LWA352 SNAP2 F-Engine Documentation

Release

caltech_lwa-45820d5:lwa_f-1.0.1-1.0.0-36-g45820d52

Jack Hickish

Nov 18, 2022

CONTENTS:

Contents:

CHAPTER

ONE

INSTALLATION

The LWA 352 F-Engine pipeline is available at https://github.com/realtimeradio/caltech-lwa. Follow the following instructions to download and install the pipeline.

Specify the build directory by defining the `BUILDDIR` environment variable, eg:

```
export BUILDDIR=~/src/
mkdir -p $BUILDDIR
```

1.1 Get the Source Code

Clone the repository and its dependencies with:

```
# Clone the main repository
cd $BUILDDIR
git clone https://github.com/realtimeradio/caltech-lwa
# Clone relevant submodules
cd caltech-lwa
git submodule init
git submodule update
```

1.2 Install Prerequisites

1.2.1 Firmware Requirements

The LWA-253 F-Engine firmware can be built with the CASPER toolflow, and was designed using the following software stack:

- Ubuntu 18.04.0 LTS (64-bit)
- MATLAB R2019a
- Simulink R2019a
- MATLAB Fixed-Point Designer Toolbox R2019a
- Xilinx Vivado HLx 2019.1.3
- Python 3.6.9

It is *strongly* recommended that the same software versions be used to rebuild the design.

CHAPTER

TWO

F-ENGINE SYSTEM OVERVIEW

2.1 Overview

The LWA352 F-Engine firmware is designed to run on a SNAP2[1] FPGA board, and provides channelization of 64 analog data streams, sampled at up to 200 Msps, into 4096 sub-bands.

After channelization, data words are requantized to 4-bit resolution (4-bit real + 4-bit imaginary) and a subset of the 4096 generated frequency channels are output as a UDP/IP stream over a single 40 Gb/s Ethernet interface.

The top-level specs of the F-Engine are:

Parameter	Value	Notes
Number of analog inputs	64	32-inputs per SNAP2 FMC+ port
Maximum sampling rate	200 Msps	Limited by ADC speed & timing constraint target
Test inputs	Noise; zeros	Firmware contains 6 independent gaussian noise generators. Any of the 64 data streams may be replaced with any of these digital noise sources, or zeros.
Delay compensation	<=8191 samples	Programmable per- input between 5 and 8191 samples
Polyphase Filter Bank Channels	4096	
Polyphase Filter Bank Window	Hamming; 4-tap	
Polyphase Filter Bank Input Bitwidth	9 bits	LSB of ADC samples are discarded. **TODO: Fix this**
FFT Coefficient Width	18 bits	
FFT Data Path Width	18 bits	
Post-FFT Scaling Coefficient Width	16	
Post-FFT Scaling Coefficient Binary Point	5	
Number of Post-FFT Scaling Coefficients	32768	One coefficient per analog input. One coefficient per 8 frequency channels
Post-Quantization Data Bitwidth	4	4-bit real; 4-bit imaginary
Frequency Channels Output	<=3072	Runtime programmable. Maximum is set by total data rate which is limited to 40Gb/s (including protocol overhead). 3072 channels = approx 38Gb/s

[1] See SNAP2 design document and SNAP2 CASPER workshop presentation (2017)

A block diagram of the F-engine – which is also the top-level of the Simulink source code for the firmware – is shown in Fig. 2.1.

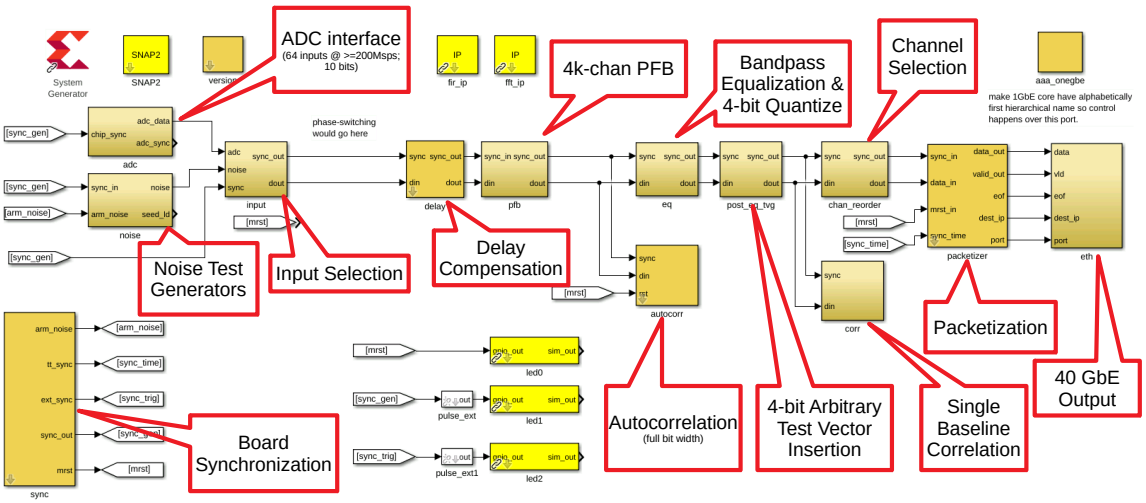

Fig. 2.1: F-Engine top-level Simulink diagram.

2.1.1 Initialization

The functionality of individual blocks is described below. However, in order to simply get the firmware into a basic working state the following process should be followed:

1. Program the FPGA
2. Initialize all blocks in the system
3. Trigger master reset and timing synchronization event.

In a multi-board system, the process of synchronizing a board can be relatively involved. For testing purposes, using single board, a simple software reset can be used in place of a hardware timing signal to perform an artificial synchronization. A software reset is automatically issued as part of system initialization.

The following commands bring the F-engine firmware into a functional state, suitable for testing. See Section 4 for a full software API description

```
# Import the SNAP2 F-Engine library
from lwa_f import snap2_fengine

# Instantiate a Snap2Fengine instance to a board with
# hostname 'snap2-rev2-11'
f = snap2_fengine.Snap2Fengine('snap2-rev2-11')

# Program a board
f.program() # Load whatever firmware is in flash
# Wait 30 seconds for the board to reboot...

# Initialize all the firmware blocks
# and issue a global software reset
f.initialize(read_only=False)
```

2.1.2 Block Descriptions

Each block in the firmware design can be controlled using an API described in Section 4. Here the basic functionality of each block is described.

ADC

The ADC block contains the interface to the physical ADC cards, and presents ADC data to the Simulink DSP pipeline. It is controlled using the API in Section 4.2.5. This API provides the ability to read ADC samples, and is required to "train" the ADC to FPGA link after the FPGA is programmed.

