Msps sampling clock, this corresponds to 1024 MHz bandwidth at 4+4 bits resolution, or 512 MHz bandwidth at 8+8 bits resolution.

Output is via UDP packet streams over a pair of 10Gb/s Ethernet interfaces.

1.1.5 Output Format

Data shall be output using Ethernet jumbo frames. Data payloads shall group multiple polarizations, frequency channels, and time samples in each packet ordered (fastest to slowest axis) as `polarization x time sample x freq. channel`

A variety of outputs with different numbers of samples, and frequency channels gathered in each packet shall be provided.

1. 16 times, 256 channels, 2 polarizations, 4+4 bit (8kB packets; 16 packets / 4096 channels)
2. 16 times, 128 channels, 2 polarizations, 4+4 bit (4kB packets; 32 packets / 4096 channels)

3. 16 times, 128 channels, 2 polarizations, 8+8 bit (8kB packets; 16 packets / 2048 channels)
4. 16 times, 64 channels, 2 polarizations, 4+4 bit (4kB packets; 32 packets / 2048 channels)

1.2 Specification

1.2.1 ADC Sample Rate

The F-Engine design meets timing at a sample rate of 2048 Msps (FPGA DSP pipeline clock rate of 256 MHz)

1.2.2 Frequency Channels

The F-engine firmware generates 4096 complex-valued frequency channels using an 8-tap PFB with a 25-bit precision FFT. This FFT has sufficient dynamic range to negate the need for FFT shfit control.

Scaling coefficients are provided with a dynamic range exceeding that of the FFT. Any FFT output power can be effectively converted to either 4- or 8-bit data.

1.2.3 Input Coarse Delay

The F-engine design provides per-input programmable delay of up to 16384 ADC samples

1.2.4 Output Bandwidth

The F-engine design can output all 4096 channels at 4+4 bit resolution.

The F-engine design can output up to 2304 (576 MHz bandwidth with a 2048 Msps sample clock) at 8+8 bit resolution using 128 channels per packet.

The F-engine design can output up to 2432 (608 MHz bandwidth with a 2048 Msps sample clock) at 8+8 bit resolution using 64 channels per packet.

Output is via UDP packet streams over a pair of 10Gb/s Ethernet interfaces, with each interface providing a different subset of the total bandwidth.

The start point of the transmitted block of channels can be placed arbitrarily with a precision of 8 channels.

1.2.5 Output Format

Data packets can be constructed with payloads either in (fastest to slowest axis) `polarization x freq. channel x time sample` or `polarization x time sample x freq channel` order.

Data payloads can be constructed with 16 time samples per packet, and any multiple of 8 (4+4 bit mode) or 4 (8+8 bit mode) frequency channels.

Any total number of frequency channels may be transmitted, subject to these restriction, and the total Ethernet bandwidth available.

In 4-bit mode, data may be optionally swizzled as per https://github.com/realtimeradio/ata_snap/issues/7 .

Each data packet has an application header of 16 bytes. Total protocol overhead (including this application header) is 70 bytes per packet (<2% for 4kB packets).

CHAPTER

TWO

ATA SNAP F-ENGINE FIRMWARE USER MANUAL

2.1 Introduction

This repository contains firmware and software for a SNAP-based F-Engine (i.e. channelizer) system for the Allen Telescope Array (ATA).

The system digitizes RF signals from the ATA at a speed, f_s, of up to 2500 Msps and generates 4096 frequency channels over the Nyquist band. Two operational modes are supported: *Spectrometer* mode, and *Voltage* mode. These different modes, shown in a high-level block diagram in `fig-ata-snap-feng-overview`, apply different processing to the channelizer output prior to outputing data over 10 gigabit Ethernet (10GbE).

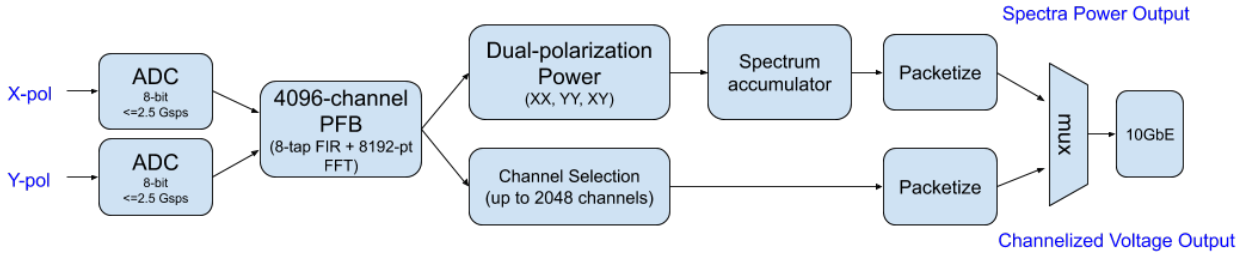

Fig. 2.1: The ATA SNAP F-Engine firmware comprises two pipelines. One of these – the *Spectrometer* pipeline – generates accumulated power spectra. The other – the *Voltage Pipeline* – generates a complex voltage stream channelized into 2048 frequency bins.

2.1.1 Spectrometer Mode

In *Spectrometer* mode, integrated power spectra are generated by the SNAP Firmware. The resolution of these spectra, f_c, is $\frac{f_s}{8192}$. At maximum sampling rate, $f_s = 2500$ Msps, resulting in a spectra resolution $f_c = 305$ kHz.

Spectra are generated for both the X- and Y-polarizations, which are formatted as an f_c-element array of 32-bit signed integers. A cross-power spectra, defined as the multiple of an X-polarization spectra with the complex conjugate of a Y-polarization spectra, is also computed. This is formatted as an f_c-element array of 32-bit complex (i.e. 32-bit real, 32-bit imaginary) signed integers.

Data are accumulated for a runtime-configurable number of spectra, and are output from the firmware as a UDP data stream over a 10 Gb Ethernet link. The output format is described in detail in *Output Data Formats*.

2.1.2 Voltage Mode

In *Voltage* mode, up to 2048 frequency channels - each with a resolution of $\frac{f_s}{8192}$ - are output over a 10 Gb Ethernet link.

In this mode, data are transmitted as complex, signed, 4-bit integers over a UDP data stream.

2.1.3 This Document

This document describes the hardware configuration required by the F-Engine system, the runtime configuration procedures, and the software control functionality made available to the user. It also provides a description of the output data formats of each of the two data processing modes.

2.2 Nomenclature

2.2.1 Data Types

Throughout this document, data types are labelled in diagrams using the nomenclature $X.Y\mathrm{b}$. Unless otherwise stated, this indicates an X-bit signed, fixed-point number, with Y bits below the binary point.

Where this document indicates an N-bit complex number, this implies a $2N$-bit value with an N-bit real part in its most-significant bits, and an N-bit imaginary part in its least-significant bits.

2.3 Output Data Formats

2.3.1 Spectrometer Packets

In versions >1.1.x of the SNAP firmware, each spectrometer dump is a 64 kiB data set, comprising 4096 channels and 4 32-bit floating point (IEEE 754 single precision: 1-bit sign, 8-bit exponent, 23-bit fraction) words per channel. Each data dump is transmitted from the SNAP in 8 UDP packets, each with an 512 channel (8 kiB) payload and 8 byte header:

```
#define N_c 512
#define N_p 4

struct spectrometer_packet {
  uint64_t header;
  float data[N_c, N_p] // IEEE 754 single precision float
};
```

The header should be read as a network-endian 64-bit unsigned integer, with the following bit fields:

- bits[7:0] (i.e. `header & 0xff`) : 8-bit antenna ID
- bits[10:8] (i.e. `(header >> 8) & 0x7`) : 3-bit channel block index
- bits[55:11] (i.e. `(header >> 11) & 0x1fffffffffff`) : 45-bit accumulation ID
- bits[63:56] (i.e. `(header >> 56) & 0xff`) : 8-bit firmware version

Headers fields should be interpretted as follows:

- *Antenna ID*: A runtime configurable ID which uniquely associates a packet with a particular SNAP board and antenna.

- *Channel block index*: Indicates which channels are in this packet. A value of b indicates that this packet contains channels $512b$ to $512(b+1)$.
- *Accumulation ID*: A counter that represents the integration number. I.e., the first integration will have an ID 0, the second and ID 1, etc. These IDs should be referred to GPS time through knowledge of the system sampling rate and accumulation length parameters, and the system was last synchronized (see *sec-timing*).
- *Firmware version*: Bit [7] is always 0 for *Spectrometer* packets. The remaining bits contain a compile-time defined firmware version, represented in the form bit[6].bits[5:3].bits[2:0]. This document refers to firmware version 2.1.0.

The data payload in each packet should be interpretted as an 8192 byte array of 32 bit floats with dimensions `channel x polarization-product`. The `channel` index runs from 0 to 511. The `polarization-product` index runs from 0-3 with:

- index 0: XX product
- index 1: YY product
- index 2: *real* part of XY^* product
- index 3: *imag* part of XY^* product

2.3.2 Voltage Packets

The *Voltage* mode of the SNAP firmware outputs a continuous stream of voltage data, encapsulated in UDP packets. Each packet contains a data payload of 8192 bytes, made up of 16 time samples for up to 256 frequency channels of dual-polarization data:

```
#define N_t 16
#define N_p 2

struct voltage_packet {
  uint8_t version;
  uint8_t type;
  uint16_t n_chans;
  uint16_t chan;
  uint16_t feng_id
  uint64_t timestamp;
  complex4 data[n_chans, N_t, N_p] // 4-bit real + 4-bit imaginary
};
```

The header entries are all encoded network-endian and should be interpretted as follows:

- `version`; *Firmware version*: Bit [7] is always 1 for *Voltage* packets. The remaining bits contain a compile-time defined firmware version, represented in the form bit[6].bits[5:3].bits[2:0]. This document refers to firmware version 2.1.0.
- `type`; *Packet type*: Bit [0] is 1 if the axes of data payload are in order [slowest to fastest] channel x time x polarization. This is currently the only supported mode. Bit [1] is 0 if the data payload comprises 4+4 bit complex integers. This is currently the only supported mode.
- `n_chans`; *Number of Channels*: Indicates the number of frequency channels present in the payload of this data packet.
- `chan`; *Channel number*: The index of the first channel present in this packet. For example, a channel number `c` implies the packet contains channels `c` to `c + n_chans - 1`.
- `feng_id`; *Antenna ID*: A runtime configurable ID which uniquely associates a packet with a particular SNAP board.

- `timestamp`; *Sample number*: The index of the first time sample present in this packet. For example, a sample number s implies the packet contains samples s to $s + 15$. Sample number can be referred to GPS time through knowledge of the system sampling rate and accumulation length parameters, and the system was last synchronized. See *sec-timing*.

The data payload in each packet is determined by the number of frequency channels it contains. The maximum is 8192 bytes. If `type & 2 == 0` each byte of data should be interpretted as a 4-bit complex number (i.e. 4-bit real, 4-bit imaginary) with the most significant 4 bits of each byte representing the real part of the complex sample in signed 2's complement format, and the least significant 4 bits representing the imaginary part of the complex sample in 2's complement format.

If `type & 1 == 1` the complete payload is an array with dimensions `channel x time x polarization`, with

- `channel` index running from 0 to `n_chans`
- `time` index running from 0 to 15
- `polarization` index running from 0 to 1 with index 0 representing the X-polarization, and index 1 the Y-polarization.

2.4 Hardware Overview

A pair of analog inputs to the SNAP system are digitized with a CASPER-designed *ADC5g* ADC card [casper_adc5g] which hosts a single e2v EV8AQ160 chip [e2v]. The EV8AQ160 is a cost-effective 8-bit, quad-channel, 1250 Msps ADC, which can be configured to run as a dual-channel 2500 Msps, or single-channel 5000 Msps digitizer. The ATA system uses dual-channel configuration, with each ADC5g card processing dual polarization inputs from a single dish.

Digital signal processing is performed using a CASPER-designed SNAP board [casper_snap]. The SNAP is a simple FPGA-based processor, featuring a low-cost Kintex 7 FPGA, and a pair of SFP+ connectors each capable of supporting a 10~Gb Ethernet link. In addition to the FPGA, the SNAP hosts three on-board Hittite HMCAD1511 ADC chips [hmcad1511], which are not used in the ATA deployment owing to their relatively low sampling rate.

SNAPs and their ADCs are housed in a custom-designed 1U enclosure, shown in `fig-snap-box-front` and `fig-snap-box-internal`. This box also houses a Raspberry Pi 2 Model B [rpi] which provides the ability to remotely program and interact with a SNAP board.