The block has the following ports:

Name	Direction	Data Type	Description
`chip_sync`	in	Bool	Path from Simulink to the (multiple) ADC physical `sync` pins. This signal is `AND` -ed with an internal control flag controlled by the ADC block's API.
`adc_data`	out	`UFix_640_0`	Concatenated, 10-bit, two's-complement ADC samples from 64 channels. Bits `10(i+1)-1` down to `10i` correspond to a sample from channel `i` in `Fix_10_9` format.
`adc_sync`	out	Bool	A 1-cycle pulse which precedes the samples from the "even-sample" ADC core.

CHAPTER

THREE

OUTPUT DATA FORMAT

UDP packets output from the F-Engine are described below. Packet format is chosen to be broadly compatible with `bifrost`'s packet receive architecture, and to have the following features:

- Supports arbitrary numbers of channels / antenna inputs per packet
- Allows channels to be easily distributed among multiple destinations (X-Engine pipelines)

```
struct f_packet {
        uint64_t  seq;
        uint32_t  sync_time;
        uint16_t  nsignal;
        uint16_t  nsignal_tot;
        uint16_t  nchan;
        uint16_t  nchan_tot;
        uint32_t  chan_block_id;
        uint32_t  chan0;
        uint32_t  signal0;
        uint8_t   data[nchan, nsignal];
};
```

Packet Fields are as follows:

Field	Format	Units	Description
`seq`	uint64	spectra count	Spectrum index, with seq=0 corre sponding to the spectra at UNIX time `sync_time`
`sync_time`	uint32	UNIX seconds	UNIX time corresponding to the first (`seq=0`) spectrum
`nsignal`	uint16		Number of inputs present in a packet
`nsignal_tot`	uint16		Number of inputs present in the complete multi-SNAP system
`nchan`	uint16		Number of frequency channels present in a packet
`nchan_tot`	uint16		Number of frequency channels present in a data stream destined for a single destination.
`chan_block_id`	uint32		Index of this block of channels. I.e. `mod(chan0, nchan)`
`chan0`	uint32		The index of the first channel present in a packet
`signal0`	uint32		The index of the first input present in this packet
`data`	uint8		An array of `nchan x nsignal` data samples, with channel the slowest-varying axis, and input number the fastest-varying axis. Each sample should be interpretted as complex data, with the most-significant and least- significant 4 bits corresponding to the real and imaginary data parts, respectively. Each component should be interpretted as a two's complement signed 4-bit integer

Packet parameters are chosen to be compatible with the total number of frequencies the system is required to process,

the number of independent X-Engine pipelines in the system, a target packet size (generally larger is better, up to the Ethernet network's largest allowed *Maximum Transmission Unit* (MTU).

In practice, the LWA352 system operates with

- `nsignal = 64`
- `nsignal_tot = 704`
- `nchan = 96`
- `nchan_tot = 192`

CHAPTER

FOUR

CONTROL INTERFACE

4.1 Overview

A Python class `Snap2Fengine` is provided to encapsulate control of individual blocks in the firmware DSP pipeline. The structure of the software interface aims to mirror the hierarchy of the firmware modules, through the use of multiple `Block` class instances, each of which encapsulates control of a single module in the firmware pipeline.

In testing, and interactive debugging, the `Snap2Fengine` class provides an easy way to probe board status for a SNAP2 board on the local network.

In order to integrate with the larger LWA352 control framework, control and monitoring of multiple F-Engines can also be carried out through the passing of JSON-encoded messages through an `etcd`[1] key-value store. This mechanismi, shown in Fig. 4.1, utilizes the Python class `Snap2FengineEtcdClient` to translate commands and responses between the `etcd` format and the underlying `Snap2Fengine` method calls.

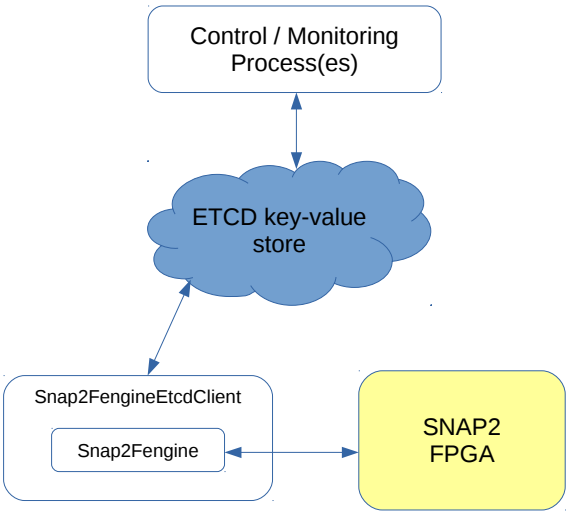

Fig. 4.1: The top-level control architecture.

[1] See etcd.io

4.2 `Snap2Fengine` Python Interface

The `Snap2Fengine` class can be instantiated and used to control a single SNAP2 board running LWA's F-Engine firmware. An example is below:

```
# Import the SNAP2 F-Engine library
from lwa_f import snap2_fengine

# Instantiate a Snap2Fengine instance to a board with
# hostname 'snap2-rev2-11'
f = snap2_fengine.Snap2Fengine('snap2-rev2-11')

# Program a board (if it is not already programmed)
# and initialize all the firmware blocks
if not f.fpga.is_programmed():
  f.program() # Load whatever firmware is in flash
  # Wait 30 seconds for the board to reboot...
  # Initialize firmware blocks, including ADC link training
  f.initialize(read_only=False)

# Blocks are available as items in the Snap2Fengine `blocks`
# dictionary, or can be accessed directly as attributes
# of the Snap2Fengine.

# Print available block names
print(sorted(f.blocks.keys()))
# Returns:
# ['adc', 'autocorr', 'corr', 'delay', 'eq', 'eq_tvg', 'eth',
# 'fpga', 'input', 'noise', 'packetizer', 'pfb', 'reorder', 'sync']

# Grab some ADC data from the ADC card(s) on FMC 1
adc_data = f.adc.get_snapshot_interleaved(1, signed=True)
print(adc_data.shape) # returns (32 [channels], 512 [time samples]
```

Details of the methods provided by individual blocks are given in the next section.

4.2.1 Top-Level Control

The Top-level `Snap2Fengine` instance can be used to perform high-level control of the firmware, such as programming and de-programming FPGA boards. It can also be used to apply configurations which affect multiple firmware subsystems, such as configuring channel selection and packet destination.

Finally, a `Snap2Fengine` instance can be used to initialize, or get status from, all underlying firmware modules.

class lwa_f.snap2_fengine.**Snap2Fengine**(*host*, *logger=None*)

A control class for LWA352's SNAP2 F-Engine firmware.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.

cold_start(*program=True*, *initialize=True*, *test_vectors=False*, *sync=True*, *sw_sync=False*, *enable_pfb=True*, *enable_eth=True*, *fft_shift=None*, *eq_coeffs=None*, *chans_per_packet=96*, *first_stand_index=0*, *nstand=32*, *macs={}*, *source_ip='10.41.0.101'*, *source_port=10000*, *dests=[]*)

Completely configure a SNAP2 F-engine from scratch.