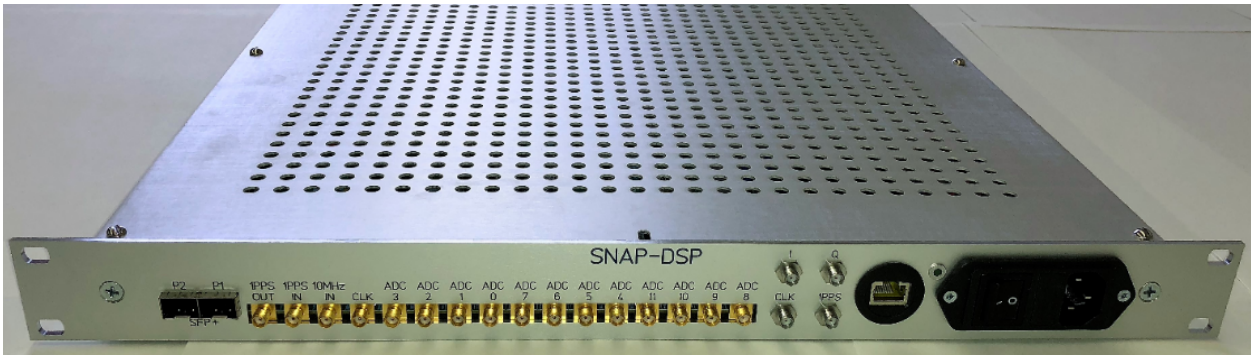

Fig. 2.2: The custom-designed 1U SNAP enclosure front panel. Interfaces (left to right): A pair of 10GbE-capable SFP+ ports; SMA input for a trigger signal; SMA output for a trigger signal; SMA input for onboard-ADC reference clock (not used at the ATA); SMA inputs for onboard-ADC sample clock (not used at the ATA); SMA inputs for 12 on-board ADC channels (not used at the ATA); four SMA inputs routed to the ADC5G card – dual RF inputs, a sampling clock, and a trigger signal; RJ45 1GbE interface routed to an internal Raspberry Pi single-board computer; 110-230V AC power.

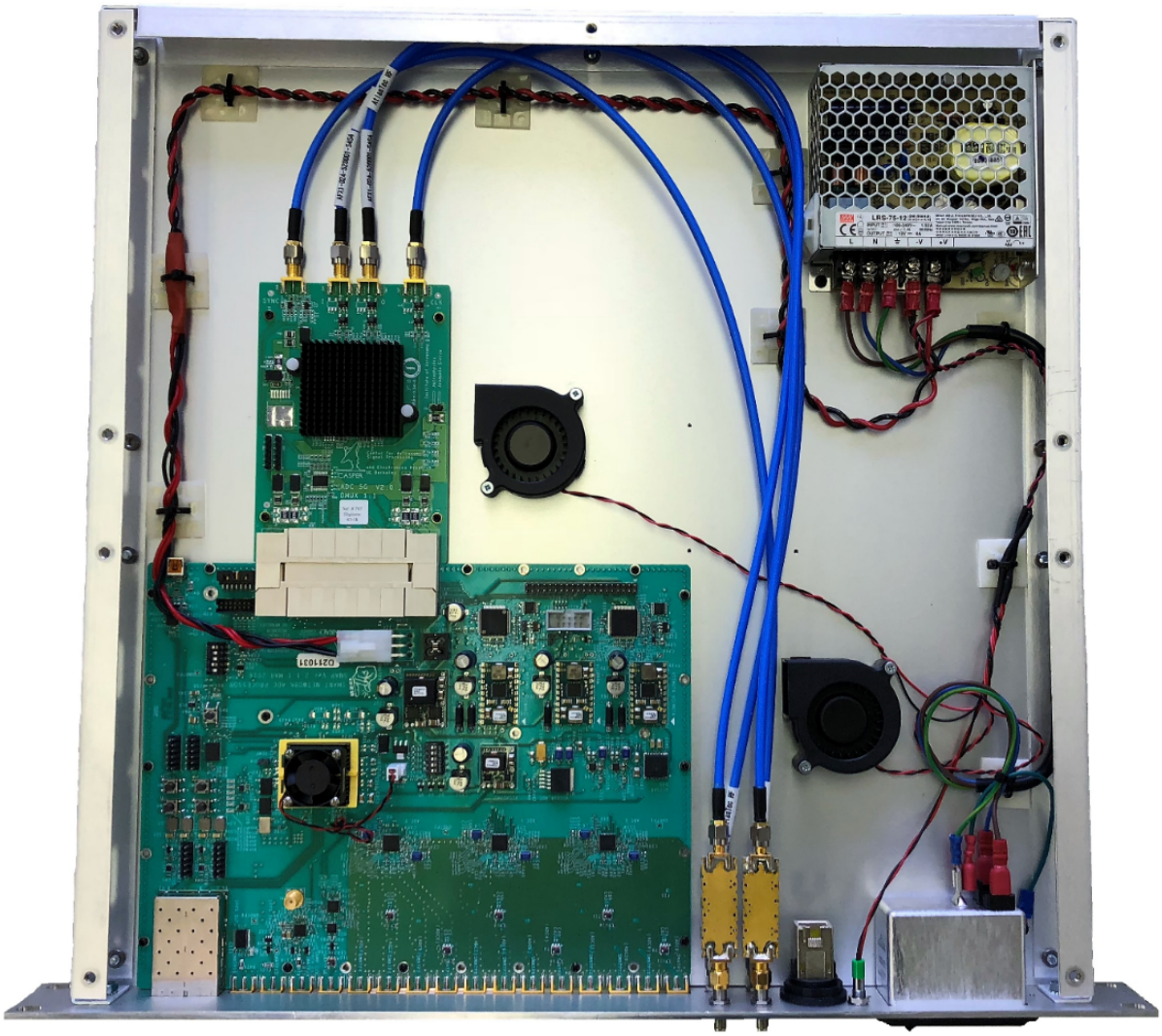

Fig. 2.3: Internal wiring of the ATA SNAP enclosure. Not shown is a Raspberry Pi control computer, whose mount holes are visible to the upper-right of the SNAP board. RF anti-aliasing filters are visible on the ADC analog RF inputs. These may be removed if upstream filtering is present in the system.

The complete ATA system, comprising multiple boards, should be driven by a common sampling clock, at a maximum rate of 2500~Msps. Each board should also be fed with a time-aligned trigger signal, which is used to synchronize the the capture of data by different boards in the system.

2.5 Firmware Overview

The firmware described in this document is designed in Mathwork's Simulink using CASPERs FPGA programming libraries. The Simulink source file is available on github.

2.5.1 Building the Simulink Model

The SNAP firmware model (`snap_adc5g_feng_rpi.slx`) was built with the following software stack. Use other versions at your peril.

- Ubuntu 18.04 64-bit
- MATLAB/Simulink 2019a, including Fixed Point Designer Toolbox
- Xilinx Vivado System Edition 2019.1.3
- `mlib_devel` (version controlled within the `ata_snap` repository.

To obtain and open the Simulink model:

```
# Clone the firmware repository
git clone https://github.com/realtimeradio/ata_snap

# Clone relevant sub-repositories
cd ata_snap
git submodule init
git submodule update

# Install the mlib_devel dependencies
# You may want to install these in a Python virtual environment
cd mlib_devel
pip install -r requirements.txt
```

Next, create a local environment specification file in the `ata_snap` directory, named `startsg.local`. An example environment file is:

```
#!/bin/bash
####### User to edit these accordingly ######
export XILINX_PATH=/data/Xilinx/Vivado/2019.1
export MATLAB_PATH=/data/MATLAB/R2019a
# PLATFORM lin64 means 64-bit Linux
export PLATFORM=lin64
# Location of your Xilinx license
export XILINXD_LICENSE_FILE=/home/jackh/.Xilinx/Xilinx.lic

# Library tweaks
export LD_PRELOAD=${LD_PRELOAD}:"/usr/lib/x86_64-linux-gnu/libexpat.so"
# An optional python virtual environment to activate on start
export CASPER_PYTHON_VENV_ON_START=/home/jackh/casper-python3-venv
```

You should edit the paths accordingly.

To open the firmware model, from the top-level of the repository (i.e. the `ata_snap` directory) run `./startsg`. This will open MATLAB with appropriate libraries, at which point you can open the `snap_adc5g_feng_rpi.slx` Simulink model.

2.5.2 Firmware Overview

The Simulink source code for the SNAP firmware is shown in `fig-simulink-diagram-labeled`. A pictorial representation with annotated data path bit widths is shown in `fig-ata-snap-feng`.

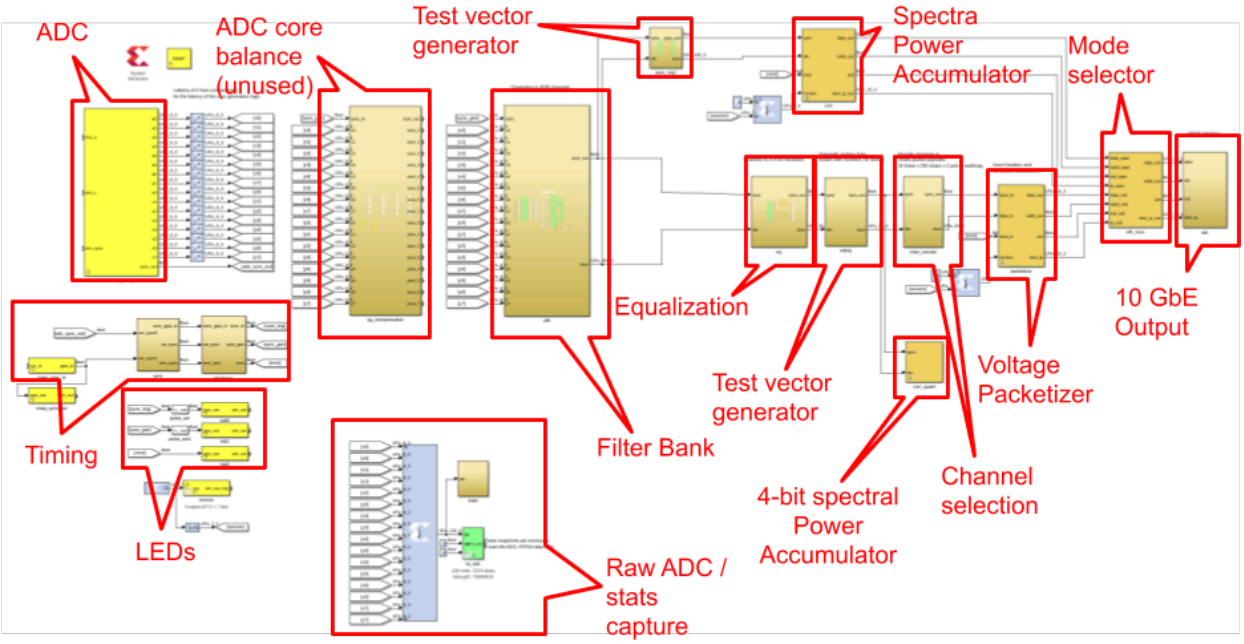

Fig. 2.4: The `snap_adc5g_feng_rpi.slx` Simulink design, with annotations.

In the remainder of this section an overview of the functional modules in the system is given.

2.5.3 Module Descriptions

Here basic explanations of the functionality of the different firmware processing modules is given. Where modules can be controlled or monitored at runtime, software routines to do so are described in *Run-time Control*.

2.5.3.1 ADC

The ADC module encapsulates an interface to an e2v EV8AQ160 [e2v] ADC chip. This chip has four independent ADC cores which can each run at up to 1250 Msps. In the ATA firmware, these are configured as a pair of 2500 Msps samplers.

On power-up, and after reprogramming the FPGA, the ADC interface must be initialized. Initialization ensures that the ADC is in the correct operating mode, and that the ADC and FPGA link is appropriately trained. Training ensures that the digital data transmitted from the ADC is successfully received by the FPGA, and involves tweaking the FPGA-side capture clock phase for reliable data reception.

To ensure that the ADCs are correctly configured when programming the SNAP boards, the boards should always be loaded with firmware using the `program` method (see *Run-time Control*).

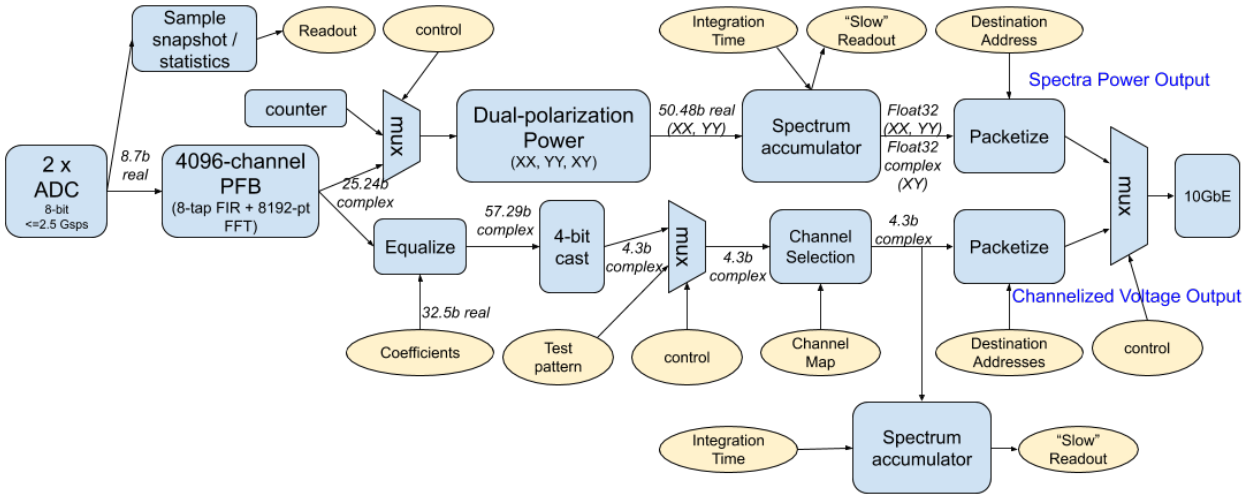

Fig. 2.5: A pictorial representation of the SNAP firmware, showing major processing modules and the bit widths of their data paths. Yellow circles in this diagram represent the runtime-controllable elements of the pipeline.

TODO: Add software method and instructions for inter-core mismatch calibration.

Relevant Software Methods

- `program`: Load new firmware onto a SNAP and intialize the ADCs
- `adc_initialize`: Perform a standalone initialization of the ADCs
- `adc_get_samples`: Get a snapshot of raw ADC samples
- `adc_get_stats`: Get the mean, mean power, and number of clipping events from the last 512k samples

2.5.3.2 Timing

The Timing module allows multiple SNAP boards to be synchronized, and locks data timestamps to a known UTC origin. Mulitple board synchronization relies on each SNAP board being fed a time-aligned, distributed pulse, with an edge rate of << 1ms. Alignment of timestamps to UTC requires that the SNAP pulses have a positive edge aligned with a UTC second.

Typically, both of the above requirements can be met by using a syncronization signal which is a distributed GPS-locked Pulse-Per-Second (PPS).

Quality of board synchronization is determined by the nature of the PPS distribution system. For commercial PPS distribution equipment, using length-matched cables, synchronization will be within < 10 ADC samples.

The synchronization process is as follows:

1. Wait for a PPS to pass
2. Arm all SNAP boards in the system to trigger on the next PPS edge
3. Reset on-board spectra counters on the next PPS edge

Relevant Software Methods

- `sync_wait_for_pps`: Wait for an external PPS pulse to pass
- `sync_arm`: Arm the firmware such that the next PPS results in a reset of local counters
- `sync_get_last_sync_time`: Return the time that the firmware was last synchronized to a PPS pulse
- `sync_get_ext_count`: Return the number of PPS pulses returned since the FPGA was last programmed
- `sync_get_fpga_clk_pps_interval`: Return the number of FPGA clock ticks between the last two PPS pulses
- `sync_get_adc_clk_freq`: Infer the ADC clock rate from the number of FPGA clock ticks between PPS pulses

2.5.3.3 Filter Bank

The Filter Bank (aka Polyphase Filter Bank, PFB [pfb]) separates the X- and Y-polarization input broad-band data streams into 4096 frequency channels, starting at DC, with centers separated by $\frac{f_s}{4096}$. These baseband frequencies, from DC to $\frac{f_s}{2}$ represent different sky-frequencies when the input analog signals are mixed with LOs upstream of the digital system.

As shown in `fig-ata-snap-feng`, the PFB receives real-valued broad-band data with 8-bits resolution, and after processing delivers complex-valued spectra with 25-bit resolution.

Internally, the FFT data path is shown in `fig-pfb-bitwidth`.

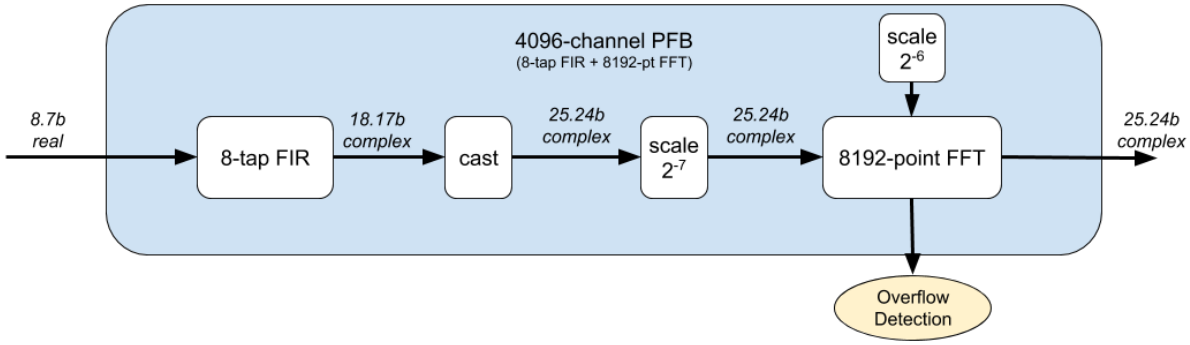

Fig. 2.6: A pictorial representation of the PFB internal datapath, showing internal data precision. Coefficients in the FIR and FFT modules are stored with 18 bits resolution. Overall signal amplitude growth in the FFT is controlled with a *shift schedule*, which is hardcoded to enforce 2^{-6} scaling. This is sufficient to guarantee against FFT overflow for any input signal.

In general, an FFT with 2^N points has N butterfly stages, and dynamic range should grow by N bits to guarantee against overflow.

The SNAP firmware processes 18-bit inputs with 25-bits of precision, allowing for 7 bits of growth. A further scaling down of 2^6 using the FFT shift schedule is sufficient to guarantee against overflow.

Relevant Software Methods

- `fft_of_detect`: Count overflows in the FFT module
- `spec_read`: Read an accumulated spectrum
- `spec_plot`: Plot an accumulated spectrum

2.5.4 Spectral Power Accumulator

The Spectral Power Accumulator generates power-spectra for the two input data streams.

First, the auto- and cross- power of the two input data streams are computed. The former are real-valued, while the latter is complex.

All are computed by:

1. Multiplying 25-bit voltage inputs to generate 50-bit powers
2. Integrating these powers into 64-bit accumulators, with a runtime-configurable integration length
3. Casting data to 32-bit floating point, and outputting over 10GbE. 64-bit integer values are available via a low speed 1GbE interface.

The nature of the bit handling in this implementation means that the accumulators can only guarantee against overflow for integrations of fewer than 2^{14} spectra. This amounts to just 53 ms of data. In practice, except in cases of high-power narrowband inputs, integrations of substantially longer without overflow are possible. In the event of overflow, data are saturated at $\pm 2^{63}$.

Spectra can be read from the power accumulator via software – appropriate for monitoring, debugging, and low time-resolution (~1s) observations – or can be streamed out of the SNAP's 10 GbE interface. The latter interface supports integration lengths of down to 150 micro-seconds, and emits data in the format descibed in sec-spec-format.

A test vector injection module exists for the purposes of testing the spectrometer pipeline. When activated, this module replaces PFB data with a test pattern whose real parts are zero, and whose imaginary parts form a counter. For the X-polariaztion, FFT channel i takes the value $32 * (8floor(\frac{i}{4}) + mod(i, 4))$. For the Y-polariaztion, FFT channel i takes the value $32 * (8floor(\frac{i}{4}) + mod(i, 4)) + 4$.

Scaling in the spectrometer is such that the test vector inputs appear at the accumulator input with FFT channel i having the values:

- For the X-polarization: $8 * floor(\frac{i}{4}) + mod(i, 4)$.
- For the Y-polarization: $8 * floor(\frac{i}{4}) + mod(i, 4) + 4$.

This pattern can be checked against observed output data to verify that the two polarizations are being correctly identified, and accumulation length is being correctly set.

- `set_accumulation_length`: Set the number of spectra to be accumulated
- `spec_read`: Read an accumulated spectrum
- `spec_plot`: Plot an accumulated spectrum
- `eth_set_mode`: Choose between spectrometer and voltage 10GbE outputs
- `eth_set_dest_port`: Set the destination UDP port for 10GbE traffic
- `spec_set_destination`: Set the destination IP address for spectrometer packets
- `spec_test_vector_mode`: Turn on and off the spectrometer test-vector mode

2.5.5 Equalization

In the voltage pipeline, post-PFB data are quantized to 4 bits prior to being transmitted over 10 GbE.

This substantial reduction of bit precision requires carefully managing input signal levels. As such, prior to equalization, spectra are multiplied by a runtime-programmable amplitude scaling factor. The scaling factors can be uniquely specified per polarization and per frequency channel, and should be used to ensure that data in each frequency channel exhibit an appropriate RMS.

Equalization coefficients can be computed either by inferring power levels from Spectrometer Mode data, or by directly interrogating the post-quantization power-levels using the dedicated 4-bit spectrometer. In this latter case, dividing out the accumulation length will yield the mean power in each 4-bit signal after scaling. Scaling coefficients can then be modified as required in order to target an optimal level (for example, that given by [quant]).

- `eq_load_coeffs`: Load a set of equalization coefficients
- `quant_spec_read`: Get power spectra from post-quantization data

2.5.6 Voltage Channel Selection

The Voltage Mode output data path requires that only 2048 of the 4096 generated frequency channels are transmitted over 10 GbE. This down-selection takes place following 4-bit quantization, and is typically configured at initialization time using an appropriate configuration file (see *sec-config-file*).

The chosen 2048 channels are split into eight groups of 256 channels, each of which may be directed to a different IP address.

Channel selection should satisfy the following rules:

1. Each destination IP address should receive a start channel which is an integer multiple of 8.
2. If channel n is sent to IP address I, this IP address should also receive channels $n+1, n+2, n+3, n+4, n+5, n+6, n+7$.

Beyond these requirements, channels selected for transmission need not be contiguous and may also be duplicated. I.e. a block of 512 channels may be sent to each of 4 IP addresses.

- `select_output_channels`: Select which frequency channels are to be sent over 10 GbE
- `eq_load_test_vectors`: Load a custom set of test vectors into the voltage data path
- `eq_test_vector_mode`: Turn on or off voltage test vector injection

2.5.7 10Gb Ethernet

- `eth_reset`: Reset the Ethernet core
- `eth_set_mode`: Choose between spectrometer and voltage 10GbE outputs
- `eth_set_dest_port`: Set the destination UDP port for 10GbE traffic
- `eth_enable_output`: Turn on 10GbE transmission
- `eth_print_counters`: Print Ethernet packet statistics

2.6 Run-time Control

2.6.1 Installing the Control Library

The *ata_snap* python library is provided to control the F-Engine firmware design. It requires Python >=3.5 and the following custom python libraries:

1. casperfpga (a control library for interacting with CASPER hardware, such as SNAP boards)
2. adc5g (a library for configuring the ADC5G card)

These libraries are bundled in the *ata_snap* repository to minimize issues with library version compatibility.

To install the control libraries:

```
# Clone the ata_snap repository
git clone https://github.com/realtimeradio/ata_snap

# Clone relevant sub-repositories
cd ata_snap
git submodule init
git submodule update

# Install casperfpga
cd sw/casperfpga
# Install casperfpga dependencies (requires pip, which can be installed with `apt␣
↪install python3-pip`
pip install -r requirements.txt
# Install casperfpga
python setup.py install

# Install the adc5g library
cd ../adc5g/adc5g
python setup.py install

# Install the ata_snap library
cd ../../../ata_snap
python setup.py install
```

If the library has installed correctly, in a Python shell, you should be able to successfully execute

```
from ata_snap import ata_snap_fengine
```

2.7 Configuration Recipes

Simple use of the *ata_snap* library comprises the following steps:

1. Program the SNAP boards with appropriate firmware
2. Configure any runtime settings
3. Synchronize SNAP boards with UTC
4. Turn on data flow

Once these steps are complete, data will be transmitted over Ethernet and downstream software can catch and process this data as desired.

For a single SNAP board, all of these steps can be carried out at once using a provided intialization scrips `snap_feng_init.py`. This script has the following use template:

```
$ snap_feng_init.py -h
usage: snap_feng_init.py [-h] [-s] [-t] [--eth_spec] [--eth_volt] [-a ACCLEN]
                         [-f FFTSHIFT] [--specdest SPECDEST]
                         host fpgfile configfile

Program and initialize a SNAP ADC5G spectrometer

positional arguments:
  host                  Hostname / IP of SNAP
  fpgfile               .fpgfile to program
  configfile            Configuration file

optional arguments:
  -h, --help            show this help message and exit
  -s                    Use this flag to re-arm the design's sync logic
                        (default: False)
  -t                    Use this flag to switch to post-fft test vector outputs
                        (default: False)
  --eth_spec            Use this flag to switch on Ethernet transmission of the
                        spectrometer (default: False)
  --eth_volt            Use this flag to switch on Ethernet transmission of
                        F-engine data (default: False)
  -a ACCLEN             Number of spectra to accumulate per spectrometer dump.
                        Default: get from config file (default: None)
  --specdest SPECDEST   Destination IP address to which spectra should be sent.
                        Default: get from config file (default: None)
```

2.7.1 Configuration File

```
# Accumulation length, in spectra.
acclen: 300000
# Coeffs should be a single number, or an array
# of 4096 numbers to set one coefficient per channel.
coeffs: 100
# UDP port for 10GbE data
dest_port: 10000
spectrometer_dest: 10.11.10.173
# Define which channels should be output
# over 10GbE in voltage dump mode.
voltage_output:
  start_chan: 0
  n_chans: 1024
  # Channels will be spread over the following
  # destinations so that the first n_chans // len(dests)
  # go to the first IP address, etc.
  dests:
      - 10.11.10.173
# All relevant IP/MAC mapping should be manually
# specified here
arp:
  10.11.10.173: 0xaeecc7b400ff
  10.11.10.174: 0xaeecc7b400a0
```

CHAPTER

THREE

ATA_SNAP SNAP API REFERENCE

class ata_snap.ata_snap_fengine.**AtaSnapFengine**(*host*, *feng_id=0*, *transport=<class 'casperfpga.transport_tapcp.TapcpTransport'>*, *use_rpi=None*)

Bases: `object`

This is a class which implements methods for programming and communicating with a SNAP board running the ATA F-Engine firmware.