Parameters

- **program** (*bool*) – If True, start by programming the SNAP2 FPGA from the image currently in flash. Also train the ADC-> FPGA links and initialize all firmware blocks. Also relock the SNAP2's internal timekeeping logic to the sync pulse (and PPS, if connected) distribution system.
- **initialize** (*bool*) – If True, put all firmware blocks in their default initial state, and relock the board's timekeeping logic to the sync pulse (and PPS, if connected) distribution system. Initialization is always performed if the FPGA has been reprogrammed, but can be run without reprogramming to (quickly) reset the firmware to a known state. Initialization does not include ADC->FPGA link training.
- **test_vectors** (*bool*) – If True, put the F-engine in "frequency ramp" test mode.
- **sync** (*bool*) – If True, synchronize (i.e., reset) the DSP pipeline.
- **sw_sync** (*bool*) – If True, issue a software reset trigger, rather than waiting for an external reset pulse to be received over SMA.
- **enable_pfb** – If True, enable the PFB FIR filter on the F-Engine.
- **enable_eth** (*bool*) – If True, enable 40G F-Engine Ethernet output.
- **fft_shift** (*int*) – If provided, set the F-engine FFT shift to the provided value.
- **eq_coeffs** (*list*) – If provided, the list of pre-quantization equalization coefficients to be loaded to F-engines. This should have *eq.n_coeffs* entries.
- **chans_per_packet** (*int*) – Number of frequency channels in each output F-engine packet
- **first_stand_index** (*int*) – Zero-indexed ID of the first stand connected to this board.
- **nstand** (*int*) – Number of stands to be sent. Values of `n*32` may be used to spood F-engine packets from multiple SNAP2 boards.
- **start_chan** (*int*) – First frequency channel to send to X-engines. Should be an integer multiple of 16.
- **macs** (*dict*) – Dictionary, keyed by dotted-quad string IP addresses, containing MAC addresses for F-engine packet destinations. I.e., IP/MAC pairs for all X-engines.
- **source_ip** (*str*) – The IP address from which this board should send packets.
- **source_port** (*int*) – The source UDP port from which F-engine packets should be sent.
- **dests** (*List of dict*) – List of dictionaries describing where packets should be sent. Each list entry should have the following keys:
  - 'ip' : The destination IP (as a dotted-quad string) to which packets should be sent.
  - 'port' : The destination UDP port to which packets should be sent.
  - 'start_chan' : The first frequency channel number which should be sent to this IP / port. `start_chan` should be an integer multiple of 16.
  - 'nchans' : The number of channels which should be sent to this IP / port. `nchans` should be a multiple of `chans_per_packet`.

cold_start_from_config(*config_file*, *program=True*, *initialize=True*, *test_vectors=False*, *sync=True*, *sw_sync=False*, *enable_eth=True*)

Completely configure a SNAP2 F-engine from scratch, using a configuration YAML file.

Parameters

- **program** (*bool*) – If True, start by programming the SNAP2 FPGA from the image currently in flash. Also train the ADC-> FPGA links and initialize all firmware blocks. Also relock the SNAP2's internal timekeeping logic to the sync pulse (and PPS, if connected) distribution system.
- **initialize** (*bool*) – If True, put all firmware blocks in their default initial state, and relock the board's timekeeping logic to the sync pulse (and PPS, if connected) distribution system. Initialization is always performed if the FPGA has been reprogrammed, but can be run without reprogramming to (quickly) reset the firmware to a known state. Initialization does not include ADC->FPGA link training.
- **test_vectors** (*bool*) – If True, put the F-engine in "frequency ramp" test mode.
- **sync** (*bool*) – If True, synchronize (i.e., reset) the DSP pipeline.
- **sw_sync** (*bool*) – If True, issue a software reset trigger, rather than waiting for an external reset pulse to be received over SMA.
- **enable_eth** (*bool*) – If True, enable 40G F-Engine Ethernet output.
- **config_file** (*str*) – Path to a configuration YAML file.

configure_output(*antenna_ids*, *n_chans_per_packet*, *n_chans_per_xeng*, *chans*, *ips*, *ports=None*, *debug=False*)

Configure channel reordering and packetizer modules to emit a selection of frequency channels.

Parameters

- **n_chans_per_packet** (*int*) – Number of channels per packet.
- **n_chans_per_xeng** (*int*) – Number of channels per xeng.
- **chans** (*list of int*) – A list of channel indices to be sent, e.g. `range(0, 1024)`. The number of channels in this list should be an integer multiple of `n_chans_per_packet`. An assertion error is raised if this is not the case.
- **ips** (*list of str*) – A list of IP addresses to which packets should be sent. The order of values in `ips` and `chans` should reflect where different channels should be sent. The `n` th IP address in `ips` is the destination to which channels `chans[n*n_chans_per_packet : (n+1)*n_chans_per_packet]` should be sent. As such, `ips` should have `len(chans) // n_chans_per_packet` elements. IP addresses should be provided in dotted-quad string representation.
- **ports** (*list of int*) – The UDP destination ports to which packets should be transmitted. Addressing rules are the same as for `ips`. If None, all packets are transmitted to UDP port 10000.
- **antenna_ids** (*list*) – A list of Antenna IDs which should be written to output packet headers. Addressing rules are the same as for `ips`.
- **debug** (*bool*) – Set to True to print extra diagnostic information.
- **n_boards** – Transmit packets with headers indicating they came from this many different boards. For example, if n_boards=2, the board will transmit two streams of packets with antenna IDs `base_ant` and `base_ant+64`

deprogram()

Reprogram the FPGA into its default boot image.

get_status_all()
Call the `get_status` methods of all blocks in `self.blocks`. If the FPGA is not programmed with F-engine firmware, will only return basic FPGA status.

Returns (status_dict, flags_dict) tuple. Each is a dictionary, keyed by the names of the blocks in `self.blocks`. These dictionaries contain, respectively, the status and flags returned by the `get_status` calls of each of this F-Engine's blocks.

hostname
hostname of the F-Engine's host SNAP2 board

initialize(*read_only=True*)
Call the `initialize methods of all underlying blocks, then optionally issue a software global reset.

Parameters **read_only** (*bool*) – If True, call the underlying initialization methods in a read_only manner, and skip software reset.

is_connected()

Returns True if there is a working connection to a SNAP2. False otherwise.

Return type bool

logger
Python Logger instance

n_signals_per_board = 64
Number of analog inputs per FPGA

n_signals_per_xeng = 704
Number of analog inputs per X-engine

print_status_all(*use_color=True*, *ignore_ok=False*)
Print the status returned by `get_status` for all blocks in the system. If the FPGA is not programmed with F-engine firmware, will only print basic FPGA status.

Parameters

- **use_color** (*bool*) – If True, highlight values with colors based on error codes.
- **ignore_ok** (*bool*) – If True, only print status values which are outside the normal range.

program(*fpgfile=None*, *force=False*)
Program an .fpg file to a SNAP2 FPGA. If the name of the file matches what is already in flash, this command will simply reboot the FPGA. If the name of the file doesn't match, the new bitstream will be uploaded. This will take <=5 minutes.