Parameters

- **host** (*str*) – Hostname of SNAP board associated with this instance.
- **feng_id** (*int*) – Antenna ID of the antenna connected to this SNAP. This value is used in the output headers of data packets.
- **transport** (*casperfpga.Transport*) – The type of connection the SNAP supports. Should be either casperfpga.TapcpTransport (if communicating over 10GbE) or casperfpga.KatcpTransport (if communicating via a Raspberry Pi)
- **use_rpi** (*Bool*) – If set, override the `transport` parameter. If True, use the KatcpTransport to talk to a SNAP's Raspberry Pi. If False, use the TapcpTransport.

adc_balance()

Attempt to balance the ADC cores offset and gains, using the current input signal.

Returns offset, gain, phase The currently loaded offset and gain setting. Each is a vector with 4 elements - one per ADC core.

adc_get_mismatch(*n_snapshot=16*)

Get the offset and gain mismatch between the two interleaved cores forming each polarization.

Parameters

- **pol** (*int*) – Polarization to interrogate (0 or 1)
- **n_snapshot** (*int*) – Number of snapshots to take. More gives better statistics, but takes longer

Returns

2x2 array [[pol0 offset, pol0 gain], [pol1 offset, pol1 gain]] offset: the number of ADC counts which core 1 is is offset by relative to core 0. E.g. if offset=0.2, then the mean of all the core 1 ADC samples is 0.2 higher than the mean of all core 0 samples.

gain: the relative gain of core 1 vs core 0. E.g., if gain = 1.03, then the mean power of all core 1 samples is 1.03 times the mean power of all core 1 samples.

Return type (float, float)

adc_get_samples()
Get a block of samples from both ADC inputs, captured simultaneously.

This method requires that the currently programmed fpg file is known. This can be achieved either by programming the board with program(<fpgfile>), or by running fpga.get_system_information(<fpgfile>) if the board was already programmed outside of this class.

Returns x, y (numpy arrays of ADC sample values)

Return type numpy.ndarray

adc_get_stats(*per_core=False*)
Get the mean value and power, and a count of overflow events for the system's ADCs. Statistics are calculated over the last 512k samples, and are computed from samples obtained from all ADC channels simultaneously.

Parameters **per_core** (*bool*) – If True, return stats for each ADC core. If false, return stats for each ADC channel.

Returns A 3-tuple of numpy.ndarray, with either 4 entries (if per_core=True) or 2 entries (if per_core=False). The tuple is (clip_count, mean, mean_power) where clip_count is the number of clipping events in the last 512k samples, mean is the average of the last 512k samples, and mean_power is the mean power of the last 512k samples. If per_core=True, each array has 4 entries, representing the four ADC cores. Cores 0, 2 digitize the X-pol RF input, and cores 1, 3 digitize the Y-pol input. In this case stats are calculated over 256k samples per core. If per_core=False, each array has 2 entries, with the first address the X-pol RF input, and the second the Y-pol. In this case statistics are calculated over 512k samples per polarization.

Return type (numpy.ndarray, np.ndarray, np.ndarray)

adc_initialize()
Initialize the ADC interface by performing FPGA<->ADC link training. Put the ADC chip in dual-input mode. This method must be called after programming a SNAP, and is called automatically if using this class's *program* method with init_adc=True.

change_feng_id(*feng_id*)
Update the Fengine ID field of both spectrometer packets and voltage packets, without otherwise effecting the configuration of a running board.

Also update this instance's *feng_id* attribute

Parameters **feng_id** (*int*) – New F-Engine ID for this instance

eq_balance(*pol*, *target_rms=0.125*, *cutoff=2.0*, *medfil_ksize=401*, *conv_ksize=50*)
Tweak the EQ coefficients for a polarization to target a particular post-EQ RMS.

Parameters

- **pol** (*int*) – Polarization to equalize.
- **target_rms** (*float*) – Target RMS. This is specified in units of 1 8-bit post-equalization least significant bit. I.e., target_rms=1 will target an RMS of 1 on a scale where 8-bit post-EQ values run from -255 to 255. This parameter is always relative to 8-bit values. So, for 4-bit quanitization, target_rms = 2**4 will target an RMS of one 4-bit LSB.
- **cutoff** (*float*) – The scale, relative to the mean coefficient, at which coeffiecients are saturated. For example, if the mean coefficient is 1000, and `cutoff` is 2, coefficients will saturated at a value of 2000. This allows the bandpass to still be visible in the equalized data.

- **medfil_ksize** (*int*) – Size of median filter kernel to use for RFI removal. Should be odd.
- **conv_ksize** (*int*) – Convolution kernel size to use for bandpass smoothing.

Returns The loaded coefficients as read back. Note that these will be slightly different to the loaded coefficients since they will have been saturated and quantized to the specifications of the firmware.

Return type Integer coefficients as returned by `eq_read_coeffs`

eq_compute_coeffs(*target_rms=0.5*, *medfil_ksize=401*, *conv_ksize=100*, *acc_len=50000*)

Get appropriate EQ coefficients to scale FFT output for an appropriate post-quantization power target. Do this by grabbing a full bit-precision spectrum, filtering out RFI and smoothing, and then computing the scaling required to reach a target power level.

Parameters

- **target_rms** (*float*) – The target voltage RMS. This should be specified relative to signed data normalized to the range +/-1. I.e., a target_rms of 1./2**7 represents an RMS of one least-significant bit after quantizing to 8-bits. A target_rms of 1./2**3 represents one least-significant bit after quantizing to 4-bits.
- **medfil_ksize** (*int*) – Size of median filter kernel to use for RFI removal. Should be odd.
- **conv_ksize** (*int*) – Convolution kernel size to use for bandpass smoothing.
- **acc_len** (*int*) – If specified, use this accumulation length for the computation. Accumulation length should be sufficiently long to obtain good autocorrelation SNR. After the EQ coeffients are obtained, the accumulation length the firmware was using before this function was invoked is reloaded.

Returns x_coeffs, y_coeffs: A tuple of coefficients. Each is a numpy array of floating-point values.

Return type numpy.array, numpy.array

eq_load_coeffs(*pol*, *coeffs*)

Load coefficients with which to multiply data prior to 4-bit quantization. Coefficients are rounded and saturated such that they fall in the range (0, 2048), with a precision of 2**-5 = 0.03125. A single coefficient can be provided, in which case coefficients for all frequency channels will be set to this value. If an array or list of coefficients are provided, there should be one coefficient per frequency channel in the firmware pipeline.

Parameters

- **pol** (*int*) – Selects which polarization vectors are being loaded to (0 or 1) 0 is the first ADC input, 1 is the second.
- **coeffs** (*float, or list / numpy.ndarray*) – The coefficients to load. If *coeffs* is a single number, this value is loaded as the coefficient for all frequency channels. If *coeffs* is an array or list, it should have length self.n_chans_f / self.n_coeff_shared or length self.n_chans_f. If the latter, the coefficients will be decimated by a factor of self.n_coeff_shared. Element [i] of this vector is the coefficient applied to channels i through i+self.n_coeff_shared-1 if the array has length self.n_chans_f / self.n_coeff_shared. If the array has length self.n_chans_f then element [self.n_coeff_shared*i] is the coefficient which will be applied to channels i through i+self.n_coeff_shared-1. Coefficients are quantized to UFix32_5 precision.

Raises **AssertionError** – If an array of coefficients is provided with an invalid size, if any coefficients are negative, or if pol is a non-allowed value

Returns 2-tuple coeffs, bin_pt. coeffs: An array of self.n_chans_f indicating the integer coefficients loaded. bin_pt: position of binary point with which firmware interprets coefficients. Eg: bin_pt=5 means that coefficients are interpreted as coeffs / 2^5

Return type numpy.ndarray, int

eq_load_test_vectors(*pol*, *tv*)
Load test vectors for the Voltage pipeline test vector injection module.

Parameters

- **pol** (*int*) – Selects which polarization vectors are being loaded to (0 or 1) 0 is the first ADC input, 1 is the second.
- **tv** (*numpy.ndarray or list of ints*) – Test vectors to be loaded. *tv* should have self.n_chans_f elements. tv[i] is the test value for channel i. Each value should be an 8-bit number - the most-significant 4 bits are interpretted as the 4-bit, signed, real part of the data stream. The least-significant 4 bits are interpretted as the 4-bit, signed, imaginary part of the data stream.

Raises **AssertionError** – If an array of test vectors is provided with an invalid size, or if pol is a non-allowed value

eq_read_coeffs(*pol*, *return_float=False*)
Read currently loaded coefficients for one polarization.

Parameters

- **pol** (*int*) – Selects which polarization vectors are being read to (0 or 1) 0 is the first ADC input, 1 is the second.
- **return_float** (*Bool*) – If True, return floating-point coefficients as interpretted by the SNAP If False, return integer coefficients and a binary point position.

Returns If return_float: coeffs; If not return_float: 2-tuple coeffs, bin_pt. coeffs: An array of self.n_chans_f indicating the coefficients loaded. bin_pt: If return_float=False, this is the position of binary point with which firmware interprets coefficients. Eg: bin_pt=5 means that coefficients are interpreted as coeffs / 2^5

Return type numpy.ndarray or numpy.ndarray, int

eq_test_vector_mode(*enable*)
Turn on or off the test vector mode downstream of the 4-bit quantizers. This mode can be used to replace the FFT output in the voltage data path with an arbitrary pattern which can be set with *eq_load_test_vectors*

Parameters **enable** (*bool*) – True to turn on the test mode, False to turn off

eth_enable_output(*enable=True*, *interface='all'*)
Enable the 10GbE output datastream. Only do this after appropriately setting an output configuration and setting the pipeline mode with *eth_set_mode*. For spectra mode, prior to enabling Ethernet the destination address should be set with *spec_set_destination*. For voltage mode, prior to enabling Ethernet configuration should be loaded with *select_output_channels*

Parameters

- **enable** (*bool*) – Set to True to enable Ethernet transmission, or False to disable.
- **interface** (*integer or 'all'*) – Which physical interface to enable / disable.

eth_print_counters()
Print ethernet statistics counters from all ethernet cores. This is a simple wrapper around casperfpgas gbes.read_counters() method.

eth_reset(*interface='all'*)
Reset the Ethernet core. This method will clear the reset after asserting, and will leave the transmission stream disabled. Reactivate the Ethernet core with *eth_enable_output*

Parameters **interface** (*integer or 'all'*) – Which physical interface to enable / disable.

eth_set_dest_port(*port*, *interface='all'*)
Set the destination UDP port for output 10GbE packets.

Parameters

- **port** (*int*) – UDP port to which traffic should be sent.
- **interface** (*integer or 'all'*) – Which physical interface to manipulate

eth_set_mode(*mode='voltage'*)
Set the 10GbE output stream to either "voltage" or "spectra" mode. To prevent undesired behaviour, this method will disable Ethernet transmission prior to switching. Transmission should be re-enabled if desired using *eth_enable_output*.

Parameters **mode** (*str*) – "voltage" or "spectra"

Raises **AssertionError** – If mode is not an allowed value

fft_of_detect()
Read the FFT overflow detection register. Will return True if an overflow has been detected in the last accumulation period. False otherwise. Increase the FFT shift schedule to avoid persistent overflows.

Returns True if FFT overflowed in the last accumulation period, False otherwise.

Return type bool

get_accumulation_length()
Get the number of spectra currently being accumulated in the on-board spectrometers.

Returns acclen; Accumulation length

Rtype int

get_delay(*pol*)
Get the currently loaded delay, in units of ADC clock cycles, of a pipeline input.

Parameters **pol** (*int*) – Polarization to read (0 or 1)

Returns delay, in units of ADC clock cycles

Return type int

get_pending_delay(*pol*)
Get the upcoming delay, in units of ADC clock cycles, of a pipeline input, and the time when it will be loaded.

Parameters **pol** (*int*) – Polarization to read (0 or 1)

Returns (delay, time_to_load) tuple delay, in units of ADC clock cycles time_to_load, in units of seconds. If negative, represents the number of seconds since the delays were loaded.

Return type (int, float)

is_programmed()
Returns True if the fpga appears to be programmed with a valid F-engine design. Returns False otherwise. The test this method uses is searching for the register named *version* in the running firmware, so it can easily be fooled.

Returns True if programmed, False otherwise

Return type bool

program(*fpgfile*, *force=False*, *init_adc=True*)
Program a SNAP with a new firmware file.

Parameters

- **fpgfile** (*str*) – .fpg file containing firmware to be programmed
- **force** (*bool*) – If True, overwrite the existing firmware even if the firmware to load appears to already be present. This only makes a difference for *TapcpTransport* connections.
- **init_adc** (*bool*) – If True, initialize the ADC cards after programming. If False, you *must* do this manually before using the firmware using the *adc_initialize* method.

quant_spec_read(*pol=0*, *flush=True*, *normalize=False*)
Read a single accumulated spectrum of the 4-bit quantized data

This method requires that the currently programmed fpg file is known. This can be achieved either by programming the board with program(<fpgfile>), or by running fpga.get_system_information(<fpgfile>) if the board was already programmed outside of this class.

Parameters

- **pol** (*int*) – Polarization to read
- **flush** (*Bool*) – If True, throw away one integration prior to getting data. This can be desirable if (eg) EQ coefficients have been recently changed.
- **normalize** (*Bool*) – If True, divide out the accumulation length and firmware scaling, returning floating point values. Otherwise, return integers and leave these factors present.

Raises **AssertionError** – if pol is not 0 or 1

Returns A numpy array containing the polarization's power spectrum

Return type numpy.array

select_output_channels(*start_chan*, *n_chans*, *dests=['0.0.0.0']*, *n_interfaces=None*, *n_bits=4*)
Select the range of channels which the voltage pipeline should output.

Example usage:

Send channels 0..255 to 10.0.0.1: select_output_channels(0, 256, dests=['10.0.0.1'])

Send channels 0..255 to 10.0.0.1, and 256..512 to 10.0.0.2 select_output_channels(0, 512, dests=['10.0.0.1', '10.0.0.2'])

Parameters

- **start_chan** (*int*) – First channel to output
- **n_chans** (*int*) – Number of channels to output
- **dests** (*list of str*) – List of IP address strings to which data should be sent. The first n_chans / len(dests) will be sent to dest[0], etc..
- **n_interfaces** – Number of 10GbE interfaces to use. Should be <= self.n_interfaces Default to using all available interfaces.

Raises **AssertionError** – If the following conditions aren't met: *start_chan* should be a multiple of self.n_chans_per_block (4) *n_chans* should be a multiple of self.n_chans_per_block (4) *interface* should be <= self.n_interfaces

Returns A dictionary, keyed by destination IP, with values corresponding to the ranges of channels destined for this IP. Eg, if sending 512 channels over two destinations '10.0.0.10' and '10.0.0.11', this function might return {'10.0.0.10': [0,1,..,255], '10.0.0.11': [256,257,..,511]}

Return type dict

set_accumulation_length(*acclen*)
Set the number of spectra to accumulate for the on-board spectrometer.

Parameters **acclen** (*int*) – Number of spectra to accumulate

set_delays(*delays*, *load_time=-1*, *clock_rate_hz=2048000000*, *sync_time=None*)
Set the delay, in units of ADC clock cycles, of a pipeline input.

Parameters

- **delay** (*float*) – tuple of delays to apply to a polarization pair [x-pol delay, y-pol delay]
- **load_time** (*int*) – UNIX time at which delays should be loaded, or -1 to load immediately
- **clock_rate_hz** (*int*) – ADC clock rate in Hz. If None, the clock rate will be computed from the observed PPS interval, which could fail if the PPS is unstable or not present.
- **sync_time** (*int*) – The time, in UNIX seconds, at which the F-engine was last synchronized. If None, the sync time will be queried from the board using the *get_last_sync_time()* method.

Returns None, unless load_time is provided. In this case return the spectrum ID (the spectrum count, relative to the F-engine sync time) at which the requested delays will be loaded

Rval int

spec_plot(*mode='auto'*)
Plot an accumulated spectrum using the matplotlib library. Frequency axis is infered from the ADC clock frequency detected with *sync_get_adc_clk_freq*.

Parameters **mode** (*str:*) – "auto" to plot a power spectrum from the X and Y polarizations on separate subplots. "cross" to plot the magnitude and phase of a complex-valued cross-power spectrum of the two polarizations.

Raises **AssertionError** – if mode is not "auto" or "cross"

spec_read(*mode='auto'*, *flush=False*, *normalize=False*)
Read a single accumulated spectrum.

This method requires that the currently programmed fpg file is known. This can be achieved either by programming the board with program(<fpgfile>), or by running fpga.get_system_information(<fpgfile>) if the board was already programmed outside of this class.

Parameters

- **mode** (*str:*) – "auto" to read an autocorrelation for each of the X and Y pols. "cross" to read a cross-correlation of Xconj(Y).
- **flush** (*Bool*) – If True, throw away one integration prior to getting data. This can be desirable if (eg) EQ coefficients have been recently changed.
- **normalize** (*Bool*) – If True, divide out the accumulation length and firmware scaling, returning floating point values. Otherwise, return integers and leave these factors present.

Raises **AssertionError** – if mode is not "auto" or "cross"

Returns If mode="auto": A tuple of two numpy arrays, xx, yy, containing a power spectrum from the X and Y polarizations. If mode="cross": A complex numpy array containing the cross-power spectrum of Xconj(Y).

Return type numpy.array

spec_set_destination(*dest_ip*)
Set the destination IP address for spectrometer packets.

Parameters **dest_ip** (*str*) – Destination IP address. E.g. "10.0.0.1"

spec_test_vector_mode(*enable*)
Turn on or off the test vector mode in the spectrometer data path. This mode replaces the FFT output in the data path with 12 bit counter which occupies the most significant bits of the imaginary part of the FFT output. I.e. when enabled, the imaginary part of each spectrum is a ramp from 0..4095 / 2**17

Parameters **enable** (*bool*) – True to turn on the test mode, False to turn off

sync_arm(*manual_trigger=False*)
Arm the FPGA's sync generators for triggering on a subsequent PPS. The arming proceedure is the following: 1. Wait for a PPS to pass using *sync_wait_for_pps*. 2. Arm the FPGA's sync register to trigger on the next+2 PPS 3. Compute the time this PPS is expected, based on this computer's clock. 4. Write this time as a 32-bit UNIX time integer to the FPGA, to record the sync event.

Parameters **manual_trigger** (*bool*) – Use a software sync, rather than relying on an external PPS pulse. See *sync_manual_trigger* for more information.

Returns Sync trigger time, in UNIX format

Rval int

sync_get_adc_clk_freq()
Infer the ADC clock period by counting FPGA clock ticks between PPS events.

Returns ADC clock rate in MHz

Rval float

sync_get_ext_count()
Read the number of external sync pulses which have been received since the board was last programmed.

Returns Number of sync pulses received

Rval int

sync_get_fpga_clk_pps_interval()
Read the number of FPGA clock ticks between the last two external sync pulses.

Returns FPGA clock ticks

Rval int

sync_get_last_sync_time()
Get the sync time currently stored on this FPGA

Returns Sync trigger time, in UNIX format

Rval int

sync_manual_trigger()
Issue a sync using the F-engine's built-in software trigger. This can be useful for single board deployments or testing where an external PPS signal may not be available. However, a manual sync cannot synchronize multiple boards, and will only be aligned to the expected sync time at the ~1s level.

sync_select_input(*pps_source*)
Select which PPS input is used to drive the design's timing subsystem.

Parameters **pps_source** (*str*) – Which PPS source should be used. "adc" indicates the ADC5G's sync input. "board" indicates the SNAP board's dedicated sync input.

sync_wait_for_pps()
Block until an external PPS trigger has passed. I.e., poll the FPGA's PPS count register and do not return until is changes.

CHAPTER

FOUR

RFSOC FIRMWARE SPECIFICATIONS

4.1 Requirements

4.1.1 Inputs

The F-engine firmware shall channelize 16 independent ADC streams (assumed to be 8 independent, dual-polarization ATA IFs).

4.1.2 ADC Sample Rate

The F-Engine firmware shall pass timing analysis for a maximum ADC sample rate of 2048 Msps.

4.1.3 Frequency Channels

The F-Engine firmware shall internally generate at least 1024 complex-valued frequency channels over the digitized Nyquist band. For an ADC sample rate of 2048 Msps, this represents a channel bandwidth of at least 1 MHz. Channels shall be generated using an 4-tap PFB frontend.

4.1.4 Input Coarse Delay

The F-engine design shall provision for a runtime programmable coarse (1 ADC sample precision) delay, individually set for each analog input. The maximum depth of this delay shall be at least 8192 ADC samples (4096 ns at 2048 Msps sample rate)

4.1.5 Output Bandwidth

The F-engine design shall be capable of outputting at least 672 MHz of bandwidth, at 8+8 bit complex resolution per sample.

Output is via UDP packet streams over a pair of 100Gb/s Ethernet interfaces.

4.1.6 Output Format

Data shall be output using Ethernet jumbo frames. Data payloads shall group pairs of polarizations, multiple frequency channels, and multiple time samples in each packet ordered (fastest to slowest axis) as `polarization x time sample x freq. channel`

A variety of outputs with different numbers of frequency channels gathered in each packet shall be provided, eg.

1. 16 times, 96 channels, 2 polarizations, 8+8 bit (6144B packets; 14 packets / 1344 channels)
2. 16 times, 128 channels, 2 polarizations, 8+8 bit (8kB packets; 8 packets / 1024 channels)

4.2 Specification

4.2.1 ADC Sample Rate

The F-Engine design meets timing at a sample rate of 2048 Msps (FPGA DSP pipeline clock rate of 256 MHz)

4.2.2 Frequency Channels

The F-engine firmware generates 2048 complex-valued frequency channels using a 4-tap PFB with a 25-bit precision FFT. This FFT has sufficient dynamic range to negate the need for FFT shift control.

Scaling coefficients are provided with a dynamic range exceeding that of the FFT. Any FFT output power can be effectively converted to 8-bit data.

4.2.3 Input Coarse Delay

The F-engine design provides per-input programmable delay of up to 8192 ADC samples

4.2.4 Output Bandwidth

The F-engine design can output at least 1344 channels at 8+8 bit resolution.

The start point of the transmitted block of channels can be placed arbitrarily with a precision of 32 channels.

4.2.5 Output Format

Data packets can be constructed with payloads in (fastest to slowest axis) `polarization x time sample x freq channel` order.

Data payloads are constructed with 16 time samples per packet, and any multiple of 32 frequency channels, subject to the total Ethernet bandwidth available, and a maximum Ethernet MTU of 9000 Bytes.

Invidiual packets may be assigned independent destination IP addresses and UDP ports.

Each data packet has an application header of 16 bytes. Total protocol overhead (including this application header) is 70 bytes per packet (<2% for 4kB packets).

CHAPTER

FIVE

ATA RFSOC F-ENGINE FIRMWARE USER MANUAL

5.1 Introduction

This repository contains software for an RFSoC-based F-Engine (i.e. channelizer) system for the Allen Telescope Array (ATA). Corresponding firmware can be found on github.

The system digitizes sets of 16 RF signals from the ATA at a speed, f_s, of up to 2048 Msps and generates 2048 frequency channels over the Nyquist band.

This document described 8-bit *Voltage* mode firmware. (*Spectrometer* mode, and 4-bit *Voltage* mode implementations have also been created, though these are not described here)

5.1.1 This Document

This document describes the hardware configuration required by the F-Engine system, the runtime configuration procedures, and the software control functionality made available to the user. It also provides a description of the output data formats of each of the two data processing modes.

5.2 Nomenclature

5.2.1 Data Types

Throughout this document, data types are labelled in diagrams using the nomenclature $X.Y\mathrm{b}$. Unless otherwise stated, this indicates an X-bit signed, fixed-point number, with Y bits below the binary point.

Where this document indicates an N-bit complex number, this implies a $2N$-bit value with an N-bit real part in its most-significant bits, and an N-bit imaginary part in its least-significant bits.

5.3 Output Data Formats

5.3.1 Voltage Packets

The *Voltage* mode of the RFSoC firmware outputs a continuous stream of voltage data, encapsulated in UDP packets. The format used is common between SNAP and RFSoC ATA backends. Each packet contains a data payload of up to 8192 bytes, made up of 16 time samples for up to 256 frequency channels of dual-polarization data:

```
#define N_t 16
#define N_p 2

struct voltage_packet {
  uint8_t version;
  uint8_t type;
  uint16_t n_chans;
  uint16_t chan;
  uint16_t feng_id
  uint64_t timestamp;
  complex8 data[n_chans, N_t, N_p] // 8-bit real + 8-bit imaginary
};
```

The header entries are all encoded network-endian and should be interpretted as follows:

- `version`; *Firmware version*: Bit [7] is always 1 for *Voltage* packets. The remaining bits contain a compile-time defined firmware version, represented in the form bit[6].bits[5:3].bits[2:0]. This document refers to firmware version 2.1.0.
- `type`; *Packet type*: Bit [0] is 1 if the axes of data payload are in order [slowest to fastest] channel x time x polarization. This is currently the only supported mode. Bit [1] is 1 if the data payload comprises 8+8 bit complex integers. This is currently the only supported mode.
- `n_chans`; *Number of Channels*: Indicates the number of frequency channels present in the payload of this data packet.
- `chan`; *Channel number*: The index of the first channel present in this packet. For example, a channel number `c` implies the packet contains channels `c` to `c + n_chans - 1`.
- `feng_id`; *Antenna ID*: A runtime configurable ID which uniquely associates a packet with a particular SNAP board.
- `timestamp`; *Sample number*: The index of the first time sample present in this packet. For example, a sample number s implies the packet contains samples s to $s + 15$. Sample number can be referred to GPS time through knowledge of the system sampling rate and accumulation length parameters, and the system was last synchronized. See *sec-timing*.