Parameters

- **fpgfile** (*str*) – The .fpg file to be loaded. Should be a path to a valid .fpg file. If None is given, the image currently in flash will be loaded.
- **force** (*boolean*) – If True, write the firmware to flash even if the SNAP claims it is already loaded. Has no effect if `fpgfile=None`.

set_equalization(*eq_start_chan=1000*, *eq_stop_chan=3300*, *start_chan=512*, *stop_chan=3584*, *filter_ksize=21*, *target_rms=0.375*)
Set the equalization coefficients to realize a target RMS.

Parameters

- **eq_start_chan** (*int*) – Frequency channels below `eq_start_chan` will be given the same EQ coefficient as `eq_start_chan`.

- **eq_stop_chan** (*int*) – Frequency channels above `eq_stop_chan` will be given the same EQ coefficient as `eq_stop_chan`.
- **start_chan** (*int*) – Frequency channels below `start_chan` will be given zero EQ coefficients.
- **stop_chan** (*int*) – Frequency channels above `stop_chan` will be given zero EQ coefficients.
- **filter_ksize** (*int*) – Filter kernel size, for rudimentary RFI removal. This should be an odd value.
- **target_rms** (*float*) – The target post-EQ RMS. This is normalized such that 0.875 is the saturation level. I.e., an RMS of 0.125 means that the RMS is one LSB of a 4-bit signed signal.

4.2.2 FPGA Control

The `FPGA` control interface allows gathering of FPGA statistics such as temperature and voltage levels. Its methods are functional regardless of whether the FPGA is programmed with an LWA F-Engine firmware design.

class `lwa_f.blocks.fpga.`**Fpga**(*host*, *name*, *logger=None*)

Instantiate a control interface for top-level FPGA control.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.

get_build_time()

Read the UNIX time at which the current firmware was built.

Return build_time Seconds since the UNIX epoch at which the running firmware was built.

Rtype int

get_firmware_version()

Read the firmware version register and return the contents as a string.

Return version major_version.minor_version.revision.bugfix

Rtype str

get_fpga_clock()

Estimate the FPGA clock, by polling the `sys_clkcounter` register.

Returns Estimated FPGA clock in MHz

Return type float

get_status()

Get status and error flag dictionaries.

Status keys:

- programmed (bool) : `True` if FPGA appears to be running DSP firmware. `False` otherwise, and flagged as a warning.
- flash_firmware (str) : The name of the firmware file currently loaded in flash memory.
- flash_firmware_md5 (str) : The MD5 checksum of the firmware file currently loaded in flash memory.

- timestamp (str) : The current time, as an ISO format string.
- serial (str) : The serial number / identifier for this board. Flagged with a warning if no serial is available.
- host (str) : The host name of this board.
- sw_version (str) : The version string of the control software package. Flagged as warning if the version indicates a build against a dirty git repository.
- fw_version (str): The version string of the currently running firmware. Available only if the board is programmed.
- fw_build_time (int): The build time of the firmware, as an ISO format string. Available only if the board is programmed.
- sys_mon (str) : `'reporting'` if the current firmware has a functioning system monitor module. Otherwise `'not reporting'`, flagged as an error.
- temp (float) : FPGA junction temperature, in degrees C. (Only reported is system monitor is available). Flagged as a warning if outside the recommended operating conditions. Flagged as an error if outside the absolute maximum ratings. See DS892.
- vccaux (float) : Voltage of the VCCAUX FPGA power rail. (Only reported is system monitor is available). Flagged as a warning if outside the recommended operating conditions. Flagged as an error if outside the absolute maximum ratings. See DS892.
- vccbram (float) : Voltage of the VCCBRAM FPGA power rail. (Only reported is system monitor is available). Flagged as a warning if outside the recommended operating conditions. Flagged as an error if outside the absolute maximum ratings. See DS892.
- vccint (float) : Voltage of the VCCINT FPGA power rail. (Only reported is system monitor is available). Flagged as a warning if outside the recommended operating conditions. Flagged as an error if outside the absolute maximum ratings. See DS892.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

is_programmed()
Lazy check to see if a board is programmed. Check for the "version_version" register. If it exists, the board is deemed programmed.

Returns True if programmed, False otherwise.

Return type bool

4.2.3 Power Monitoring

The `PowerMon` interface allows gathering of power supply statistics such as voltage and currnt levels.

class lwa_f.blocks.powermon.**PowerMon**(*host*, *name*, *logger=None*)
Instantiate a Power Monitor control block.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.

- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.

get_status()
Get status and error flag dictionaries.

Status keys:

- <rail_name>_voltage (float) : Voltage measured on <rail_name> in units of Volts.
- <rail_name>_current(float) : Current draw measured on <rail_name> in units of Amps.

Currents are flagged as warnings if they exceed 80% of the allowed maximum, and errors if they exceed the allowed maximum.

Voltages are flagged as errors if they are not within 3% of the expected level.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)
Initialize block.

Parameters **read_only** (*bool*) – If False, configure I2C interface. If True, instantiate I2C control object and just try to use it.

4.2.4 Timing Control

The `Sync` control interface provides an interface to configure and monitor the multi-SNAP2 timing distribution system.

class lwa_f.blocks.sync.**Sync**(*host*, *name*, *logger=None*)

arm_noise()
Arm noise generator resets.

arm_sync()
Arm sync pulse generator, which passes sync pulses to the design DSP.

count_ext()

Returns Number of external sync pulses received from the timing distribution system.

Rtype int

count_int()

Returns Number of internally generated sync pulses counted. The "internal" pulses are output on a physical board connector. For one board, this signal drives the timing distribution system.

Rtype int

count_pps()

Returns Number of external PPS pulses received. This counter will only increment reliably on the single board which has a PPS connected.

Rtype int

disable_loopback()
Disable the internal loopback between the sync output and input.

enable_loopback()
Internally loop back the sync output and input.

get_latency()

Returns Number of FPGA clock ticks between sync transmission and reception. This measurement is only meaningful for the board which has its internal pulse output connected to the timing distribution system input.

Rtype int

get_latency_variations()

Returns Number of latency variations between sync transmission and reception since reset_error_count

Rtype int

get_period_variations()

Returns The number of sync period variations in pulses received from the timing distribution system since last reset_error_count()

Rtype int

get_pps_period_variations()

Returns The number of PPS period variations since last reset_error_count(). This is only meaninful if a board has a PPS input connected.

Rtype int

get_status()
Get status and error flag dictionaries.