The data payload in each packet is determined by the number of frequency channels it contains. The maximum is 8192 bytes. If `type & 2 == 1` each byte of data should be interpretted as an 8-bit complex number (i.e. 8-bit real, 8-bit imaginary) with the most significant 8 bits of each byte representing the real part of the complex sample in signed 2's complement format, and the least significant 8 bits representing the imaginary part of the complex sample in 2's complement format.

If `type & 1 == 1` the complete payload is an array with dimensions `channel x time x polarization`, with

- `channel` index running from 0 to `n_chans - 1`
- `time` index running from 0 to 15
- `polarization` index running from 0 to 1 with index 0 representing the X-polarization, and index 1 the Y-polarization.

5.4 Firmware Overview

The firmware described in this document is designed in Mathwork's Simulink using CASPERs FPGA programming libraries. The Simulink source file is available on github.

5.4.1 Building the Simulink Model

The SNAP firmware model (`zrf_volt_8ant_8bit.slx`) was built with the following software stack. Use other versions at your peril.

- Ubuntu 18.04 64-bit
- MATLAB/Simulink 2019a, including Fixed Point Designer Toolbox
- Xilinx Vivado System Edition 2020.2.1
- `mlib_devel` (version controlled within the `ata_rfsoc` repository.

To obtain and open the Simulink model:

```
# Clone the firmware repository
git clone https://github.com/realtimeradio/ata_rfsoc

# Clone relevant sub-repositories
cd ata_rfsoc
git submodule init
git submodule update

# Install the mlib_devel dependencies
# You may want to install these in a Python virtual environment
cd mlib_devel
pip install -r requirements.txt
```

Next, create a local environment specification file in the `ata_rfsoc` directory, named `startsg.local`. An example environment file is:

```
#!/bin/bash
###### User to edit these accordingly ######
export XILINX_PATH=/data/Xilinx/Vivado/2020.1
export MATLAB_PATH=/data/MATLAB/R2019a
# PLATFORM lin64 means 64-bit Linux
export PLATFORM=lin64
# Location of your Xilinx license
export XILINXD_LICENSE_FILE=/home/jackh/.Xilinx/Xilinx.lic

# Library tweaks
export LD_PRELOAD=${LD_PRELOAD}:"/usr/lib/x86_64-linux-gnu/libexpat.so"
# An optional python virtual environment to activate on start
export CASPER_PYTHON_VENV_ON_START=/home/jackh/casper-python3-venv
```

You should edit the paths accordingly.

To open the firmware model, from the top-level of the repository (i.e. the `ata_rfsoc` directory) run `./startsg`. This will open MATLAB with appropriate libraries, at which point you can open the `zrf_volt_8ant_8bit.slx` Simulink model.

5.4.2 Firmware Overview

A pictorial representation of the RFSoC firmware with annotated data path bit widths is shown in Fig. 5.1.

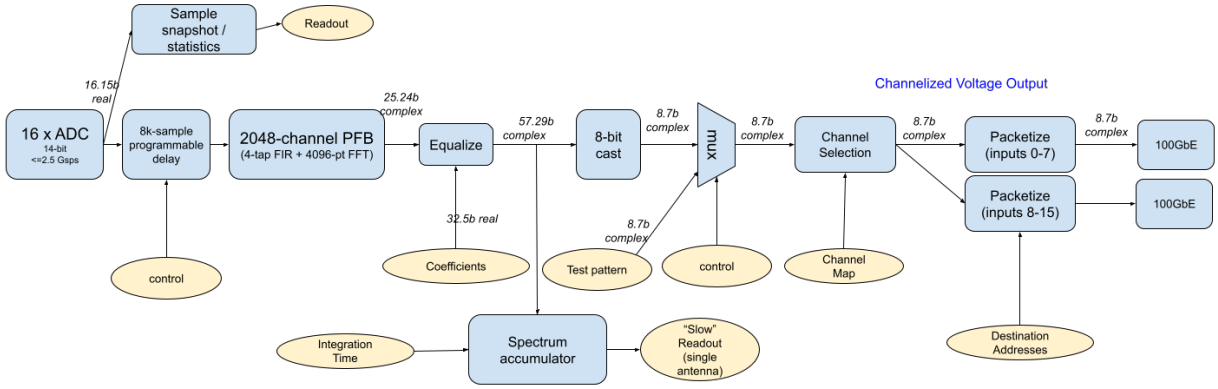

Fig. 5.1: A pictorial representation of the RFSoC firmware, showing major processing modules and the bit widths of their data paths. Yellow circles in this diagram represent the runtime-controllable elements of the pipeline.

In the remainder of this section an overview of the functional modules in the system is given.

5.4.3 Module Descriptions

Here basic explanations of the functionality of the different firmware processing modules is given. Where modules can be controlled or monitored at runtime, software routines to do so are described in *Run-time Control*.

5.4.3.1 Delay

The Delay module allows individual RFSoC inputs to be delayed by an integer number of ADC samples.

5.4.3.2 Timing

The Timing module allows multiple SNAP boards to be synchronized, and locks data timestamps to a known UTC origin. Mulitple board synchronization relies on each SNAP board being fed a time-aligned, distributed pulse, with an edge rate of << 1ms. Alignment of timestamps to UTC requires that the SNAP pulses have a positive edge aligned with a UTC second.

Typically, both of the above requirements can be met by using a syncronization signal which is a distributed GPS-locked Pulse-Per-Second (PPS).

Quality of board synchronization is determined by the nature of the PPS distribution system. For commercial PPS distribution equipment, using length-matched cables, synchronization will be within < 10 ADC samples.

The synchronization process is as follows:

1. Wait for a PPS to pass
2. Arm all SNAP boards in the system to trigger on the next PPS edge
3. Reset on-board spectra counters on the next PPS edge

Relevant Software Methods

- `sync_wait_for_pps`: Wait for an external PPS pulse to pass
- `sync_arm`: Arm the firmware such that the next PPS results in a reset of local counters
- `sync_get_last_sync_time`: Return the time that the firmware was last synchronized to a PPS pulse
- `sync_get_ext_count`: Return the number of PPS pulses returned since the FPGA was last programmed
- `sync_get_fpga_clk_pps_interval`: Return the number of FPGA clock ticks between the last two PPS pulses
- `sync_get_adc_clk_freq`: Infer the ADC clock rate from the number of FPGA clock ticks between PPS pulses

5.4.3.3 Filter Bank

The Filter Bank (aka Polyphase Filter Bank, PFB [pfb]) separates the X- and Y-polarization input broad-band data streams into 2048 frequency channels, starting at DC, with centers separated by $\frac{f_s}{4096}$. These baseband frequencies, from DC to $\frac{f_s}{2}$ represent different sky-frequencies when the input analog signals are mixed with LOs upstream of the digital system.

As shown in `fig-ata-rfsoc-feng`, the PFB receives real-valued broad-band data with 16-bits resolution (though only the most-significant 14 bits are populated by the ADC), and after processing delivers complex-valued spectra with 25-bit resolution.

Internally, the FFT data path is shown in Fig. 5.2.

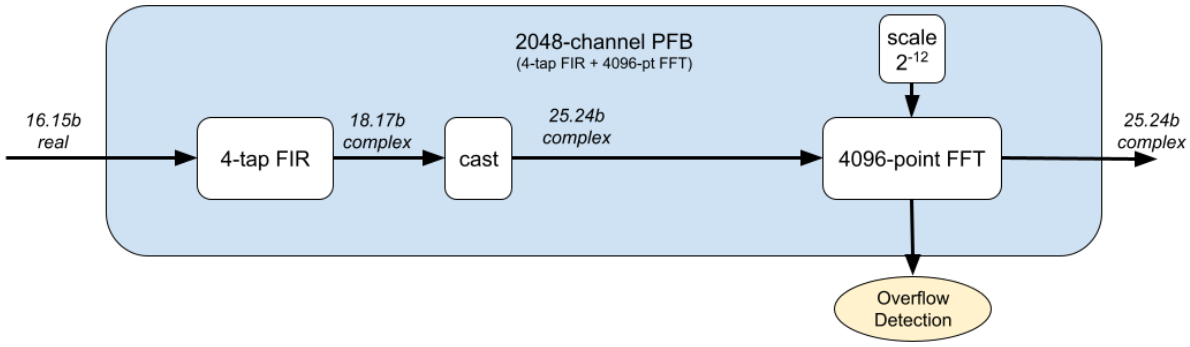

Fig. 5.2: A pictorial representation of the PFB internal datapath, showing internal data precision. Coefficients in the FIR and FFT modules are stored with 18 bits resolution. Overall signal amplitude growth in the FFT is controlled with a *shift schedule*, which is hardcoded to enforce 2^{-12} scaling. This is sufficient to guarantee against FFT overflow for any input signal.

In general, an FFT with 2^N points has N butterfly stages, and dynamic range should grow by N bits to guarantee against overflow.

Relevant Software Methods

- `fft_of_detect`: Count overflows in the FFT module
- `spec_read`: Read an accumulated spectrum
- `spec_plot`: Plot an accumulated spectrum

5.4.4 Spectral Power Accumulator

The Spectral Power Accumulator generates auto-correlation power-spectra for the two input data streams.

Spectra are computed by:

1. Multiplying 25-bit voltage inputs to generate 50-bit powers
2. Integrating these powers into 64-bit accumulators, with a runtime-configurable integration length

The nature of the bit handling in this implementation means that the accumulators can only guarantee against overflow for integrations of fewer than 2^{14} spectra. This amounts to just 53 ms of data. In practice, except in cases of high-power narrowband inputs, integrations of substantially longer without overflow are possible. In the event of overflow, data are saturated at $\pm 2^{63}$.

Spectra can be read from the power accumulator via software – appropriate for monitoring, debugging, and low time-resolution (~1s) observations.

A test vector injection module exists for the purposes of testing the spectrometer pipeline. When activated, this module replaces PFB data with a test pattern whose real parts are zero, and whose imaginary parts form a counter. For the X-polariaztion, FFT channel i takes the value $32 * (8floor(\frac{i}{4}) + mod(i, 4))$. For the Y-polariaztion, FFT channel i takes the value $32 * (8floor(\frac{i}{4}) + mod(i, 4)) + 4$.

Scaling in the spectrometer is such that the test vector inputs appear at the accumulator input with FFT channel i having the values:

- For the X-polarization: $8 * floor(\frac{i}{4}) + mod(i, 4)$.
- For the Y-polarization: $8 * floor(\frac{i}{4}) + mod(i, 4) + 4$.

This pattern can be checked against observed output data to verify that the two polarizations are being correctly identified, and accumulation length is being correctly set.

- `set_accumulation_length`: Set the number of spectra to be accumulated
- `spec_read`: Read an accumulated spectrum
- `spec_plot`: Plot an accumulated spectrum
- `spec_test_vector_mode`: Turn on and off the spectrometer test-vector mode

5.4.5 Equalization

In the voltage pipeline, post-PFB data are quantized to 8 bits prior to being transmitted over 100 GbE.

This substantial reduction of bit precision requires carefully managing input signal levels. As such, prior to equalization, spectra are multiplied by a runtime-programmable amplitude scaling factor. The scaling factors can be uniquely specified per polarization and per frequency channel, and should be used to ensure that data in each frequency channel exhibit an appropriate RMS.

Equalization coefficients can be computed by inferring power levels from the inbuilt spectral-power accumulator.

- `eq_load_coeffs`: Load a set of equalization coefficients

5.4.6 Voltage Channel Selection

The Voltage Mode output data path requires that only 2048 of the 4096 generated frequency channels are transmitted over 10 GbE. This down-selection takes place following 4-bit quantization, and is typically configured at initialization time using an appropriate configuration file (see *sec-config-file*).

The chosen 2048 channels are split into eight groups of 256 channels, each of which may be directed to a different IP address.

Channel selection should satisfy the following rules:

1. Each destination IP address should receive a start channel which is an integer multiple of 32.
2. If channel n is sent to IP address I, this IP address should also receive channels $n+1, n+2, n+3, n+4, n+5, n+6, n+7$.

Beyond these requirements, channels selected for transmission need not be contiguous and may also be duplicated. I.e. a block of 512 channels may be sent to each of 4 IP addresses.

- `select_output_channels`: Select which frequency channels are to be sent over 10 GbE
- `eq_load_test_vectors`: Load a custom set of test vectors into the voltage data path
- `eq_test_vector_mode`: Turn on or off voltage test vector injection

5.4.7 100Gb Ethernet

- `eth_reset`: Reset the Ethernet core
- `eth_set_mode`: Choose between spectrometer and voltage 10GbE outputs
- `eth_set_dest_port`: Set the destination UDP port for 10GbE traffic
- `eth_enable_output`: Turn on 100GbE transmission
- `eth_print_counters`: Print Ethernet packet statistics

5.5 Run-time Control

5.5.1 Installing the Control Library

The *ata_snap* python library is provided to control the F-Engine firmware design. It requires Python >=3.5 and the following custom python libraries:

1. casperfpga (a control library for interacting with CASPER hardware, such as SNAP boards)
2. adc5g (a library for configuring the ADC5G card)

These libraries are bundled in the *ata_snap* repository to minimize issues with library version compatibility.

To install the control libraries:

```
# Clone the ata_snap repository
git clone https://github.com/realtimeradio/ata_snap

# Clone relevant sub-repositories
cd ata_snap
git submodule init
git submodule update
```

(continues on next page)

(continued from previous page)

```
# Install casperfpga
cd sw/casperfpga
# Install casperfpga dependencies (requires pip, which can be installed with `apt␣
↪install python3-pip`
pip install -r requirements.txt
# Install casperfpga
python setup.py install

# Install the adc5g library
cd ../adc5g/adc5g
python setup.py install

# Install the ata_snap library
cd ../../../ata_snap
python setup.py install
```

If the library has installed correctly, in a Python shell, you should be able to successfully execute

```
from ata_snap import ata_snap_fengine
```

5.6 Configuration Recipes

Simple use of the *ata_snap* library comprises the following steps:

1. Program the RFSoC boards with appropriate firmware
2. Configure any runtime settings
3. Synchronize RFSoC boards with UTC
4. Turn on data flow

Once these steps are complete, data will be transmitted over Ethernet and downstream software can catch and process this data as desired.

For a single SNAP board, all of these steps can be carried out at once using a provided intialization scrips `snap_feng_init.py`. This script has the following use template:

```
$ snap_rfsoc_init.py -h
usage: snap_feng_init.py [-h] [-s] [-t] [--eth_spec] [--eth_volt] [-a ACCLEN]
                         [-f FFTSHIFT] [--specdest SPECDEST]
                         host fpgfile configfile

Program and initialize a SNAP ADC5G spectrometer

positional arguments:
  host                  Hostname / IP of SNAP
  fpgfile               .fpgfile to program
  configfile            Configuration file

optional arguments:
  -h, --help            show this help message and exit
  -s                    Use this flag to re-arm the design's sync logic
                        (default: False)
  -t                    Use this flag to switch to post-fft test vector outputs
```

(continues on next page)

(continued from previous page)

```
                        (default: False)
  --eth_spec            Use this flag to switch on Ethernet transmission of the
                        spectrometer (default: False)
  --eth_volt            Use this flag to switch on Ethernet transmission of
                        F-engine data (default: False)
  -a ACCLEN             Number of spectra to accumulate per spectrometer dump.
                        Default: get from config file (default: None)
  --specdest SPECDEST   Destination IP address to which spectra should be sent.
                        Default: get from config file (default: None)
```

5.6.1 Configuration File

```
# Accumulation length, in spectra.
acclen: 300000
# Coeffs should be a single number, or an array
# of 4096 numbers to set one coefficient per channel.
coeffs: 100
# UDP port for 10GbE data
dest_port: 10000
spectrometer_dest: 10.11.10.173
# Define which channels should be output
# over 10GbE in voltage dump mode.
voltage_output:
  start_chan: 0
  n_chans: 1024
  # Channels will be spread over the following
  # destinations so that the first n_chans // len(dests)
  # go to the first IP address, etc.
  dests:
      - 10.11.10.173
# All relevant IP/MAC mapping should be manually
# specified here
arp:
  10.11.10.173: 0xaeecc7b400ff
  10.11.10.174: 0xaeecc7b400a0
```

CHAPTER

SIX

ATA_SNAP RFSOC FIRMWARE API REFERENCE

class ata_snap.ata_rfsoc_fengine.**AtaRfsocFengine**(*host*, *feng_id=0*, *pipeline_id=0*)

Bases: *ata_snap.ata_snap_fengine.AtaSnapFengine*

This is a class which implements methods for programming and communicating with an RFSoC board running the ATA F-Engine firmware.

Parameters

- **host** (*str or casperfpga.CasperFpga*) – Hostname of board associated with this instance. Or, pre-created casperfpga.CasperFpga instance to the relevant host
- **feng_id** (*int*) – Antenna ID of the antenna connected to this SNAP. This value is used in the output headers of data packets.
- **pipeline_id** (*int*) – pipeline ID of the antenna connected to this SNAP. This value is used to associate an F-Engine with a pipeline instance

adc_balance()

Attempt to balance the ADC cores offset and gains, using the current input signal.

Returns offset, gain, phase The currently loaded offset and gain setting. Each is a vector with 4 elements - one per ADC core.

adc_get_mismatch(*n_snapshot=16*)

Get the offset and gain mismatch between the two interleaved cores forming each polarization.

Parameters

- **pol** (*int*) – Polarization to interrogate (0 or 1)
- **n_snapshot** (*int*) – Number of snapshots to take. More gives better statistics, but takes longer

Returns

2x2 array [[pol0 offset, pol0 gain], [pol1 offset, pol1 gain]] offset: the number of ADC counts which core 1 is is offset by relative to core 0. E.g. if offset=0.2, then the mean of all the core 1 ADC samples is 0.2 higher than the mean of all core 0 samples.

gain: the relative gain of core 1 vs core 0. E.g., if gain = 1.03, then the mean power of all core 1 samples is 1.03 times the mean power of all core 1 samples.

Return type (float, float)

adc_get_samples()

Get a block of samples from both ADC inputs, captured simultaneously.

This method requires that the currently programmed fpg file is known. This can be achieved either by programming the board with program(<fpgfile>), or by running fpga.get_system_information(<fpgfile>) if the board was already programmed outside of this class.

Returns x, y (numpy arrays of ADC sample values)

Return type numpy.ndarray

adc_get_stats(*per_core=False*)
Get the mean value and power, and a count of overflow events for the system's ADCs. Statistics are calculated over the last 512k samples, and are computed from samples obtained from all ADC channels simultaneously.

Parameters **per_core** (*bool*) – If True, return stats for each ADC core. If false, return stats for each ADC channel.

Returns A 3-tuple of numpy.ndarray, with either 4 entries (if per_core=True) or 2 entries (if per_core=False). The tuple is (clip_count, mean, mean_power) where clip_count is the number of clipping events in the last 512k samples, mean is the average of the last 512k samples, and mean_power is the mean power of the last 512k samples. If per_core=True, each array has 4 entries, representing the four ADC cores. Cores 0, 2 digitize the X-pol RF input, and cores 1, 3 digitize the Y-pol input. In this case stats are calculated over 256k samples per core. If per_core=False, each array has 2 entries, with the first address the X-pol RF input, and the second the Y-pol. In this case statistics are calculated over 512k samples per polarization.

Return type (numpy.ndarray, np.ndarray, np.ndarray)

adc_initialize()
Initialize the ADC interface by performing FPGA<->ADC link training. Put the ADC chip in dual-input mode. This method must be called after programming a SNAP, and is called automatically if using this class's *program* method with init_adc=True.

change_feng_id(*feng_id*)
Update the Fengine ID field of both spectrometer packets and voltage packets, without otherwise effecting the configuration of a running board.

Also update this instance's *feng_id* attribute

Parameters **feng_id** (*int*) – New F-Engine ID for this instance

eq_balance(*pol*, *target_rms=0.125*, *cutoff=2.0*, *medfil_ksize=401*, *conv_ksize=50*)
Tweak the EQ coefficients for a polarization to target a particular post-EQ RMS.

Parameters

- **pol** (*int*) – Polarization to equalize.
- **target_rms** (*float*) – Target RMS. This is specified in units of 1 8-bit post-equalization least significant bit. I.e., target_rms=1 will target an RMS of 1 on a scale where 8-bit post-EQ values run from -255 to 255. This parameter is always relative to 8-bit values. So, for 4-bit quanitization, target_rms = 2**4 will target an RMS of one 4-bit LSB.
- **cutoff** (*float*) – The scale, relative to the mean coefficient, at which coeffiecients are saturated. For example, if the mean coefficient is 1000, and `cutoff` is 2, coefficients will saturated at a value of 2000. This allows the bandpass to still be visible in the equalized data.
- **medfil_ksize** (*int*) – Size of median filter kernel to use for RFI removal. Should be odd.

- **conv_ksize** (*int*) – Convolution kernel size to use for bandpass smoothing.

Returns The loaded coefficients as read back. Note that these will be slightly different to the loaded coefficients since they will have been saturated and quantized to the specifications of the firmware.

Return type Integer coefficients as returned by `eq_read_coeffs`

eq_compute_coeffs(*target_rms=0.5*, *medfil_ksize=401*, *conv_ksize=100*, *acc_len=50000*)

Get appropriate EQ coefficients to scale FFT output for an appropriate post-quantization power target. Do this by grabbing a full bit-precision spectrum, filtering out RFI and smoothing, and then computing the scaling required to reach a target power level.

Parameters

- **target_rms** (*float*) – The target voltage RMS. This should be specified relative to signed data normalized to the range +/-1. I.e., a target_rms of 1./2**7 represents an RMS of one least-significant bit after quantizing to 8-bits. A target_rms of 1./2**3 represents one least-significant bit after quantizing to 4-bits.
- **medfil_ksize** (*int*) – Size of median filter kernel to use for RFI removal. Should be odd.
- **conv_ksize** (*int*) – Convolution kernel size to use for bandpass smoothing.
- **acc_len** (*int*) – If specified, use this accumulation length for the computation. Accumulation length should be sufficiently long to obtain good autocorrelation SNR. After the EQ coeffients are obtained, the accumulation length the firmware was using before this function was invoked is reloaded.

Returns x_coeffs, y_coeffs: A tuple of coefficients. Each is a numpy array of floating-point values.

Return type numpy.array, numpy.array

eq_load_coeffs(*pol*, *coeffs*)

Load coefficients with which to multiply data prior to 4-bit quantization. Coefficients are rounded and saturated such that they fall in the range (0, 2048), with a precision of 2**-5 = 0.03125. A single coefficient can be provided, in which case coefficients for all frequency channels will be set to this value. If an array or list of coefficients are provided, there should be one coefficient per frequency channel in the firmware pipeline.

Parameters

- **pol** (*int*) – Selects which polarization vectors are being loaded to (0 or 1) 0 is the first ADC input, 1 is the second.
- **coeffs** (*float, or list / numpy.ndarray*) – The coefficients to load. If *coeffs* is a single number, this value is loaded as the coefficient for all frequency channels. If *coeffs* is an array or list, it should have length self.n_chans_f / self.n_coeff_shared or length self.n_chans_f. If the latter, the coefficients will be decimated by a factor of self.n_coeff_shared. Element [i] of this vector is the coefficient applied to channels i through i+self.n_coeff_shared-1 if the array has length self.n_chans_f / self.n_coeff_shared. If the array has length self.n_chans_f then element [self.n_coeff_shared*i] is the coefficient which will be applied to channels i through i+self.n_coeff_shared-1. Coefficients are quantized to UFix32_5 precision.

Raises **AssertionError** – If an array of coefficients is provided with an invalid size, if any coefficients are negative, or if pol is a non-allowed value

Returns 2-tuple coeffs, bin_pt. coeffs: An array of self.n_chans_f indicating the integer coefficients loaded. bin_pt: position of binary point with which firmware interprets coefficients. Eg: bin_pt=5 means that coefficients are interpreted as coeffs / 2^5

Return type numpy.ndarray, int

eq_load_test_vectors(*pol*, *tv*)
Load test vectors for the Voltage pipeline test vector injection module.

Parameters

- **pol** (*int*) – Selects which polarization vectors are being loaded to (0 or 1) 0 is the first ADC input, 1 is the second.
- **tv** (*numpy.ndarray or list of ints*) – Test vectors to be loaded. *tv* should have self.n_chans_f elements. tv[i] is the test value for channel i. Each value should be an 8-bit number - the most-significant 4 bits are interpretted as the 4-bit, signed, real part of the data stream. The least-significant 4 bits are interpretted as the 4-bit, signed, imaginary part of the data stream.

Raises **AssertionError** – If an array of test vectors is provided with an invalid size, or if pol is a non-allowed value

eq_read_coeffs(*pol*, *return_float=False*)
Read currently loaded coefficients for one polarization.

Parameters

- **pol** (*int*) – Selects which polarization vectors are being read to (0 or 1) 0 is the first ADC input, 1 is the second.
- **return_float** (*Bool*) – If True, return floating-point coefficients as interpretted by the SNAP If False, return integer coefficients and a binary point position.