Status keys:

- uptime_fpga_clks (int) : Number of FPGA clock ticks (= ADC clock ticks) since the FPGA was last programmed.
- period_fpga_clks (int) : Number of FPGA clock ticks (= ADC clock ticks) between the last two internal sync pulses.
- period_variations (int) : Number of different external sync periods measured since the last error count reset. Any value other than zero is flagged as a warning.
- period_pps_fpga_clks (int) : Number of FPGA clock ticks (= ADC clock ticks) between the last two external PPS sync pulses.
- ext_count (int) : The number of external sync pulses since the FPGA was last programmed.
- int_count (int) : The number of internal sync pulses since the FPGA was last programmed.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

get_tt_of_pps(*wait_for_sync=True*)
Get the internal TT at which the last PPS pulse arrived, optionally waiting for a pulse to pass before reading its arrival time and returning.

Parameters **wait_for_sync** (*bool*) – If True, wait for a sync pulse to pass before measuring its arrival time and returning.

Returns (tt, sync_number). `tt` is the internal TT of the last PPS. `sync_number` is the PPS pulse count corresponding to this TT.

Rtype int

get_tt_of_sync(*wait_for_sync=True*)
Get the internal TT at which the last sync pulse arrived, optionally waiting for a pulse to pass before reading its arrival time and returning.

Parameters **wait_for_sync** (*bool*) – If True, wait for a sync pulse to pass before measuring its arrival time and returning.

Returns (tt, sync_number). `tt` is the internal TT of the last sync. `sync_number` is the sync pulse count corresponding to this TT.

Rtype int

initialize(*read_only=False*)
Initialize block.

Parameters **read_only** (*bool*) – If False, initialize system control register to 0 and reset error counters. If True, do nothing.

load_internal_time(*tt*, *software_load=False*)
Load a new starting value into the _internal_ telescope time counter on the next sync.

Parameters

- **tt** (*int*) – Telescope time to load
- **software_load** (*bool*) – If True, immediately load via a software trigger. Else load on the next external sync pulse arrival.

load_telescope_time(*tt*, *software_load=False*)
Load a new starting value into the telescope time counter on the next PPS.

Parameters

- **tt** (*int*) – Telescope time to load
- **software_load** (*bool*) – If True, immediately load via a software trigger. Else load on the next PPS arrival.

period()

Returns The number of FPGA clock ticks between the last two external sync pulses received from the timing distribution system.

Rtype int

period_pps()

Returns The number of FPGA clock ticks between the last two PPS pulses. This period report will only be meaningful is this board has a PPS connected.

Rtype int

reset_error_count()
Reset error counters to 0.

reset_telescope_time()
Reset the telescope time counter to 0 immediately.

set_output_sync_rate(*mask*)
Set the output sync generation rate. A sync is issued when the lower 32-bits of the telescope time counter, masked with `~mask == 0`. I.e., a mask of 0 will cause a sync every 2^32 clock cycles. A mask of 0xffff0000 will create an output pulse every 2^16 clock cycles. Output sync pulses are extended by 256 clocks, so the output pulse rate should be lower than this.

Parameters **mask** (*int*) – Mask with which to bitwise AND the telescope time counter before comparing to 0.

sw_sync()
Issue a sync pulse from software. This will only do anything if appropriate arming commands have been made in advance.

update_internal_time(*fs_hz=196000000*)
Load the sync-pulse-locked telescope time counters with the correct time on the next sync pulse. Since sync pulses are derived from the telescope time of the one SNAP board which drives the timing distribution network, `update_telescope_time()` should have been run on this unique board prior to the use of `update_internal_time`.

Loading procedure is:

1. Wait for a sync pulse to pass
2. Compute how many, `m`, sync periods (determined by `period()`) have passed since UNIX time 0, inferring the exact arrival time of the last sync by comparison with system time, which is assumed to be aligned to GPS time to better than 50% of a sync pulse period.
3. Compute the telescope time (`(m+1)*period`) of the next expected sync pulse arrival.
4. Load this value on the next sync pulse using `load_internal_time`
5. Verify (using `count_ext`) that no sync pulses have occurred while performing steps 2 and 3. Generate an error if this is not the case.

Parameters **fs_hz** (*int*) – The ADC clock rate, in Hz. Used to set the telescope time counter.

update_telescope_time(*fs_hz=196000000*)
Load the PPS-locked telescope time counters with the correct time on the next PPS pulse.

Loading procedure is:

1. Wait for a PPS to pass, or for a timeout waiting for a PPS
2. If no PPS was detected. Do nothing and return from this function, skipping steps 3,4,5
3. Inferring the exact time of the observed PPS arrival via current system time, which is assumed to be aligned to GPS time to better than 0.5 seconds, compute how many ADC clocks will have occurred at the time of the upcoming PPS.
4. Load this value on the next PPS pulse using `load_telescope_time`
5. Verify (using `count_pps`) that no PPS pulses have occurred while performing steps 2 and 3. Generate an error if this is not the case.

Parameters **fs_hz** (*int*) – The ADC clock rate, in Hz. Used to set the telescope time counter.

uptime()

Returns Time in FPGA clock ticks since the FPGA was last programmed. Resolution is 2**32 (21 seconds at 200 MHz)

Return type int

wait_for_pps (*timeout=2.0*)
Block until a PPS has been received.

Parameters **timeout** (*float*) – Timeout, in seconds, to wait.

Returns least-significant 32-bits of telescope time of last PPS pulse. Or, -1, on timeout.

Rtype int

wait_for_sync (*timeout=20*)
Block until a sync has been received.

Parameters **timeout** (*float*) – Timeout, in seconds, to wait.

4.2.5 ADC Control

The `Adc` control interface allows link training (aka "calibration") of the ADC->FPGA data link.

class `lwa_f.blocks.adc.`**Adc** (*host*, *name*, *logger=None*)
Instantiate a control interface for an ADC block.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.

calibrate (*use_ramp=False*, *fail_hard=True*, *step_size=8*, *verbose=False*)
Compute and set all ADC data lane input delays to their optimal values. After this call, the ADCs are left in test mode.

Parameters

- **use_ramp** (*bool*) – If True, after calibration, use the ramp test pattern to verify the ADC->FPGA link is functioning correctly. If False, perform this verification with the same constant test value used for the calibration procedure.
- **fail_hard** (*bool*) – If True, raise an exception if the calibration run doesn't go to plan.
- **verbose** (*bool*) – If True, print alignment information after each scan.
- **step_size** (*int*) – Number of IDELAY tap steps between error counts.

Returns (status, delays, slack) tuple. `status` is True if the calibration procedure succeeded. False otherwise. `delays` is an N_ADC_BOARD x N_ADC_CHIP x DATA_LANES_PER_CHIP array of chosen delays. `slack` is an N_ADC_BOARD x N_ADC_CHIP x DATA_LANES_PER_CHIP array of slacks – the number of delay taps between the chosen delay and the nearest delay which showed ADC errors.

Return type bool, list, list

get_snapshot (*fmc*, *signed=False*, *trigger=True*)
Read a snapshot of data from all ADC channels on a single FMC card. Return data without interleaving ADC cores.

Parameters

- **fmc** (*int*) – Which FMC port (0 or 1) to read.
- **signed** (*bool*) – If True, return data interpretted as signed 2's complement integers. If False, return data as unsigned integers.

- **trigger** (*bool*) – If True, trigger a new simultaneous capture of data from all channels. If False, read existing data capture. Grabbing data without a new trigger may be useful if you wish to read channels from a second FMC port which were captured simultaneously with another port.

Returns Array of captured data with dimensions [ADC_CHIPS_PER_FMC, ADC_LANES, TIME_SAMPLES]. Data from ADC lanes representing the same analog input are _not_ interleaved. Data from ADC lanes n,n+1 are associated with the same analog input.

Return type numpy.array

get_snapshot_interleaved(*fmc*, *signed=False*, *trigger=True*)

Read a snapshot of data from all ADC channels on a single FMC card. Return data with ADC cores interleaved.

Parameters

- **fmc** (*int*) – Which FMC port (0 or 1) to read.
- **signed** (*bool*) – If True, return data interpretted as signed 2's complement integers. If False, return data as unsigned integers.
- **trigger** (*bool*) – If True, trigger a new simultaneous capture of data from all channels. If False, read existing data capture. Grabbing data without a new trigger may be useful if you wish to read channels from a second FMC port which were captured simultaneously with another port.

Returns numpy array of captured data with dimensions [ADC_CHANNELS_PER_FMC, TIME_SAMPLES].

Return type numpy.ndarray

initialize(*read_only=False*, *clocksource=1*, *fail_hard=False*)

Initialize connected ADC boards.

Parameters

- **read_only** (*bool*) – If True, don't do anything which would affect a running system. If False, train ADC->FPGA data links.
- **clocksource** (*int*) – Which ADC board (0 or 1) on an FMC card should serve as the source of the clocks. Note that while this parameter is set for boards on all FMC cards, only the FMC card selected as the clock source at Simulink compile-time will be used for clocking.
- **fail_hard** (*Bool*) – If True, raise RuntimeError if initialization fails.

Returns True if initialization was successful. False otherwise.

mmcm_is_locked()

Read the ADC control register to determine if the clock PLLs are locked.

Returns True if the ADC clocks are locked. False otherwise.

Return type bool

print_sweep(*errs*, *best_delays=None*, *step_size=8*)

Print, using ASCII, the valid data capture eye as a function of delay setting. Delays are printed such that one row represents a sweep of delays for a single ADC data lane. Each column in the row is `X` if data contained errors at this delay, `-` if no errors were detected at this delay, and `|`, if this delay is considered the best setting in the sweep range.

Parameters

- **errs** (*list*) – Array of error counts with dimensions [DELAY_TRIALS, ADC_CHIPS, DATA_LANES_PER_ADC_CHIP] such as that returned by `_get_errs_by_delay`.
- **best_delays** (*list*) – Array of best delays, with shape [ADC_CHIPS, DATA_LANES_PER_ADC_CHIP], such as that returned by `_get_best_delays`. These delays are marked with an ASCII `|`.
- **step_size** (*int*) – Number of IDELAY tap steps between delay trials.

reset()
Toggle the ADC reset input.

sync()
Toggle the ADC sync input.

use_data()
Set all ADCs into normal operating mode, in which analog inputs are digitized and transmitted.

use_ramp()
Set all ADCs into the "ramp" test mode, in which ADC samples are replaced with a 10-bit counter which increments with each ADC clock.

use_toggle(*val0*, *val1*)
Set all ADCs into the "toggle" test mode, in which ADC samples are replaced with a toggling pattern of two 10-bit values.

Parameters

- **val0** (*int*) – First value ADCs should output.
- **val1** (*int*) – Second value ADCs should output.

4.2.6 Input Control

class lwa_f.blocks.input.**Input**(*host*, *name*, *n_streams=64*, *n_bits=10*, *logger=None*)
Instantiate a control interface for an Input block. This block allows switching data streams between constant-zeros, digital noise, and ADC streams.

A statistics interface is also provided, providing bit statistics and histograms.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **n_streams** (*int*) – Number of independent streams which may be delayed
- **n_bits** (*int*) – Number of bits per ADC sample.

Variables

- **n_streams** – Number of streams this interface handles
- **n_bits** – Number of bits per ADC sample

get_all_histograms()
Get histograms for all signals, summing over all interleaving cores.

Returns (vals, hists). `vals` is a list of histogram bin centers. `hists` is an `[n_stream x 2**n_bits]` list of histogram data.

get_bit_stats()
Get the mean, RMS, and mean powers of all ADC streams.

Returns (means, powers, rmss) tuple. Each member of the tuple is an array with `self.n_streams` elements.

Rval (numpy.ndarray, numpy.ndarray, numpy.ndarray)

get_histogram(*stream*, *sum_cores=True*)
Get a histogram for an ADC stream.

Parameters

- **stream** (*int*) – ADC stream from which to get data.
- **sum_cores** (*bool*) – If True, compute one histogram from both pairs of interleaved ADC cores associated with an analog input. If False, compute separate histograms.

Returns If `sum_cores` is True, return `(vals, hist)`, where `vals` is a list of histogram bin centers, and `hist` is a list of histogram data points. If `sum_cores` is False, return `(vals, hist_a, hist_b)`, where `hist_a` and `hist_b` are separate histogram data sets for the even-sample and odd-sample ADC cores, respectively.

get_status()
Get status and error flag dictionaries.

Status keys:

- switch_position<n> (str) : Switch position ('noise', 'adc' or 'zero') for input stream `n`, where `n` is a two-digit integer starting at 00. Any input position other than 'adc' is flagged with "NOTIFY".
- power<n> (float) : Mean power of input stream `n`, where `n` is a two-digit integer starting at 00. In units of (ADC LSBs)**2.
- rms<n> (float) : RMS of input stream `n`, where `n` is a two-digit integer starting at 00. In units of ADC LSBs. Value is flagged as a warning if it is >30 or <5.
- mean<n> (float) : Mean sample value of input stream `n`, where `n` is a two-digit integer starting at 00. In units of ADC LSBs. Value is flagged as a warning if it is > 2.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

get_switch_positions()
Get the positions of the input switches.

Returns List of switch positions. Entry `n` contains the position of the switch associated with ADC input `n`. Switch positions are "noise" (internal digital noise generators), "adc" (digitized ADC stream), or "zero" (constant 0).

Return type list of str

initialize(*read_only=False*)
Initialize the block.

Parameters **read_only** (*bool*) – If True, do nothing. If False, set the input multiplexers to ADC data and enable statistic computation.

plot_histogram(*stream*)
Plot a histogram.

Parameters **stream** (*int*) – ADC stream from which to get data.

print_histograms()
Print histogram stats to screen.

use_adc(*stream=None*)
Switch input to ADC.

Parameters **stream** (*int or None*) – Which stream to switch. If None, switch all.

use_noise(*stream=None*)
Switch input to internal noise source.

Parameters **stream** (*int or None*) – Which stream to switch. If None, switch all.

use_zero(*stream=None*)
Switch input to zeros.