Returns If return_float: coeffs; If not return_float: 2-tuple coeffs, bin_pt. coeffs: An array of self.n_chans_f indicating the coefficients loaded. bin_pt: If return_float=False, this is the position of binary point with which firmware interprets coefficients. Eg: bin_pt=5 means that coefficients are interpreted as coeffs / 2^5

Return type numpy.ndarray or numpy.ndarray, int

eq_test_vector_mode(*enable*)
Turn on or off the test vector mode downstream of the 4-bit quantizers. This mode can be used to replace the FFT output in the voltage data path with an arbitrary pattern which can be set with *eq_load_test_vectors*

Parameters **enable** (*bool*) – True to turn on the test mode, False to turn off

eth_enable_output(*enable=True*, *interface='all'*)
Enable the 10GbE output datastream. Only do this after appropriately setting an output configuration and setting the pipeline mode with *eth_set_mode*. For spectra mode, prior to enabling Ethernet the destination address should be set with *spec_set_destination*. For voltage mode, prior to enabling Ethernet configuration should be loaded with *select_output_channels*

Parameters

- **enable** (*bool*) – Set to True to enable Ethernet transmission, or False to disable.
- **interface** (*integer or 'all'*) – Which physical interface to enable / disable.

eth_print_counters()
Print ethernet statistics counters from all ethernet cores. This is a simple wrapper around casperfpgas gbes.read_counters() method.

eth_reset(*interface='all'*)
Reset the Ethernet core. This method will clear the reset after asserting, and will leave the transmission stream disabled. Reactivate the Ethernet core with *eth_enable_output*

Parameters **interface** (*integer or 'all'*) – Which physical interface to enable / disable.

eth_set_dest_port(*port*, *interface=None*)
Set the destination UDP port for output 100GbE packets.

Parameters

- **port** (*int*) – UDP port to which traffic should be sent.
- **interface** (*None*) – Unused. Maintained for API compatibility

eth_set_mode(*mode='voltage'*)
Set the 10GbE output stream to either "voltage" or "spectra" mode. To prevent undesired behaviour, this method will disable Ethernet transmission prior to switching. Transmission should be re-enabled if desired using *eth_enable_output*.

Parameters **mode** (*str*) – "voltage" or "spectra"

Raises **AssertionError** – If mode is not an allowed value

fft_of_detect()
Read the FFT overflow detection register. Will return True if an overflow has been detected since the last reset. False otherwise.

Returns True if FFT overflowed since the last reset accumulation period, False otherwise.

Return type bool

fft_of_detect_reset()
Reset the FFT overflow detection register.

get_accumulation_length()
Get the number of spectra currently being accumulated in the on-board spectrometers.

Returns acclen; Accumulation length

Rtype int

get_delay(*pol*)
Get the currently loaded delay, in units of ADC clock cycles, of a pipeline input.

Parameters **pol** (*int*) – Polarization to read (0 or 1)

Returns delay, in units of ADC clock cycles

Return type int

get_pending_delay(*pol*)
Get the upcoming delay, in units of ADC clock cycles, of a pipeline input, and the time when it will be loaded.

Parameters **pol** (*int*) – Polarization to read (0 or 1)

Returns (delay, time_to_load) tuple delay, in units of ADC clock cycles time_to_load, in units of seconds. If negative, represents the number of seconds since the delays were loaded.

Return type (int, float)

is_programmed()
Returns True if the fpga appears to be programmed with a valid F-engine design. Returns False otherwise.

The test this method uses is searching for the register named *version* in the running firmware, so it can easily be fooled.

Returns True if programmed, False otherwise

Return type bool

n_ants_per_board = 8
Number of antennas on a board

n_ants_per_output
Number of antennas per 100G link

n_chans_per_block = 32
Number of channels reordered in a single word

n_interfaces = 1
Number of available 10GbE interfaces

pps_source = 'board'
After programming set the PPS source to the front panel input

program(*fpgfile*)
Program a SNAP with a new firmware file.

Parameters **fpgfile** (*str*) – .fpg file containing firmware to be programmed

quant_spec_read(*pol=0*, *flush=True*, *normalize=False*)
Read a single accumulated spectrum of the 4-bit quantized data

This method requires that the currently programmed fpg file is known. This can be achieved either by programming the board with program(<fpgfile>), or by running fpga.get_system_information(<fpgfile>) if the board was already programmed outside of this class.

Parameters

- **pol** (*int*) – Polarization to read
- **flush** (*Bool*) – If True, throw away one integration prior to getting data. This can be desirable if (eg) EQ coefficients have been recently changed.
- **normalize** (*Bool*) – If True, divide out the accumulation length and firmware scaling, returning floating point values. Otherwise, return integers and leave these factors present.

Raises **AssertionError** – if pol is not 0 or 1

Returns A numpy array containing the polarization's power spectrum

Return type numpy.array

select_output_channels(*start_chan*, *n_chans*, *dests=['0.0.0.0']*, *dest_ports=[10000]*, *blank=False*)
Select the range of channels which the voltage pipeline should output.

Example usage:

Send channels 0..255 to 10.0.0.1: select_output_channels(0, 256, dests=['10.0.0.1'])

Send channels 0..255 to 10.0.0.1, and 256..512 to 10.0.0.2 select_output_channels(0, 512, dests=['10.0.0.1', '10.0.0.2'])

Parameters

- **start_chan** (*int*) – First channel to output
- **n_chans** (*int*) – Number of channels to output

- **dests** (*list of int*) – List of IP address strings to which data should be sent. The first n_chans / len(dests) will be sent to dest[0], etc..
- **dest_ports** – List of destination UDP ports to which data should be sent. The first n_chans / len(dests) will be sent to dest[0], etc.. The length of this list should be the same as the length of the `dests` list.
- **blank** (*bool*) – If True, disable this output stream, but configure upstream reorders as if it were enabled.

Raises **AssertionError** – If the following conditions aren't met: *start_chan* should be a multiple of self.n_chans_per_block (4) *n_chans* should be a multiple of self.n_chans_per_block (4) *interface* should be <= self.n_interfaces

Returns A dictionary, keyed by destination IP, with values corresponding to the ranges of channels destined for this IP. Eg, if sending 512 channels over two destinations '10.0.0.10' and '10.0.0.11', this function might return {'10.0.0.10': [0,1,..,255], '10.0.0.11': [256,257,..,511]}

Return type dict

set_accumulation_length(*acclen*)
Set the number of spectra to accumulate for the on-board spectrometer.

Parameters **acclen** (*int*) – Number of spectra to accumulate

set_delay_tracking(*delay*, *delay_rate*, *load_time=None*, *load_sample=None*)
Set this F-engine to track a given delay curve.

Parameters

- **load_time** (*float*) – Unix time at which delay should be applied. If None, a load_sample should be given.
- **load_sample** (*int*) – Spectrum index at which delay should be applied. Used only if `load_time` is not provided.
- **delay** (*float*) – Delay, in nanoseconds, which should be applied at the appropriate time. Whole ADC sample delays are implemented using a coarse delay, while sub-sample delays are implemented as a post-FFT phase rotation.
- **rate** (*delay*) – Delay rate, in nanoseconds per second. The incremental delay which should be added to the current delay each second after the given load time. Delay rate is converted from nanoseconds-per-second to nanoseconds-per-1024-spectra, which is the update cadence of the underlying firmware. For a 2.048GHz sampling rate, and 4096-point FFT, this corresponds to an updated delay every 2 ms.

Returns Spectrum index at which new delays will be loaded. This should either be equal to `load_sample`, if provided, or will represent the first sample boundary after `load_time`.

Return type int

set_delays(*delays*, *load_time=-1*, *clock_rate_hz=2048000000*, *sync_time=None*)
Set the delay, in units of ADC clock cycles, of a pipeline input.

Parameters

- **delay** (*float*) – tuple of delays to apply to a polarization pair [x-pol delay, y-pol delay]
- **load_time** (*int*) – UNIX time at which delays should be loaded, or -1 to load immediately

- **clock_rate_hz** (*int*) – ADC clock rate in Hz. If None, the clock rate will be computed from the observed PPS interval, which could fail if the PPS is unstable or not present.
- **sync_time** (*int*) – The time, in UNIX seconds, at which the F-engine was last synchronized. If None, the sync time will be queried from the board using the *get_last_sync_time()* method.

Returns None, unless load_time is provided. In this case return the spectrum ID (the spectrum count, relative to the F-engine sync time) at which the requested delays will be loaded

Rval int

spec_plot(*mode='auto'*)

Plot an accumulated spectrum using the matplotlib library. Frequency axis is infered from the ADC clock frequency detected with *sync_get_adc_clk_freq*.

Parameters **mode** (*str:*) – "auto" to plot a power spectrum from the X and Y polarizations on separate subplots. "cross" to plot the magnitude and phase of a complex-valued cross-power spectrum of the two polarizations.

Raises **AssertionError** – if mode is not "auto" or "cross"

spec_read(*mode='auto'*, *flush=False*, *normalize=False*)

Read a single accumulated spectrum.

This method requires that the currently programmed fpg file is known. This can be achieved either by programming the board with program(<fpgfile>), or by running fpga.get_system_information(<fpgfile>) if the board was already programmed outside of this class.

Parameters

- **mode** (*str:*) – "auto" to read an autocorrelation for each of the X and Y pols. "cross" to read a cross-correlation of Xconj(Y).
- **flush** (*Bool*) – If True, throw away one integration prior to getting data. This can be desirable if (eg) EQ coefficients have been recently changed.
- **normalize** (*Bool*) – If True, divide out the accumulation length and firmware scaling, returning floating point values. Otherwise, return integers and leave these factors present.

Raises **AssertionError** – if mode is not "auto" or "cross"

Returns If mode="auto": A tuple of two numpy arrays, xx, yy, containing a power spectrum from the X and Y polarizations. If mode="cross": A complex numpy array containing the cross-power spectrum of Xconj(Y).

Return type numpy.array

spec_set_destination(*dest_ip*)

Set the destination IP address for spectrometer packets.

Parameters **dest_ip** (*str*) – Destination IP address. E.g. "10.0.0.1"

spec_test_vector_mode(*enable*)

Turn on or off the test vector mode in the spectrometer data path. This mode replaces the FFT output in the data path with 12 bit counter which occupies the most significant bits of the imaginary part of the FFT output. I.e. when enabled, the imaginary part of each spectrum is a ramp from 0..4095 / 2**17

Parameters **enable** (*bool*) – True to turn on the test mode, False to turn off

sync_arm(*manual_trigger=False*)

Arm the FPGA's sync generators for triggering on a subsequent PPS. The arming proceedure is the following: 1. Wait for a PPS to pass using *sync_wait_for_pps*. 2. Arm the FPGA's sync register to trigger on

the next+2 PPS 3. Compute the time this PPS is expected, based on this computer's clock. 4. Write this time as a 32-bit UNIX time integer to the FPGA, to record the sync event.

Parameters **manual_trigger** (*bool*) – Use a software sync, rather than relying on an external PPS pulse. See *sync_manual_trigger* for more information.

Returns Sync trigger time, in UNIX format

Rval int

sync_get_adc_clk_freq()
Infer the ADC clock period by counting FPGA clock ticks between PPS events.

Returns ADC clock rate in MHz

Rval float

sync_get_ext_count()
Read the number of external sync pulses which have been received since the board was last programmed.

Returns Number of sync pulses received

Rval int

sync_get_fpga_clk_pps_interval()
Read the number of FPGA clock ticks between the last two external sync pulses.

Returns FPGA clock ticks

Rval int

sync_get_last_sync_time()
Get the sync time currently stored on this FPGA

Returns Sync trigger time, in UNIX format

Rval int

sync_manual_trigger()
Issue a sync using the F-engine's built-in software trigger. This can be useful for single board deployments or testing where an external PPS signal may not be available. However, a manual sync cannot synchronize multiple boards, and will only be aligned to the expected sync time at the ~1s level.

sync_select_input(*pps_source*)
Select which PPS input is used to drive the design's timing subsystem.

Parameters **pps_source** (*str*) – Which PPS source should be used. "adc" indicates the ADC5G's sync input. "board" indicates the SNAP board's dedicated sync input.

sync_wait_for_pps()
Block until an external PPS trigger has passed. I.e., poll the FPGA's PPS count register and do not return until is changes.

CHAPTER

SEVEN

INDEX

- genindex
- modindex
- search

BIBLIOGRAPHY

[e2v] e2v EV8AQ160 datasheet (https://www.teledyne-e2v.com/content/uploads/2019/10/EV8AQ160_0846K.pdf)

[hmcad1511] Analog Devices HMCAD1511 datasheet (https://www.analog.com/media/en/technical-documentation/data-sheets/hmcad1511.pdf)

[casper_snap] See https://github.com/casper-astro/casper-hardware/blob/master/FPGA_Hosts/SNAP/README.md

[casper_adc5g] See https://casper.ssl.berkeley.edu/wiki/ADC1x5000-8

[rpi] See https://www.raspberrypi.org/products/raspberry-pi-2-model-b

[pfb] Price, D; Spectrometers and Polyphse Filterbanks in Radio Astronomy; 2016; https://arxiv.org/abs/1607.03579

[quant] Thompson, A et al.; Convenient Formulas for Quantization Efficiency; 2007; https://agupubs.onlinelibrary.wiley.com/doi/full/10.1029/2006RS003585

PYTHON MODULE INDEX

a

INDEX

F

G

I

M

N

P

Q

S