Parameters **stream** (*int or None*) – Which stream to switch. If None, switch all.

4.2.7 Noise Generator Control

class lwa_f.blocks.noisegen.**NoiseGen**(*host*, *name*, *n_noise=5*, *n_outputs=64*, *logger=None*)
Noise Generator controller

This block controls a digital noise source, which can generate multiple independent channels of gaussian noise. These channels can be assigned to multiple outputs of this block, to create correlated or uncorrelated noise streams.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **n_noise** (*int*) – The number of independent noise generation cores in the underlying block. 2*n_noise independent noise streams will be produced.
- **n_outputs** (*int*) – The number of output channels from the block.

assign_output(*output*, *noise*)
Assign an output channel to a given noise stream. Note that the output stream will not be affected unless the downstream input multiplexors are set to noise mode.

Parameters

- **output** (*int*) – The index of the output stream to be assigned.
- **noise** (*int*) – The index of the noise stream to assign to *output*. Note that each noise generator core generates two independent streams, so *noise* can be in range(0, 2*self.n_noise)

get_output_assignment(*output*)
Get the index of the noise stream assigned to an output.

Parameters **output** (*int*) – The index of the output stream to query.

Returns The index of the noise stream to assign to *output*. Note that each noise generator core generates two independent streams, so *noise* can be in range(0, 2*self.n_noise)

Return type int

get_seed(*n*)
Get the seed of a noise generator.

Parameters **n** (*int*) – Noise generator ID whose seed to read.

Returns Noise generator seed.

Rtype int

get_status()
Get status and error flag dictionaries.

Status keys:

- noise_core<m>_seed (int): Seed currently loaded into noise generator core m. m should be a two-digit integer starting at 00.
- output_assignment<n> (int): The noise generator ID currently assigned to output stream n, where n is a two-digit integer starting at 00.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)
Initialize the block

Parameters **read_only** (*bool*) – If False, set the seen of noise generator n to n. If True, do nothing.

set_seed(*n*, *seed*)
Set the seed of a noise generator.

Parameters

- **n** (*int*) – Noise generator ID to seed.
- **seed** (*int*) – Noise generator seed to load.

4.2.8 Delay Control

class lwa_f.blocks.delay.**Delay**(*host*, *name*, *n_streams=64*, *logger=None*)
Instantiate a control interface for a Delay block.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **n_streams** (*int*) – Number of independent streams which may be delayed

MIN_DELAY = 0
minimum delay allowed

get_delay(*stream*)
Get the current delay for a given input.

Parameters **stream** (*int*) – Which ADC input index to query

Returns Currently loaded delay, in ADC samples

Return type int

get_max_delay()
Query the firmware to get the maximum delay it supports.

Returns Maximum supported delay, in ADC samples

Return type int

get_status()
Get status and error flag dictionaries.

Status keys:

- delay<n>: Currently loaded delay for ADC input index n. in units of ADC samples.
- max_delay: The maximum delay supported by the firmware.
- min_delay: The minimum delay supported by the firmware.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)
Initialize all delays.

Parameters **read_only** (*bool*) – If True, do nothing. If False, initialize all delays to the minimum allowed value.

set_delay(*stream*, *delay*)
Set the delay for a given input stream.

Parameters

- **stream** (*int*) – ADC stream index to which delay should be applied.
- **delay** (*int*) – Number of ADC clock cycles delay to load.

4.2.9 PFB Control

class lwa_f.blocks.pfb.**Pfb**(*host*, *name*, *logger=None*)

fir_disable()
Disable the PFB's FIR frontend to leave a simple FFT.

fir_enable()
Enable the PFB's FIR frontend.

fir_is_enabled()
Query whether the PFB FIR frontend is enabled or not.

Return enabled True if the FIR is enabled, else False.

Rtype bool

get_fft_shift()
Get the currently applied FFT shift schedule. The returned value takes into account any hardcoding of the shift settings by firmware.

Returns Shift schedule

Return type int

get_overflow_count()
Get the total number of FFT overflow events, since the last statistics reset.

Returns Number of overflows

Return type int

get_status()
Get status and error flag dictionaries.

Status keys:

- overflow_count (int) : Number of FFT overflow events since last statistics reset. Any non-zero value is flagged with "WARNING".
- fft_shift (str) : Currently loaded FFT shift schedule, formatted as a binary string, prefixed with "0b".
- fir_enabled (bool) : True if the PFB FIR is enabled. False otherwise. If the FIR is disabled this is flagged as "NOTIFY"

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)

Parameters **read_only** (*bool*) – If False, enable the PFB FIR, set the FFT shift to the default value, and reset the overflow count. If True, do nothing.

rst_stats()
Reset overflow event counters.

set_fft_shift(*shift*)
Set the FFT shift schedule.

Parameters **shift** (*int*) – Shift schedule to be applied.

4.2.10 Auto-correlation Control

class lwa_f.blocks.autocorr.**AutoCorr**(*host*, *name*, *acc_len=32768*, *logger=None*, *n_chans=4096*, *n_signals=64*, *n_parallel_streams=8*, *n_cores=4*, *use_mux=True*)
Instantiate a control interface for an Auto-Correlation block. This provides auto-correlation spectra of post-FFT data.

In order to save FPGA resourece, the auto-correlation block may use a single correlation core to compute the auto-correlation of a subset of the total number of ADC channels at any given time. This is the case when the block is instantiated with `n_cores > 1` and `use_mux=True`. In this case, auto-correlation spectra are captured `n_signals / n_cores` channels at a time.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.

- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **acc_len** (*int*) – Accumulation length initialization value, in spectra.
- **n_chans** (*int*) – Number of frequency channels.
- **n_signals** (*int*) – Number of individual data streams.
- **n_parallel_streams** (*int*) – Number of streams processed by the firmware module in parallel.
- **n_cores** (*int*) – Number of accumulation cores in firmware design.
- **use_mux** (*bool*) – If True, only one core is instantiated and a multiplexer is used to switch different inputs into it. If False, multiple cores are instantiated simultaneously in firmware.

Variables **n_signals_per_block** – Number of signal streams handled by a single correlation core.

get_acc_cnt()
Get the current accumulation count.

Return count Current accumulation count

Rtype count int

get_acc_len()
Get the currently loaded accumulation length in units of spectra.

Returns Current accumulation length

Return type int

get_new_spectra(*signal_block=0*, *flush_vacc='auto'*, *filter_ksize=None*)
Get a new average power spectra.

Parameters

- **signal_block** (*int*) – If using multiplexing, read data for this signal block. If not using multiplexing, this parameter does nothing, and data from all inputs will be returned. When multiplexing, Each call will return data for inputs `self.n_signals_per_block x signal_block` to `self.n_signals_per_block x (signal_block+1) - 1`.
- **flush_vacc** (*Bool or string*) – If `True`, throw away a spectra before grabbing a valid one. This can be useful if the upstream analog settings may have changed during the last integration. If `False`, return the first spectra available. If `'auto'` perform a flush if the input multiplexer has changed positions.
- **filter_ksize** (*int*) – If not None, apply a spectral median filter with this kernel size. The kernet size should be odd.

Returns Float32 array of dimensions [POLARIZATION, FREQUENCY CHANNEL] containing autocorrelations with accumulation length divided out.

Return type numpy.array

get_status()
Get status and error flag dictionaries.

Status keys:

- acc_len (int) : Currently loaded accumulation length in number of spectra.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)
Initialize the block, setting (or reading) the accumulation length.

Parameters **read_only** (*bool*) – If False, set the accumulation length to the value provided when this block was instantiated. If True, use whatever accumulation length is currently loaded.

plot_all_spectra(*db=True*, *show=True*, *filter_ksize=None*)
Plot the spectra of all signals, with accumulation length divided out

Parameters

- **db** (*bool*) – If True, plot 10log10(power). Else, plot linear.
- **show** (*bool*) – If True, call matplotlib's *show* after plotting
- **filter_ksize** (*int*) – If not None, apply a spectral median filter with this kernel size. The kernet size should be odd.

Returns matplotlib.Figure

plot_spectra(*signal_block=0*, *db=True*, *show=True*, *filter_ksize=None*)
Plot the spectra of all signals in a single signal_block, with accumulation length divided out

Parameters

- **signal_block** (*int*) – If using multiplexing, plot data for this signal block. If not using multiplexing, this parameter does nothing, and data from all inputs will be plotted. When multiplexing, Each call will plot data for inputs `self.n_signals_per_block x signal_block` to `self.n_signals_per_block x (signal_block+1) - 1`.
- **db** (*bool*) – If True, plot 10log10(power). Else, plot linear.
- **show** (*bool*) – If True, call matplotlib's *show* after plotting
- **filter_ksize** (*int*) – If not None, apply a spectral median filter with this kernel size. The kernet size should be odd.

Returns matplotlib.Figure

set_acc_len(*acc_len*)
Set the number of spectra to accumulate.

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

4.2.11 Correlation Control

class lwa_f.blocks.corr.**Corr**(*host*, *name*, *acc_len=1024*, *logger=None*, *n_chans=1024*, *n_signals=64*)
Instantiate a control interface for a Correlation block.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.

- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **acc_len** (*int*) – Accumulation length initialization value, in spectra.
- **n_chans** (*int*) – Number of frequency channels in the correlation output.
- **n_signals** (*int*) – Number of independent signals which may be correlated.

get_acc_len()
Get the currently loaded accumulation length in units of spectra.

Returns Current accumulation length

Return type int

get_new_corr(*signal1*, *signal2*, *flush_vacc=True*)
Get a new correlation. Data are returned with summing factor divided out, and normalized to correspond to an input signal with real and imaginary parts each having a range of +/- 0.875

Parameters

- **signal1** (*int*) – First (unconjugated) signal index.
- **signal2** (*int*) – Second (conjugated) signal index.
- **flush_vacc** (*bool*) – If True, throw away the first accumulation after setting the input selection registers. This is good practice the first time a new signal pair is read.

Returns Complex-valued cross-correlation spectra of *signal1* and *signal2* with accumulation length and frequency summing factor divided out.

Return type numpy.array

get_status()
Get status and error flag dictionaries.

Status keys:

- acc_len : Currently loaded accumulation length in number of spectra.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)
Initialize the block, setting (or reading) the accumulation length.

Parameters **read_only** (*bool*) – If False, set the accumulation length to the value provided when this block was instantiated. If True, use whatever accumulation length is currently loaded.

plot_all_spectra(*db=False*, *show=True*)
Plot all auto-correlation spectra, with accumulation length divided out.

Parameters

- **db** (*bool*) – If True, plot 10log10(power). Else, plot linear.
- **show** (*bool*) – If True, call matplotlib's *show* after plotting

Returns matplotlib.Figure

plot_corr(*signal1*, *signal2*, *show=False*)
Plot a correlation spectra, with accumulation length and frequency summing factor divided out, and normalized to correspond to an input signal with real and imaginary parts each having a range of +/- 0.875

Parameters

- **signal1** (*int*) – First (unconjugated) signal index.
- **signal2** (*int*) – Second (conjugated) signal index.
- **show** (*bool*) – If True, call matplotlib's *show* after plotting

Returns matplotlib.Figure

set_acc_len(*acc_len*)
Set the number of spectra to accumulate.

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

4.2.12 Post-FFT Test Vector Control

class lwa_f.blocks.eqtvg.**EqTvg**(*host*, *name*, *n_streams=64*, *n_chans=4096*, *logger=None*)
Instantiate a control interface for a post-equalization test vector generator block.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **n_streams** (*int*) – Number of independent streams which may be delayed
- **n_chans** (*int*) – Number of frequency channels.

get_status()
Get status and error flag dictionaries.

Status keys:

- tvg_enabled: Currently state of test vector generator. `True` if the generator is enabled, else `False`.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)
Initialize the block.

Parameters **read_only** (*bool*) – If True, do nothing. If False, load frequency-ramp test vectors, but disable the test vector generator.

read_stream_tvg(*stream*, *makecomplex=False*)
Read the test vector loaded to an ADC stream.

Parameters

- **stream** (*int*) – Index of stream from which test vectors should be read.

- **makecomplex** (*Bool*) – If True, return an array of 4+4 bit complex numbers, as interpretted by the correlator. If False, return the raw unsigned 8-bit values loaded in FPGA memory.

Returns Test vector array

Return type numpy.ndarray

tvg_disable()
Disable the test vector generator

tvg_enable()
Enable the test vector generator.

tvg_is_enabled()
Query the current test vector generator state.

Returns True if the test vector generator is enabled, else False.

Return type bool

write_const_per_stream()
Write a constant to all the channels of a stream, with stream *i* taking the value *i*.

write_freq_ramp()
Write a frequency ramp to the test vector that is repeated for all ADC inputs. Data are wrapped to fit into 8 bits. I.e., the test vector value for channel 257 takes the value `1`.

write_stream_tvg(*stream*, *test_vector*)
Write a test vector pattern to a single signal stream.

Parameters

- **stream** (*int*) – Index of stream to which test vectors should be loaded.
- **test_vector** (*list or numpy.ndarray*) – *self.n_chans*-element test vector. Values should be representable in 8-bit unsigned integer format. Data are loaded such that the most-significant 4 bits of the test_vectors are interpretted as the 2's complement 4-bit real sample data. The least-significant 4 bits of the test vectors are interpretted as the 2's complement 4-bit imaginary sample data.

4.2.13 Equalization Control

class lwa_f.blocks.eq.**Eq**(*host*, *name*, *n_streams=64*, *n_coeffs=512*, *logger=None*)
Instantiate a control interface for an Equalization block.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **n_streams** (*int*) – Number of independent streams which may be delayed
- **n_coeffs** (*int*) – Number of coefficients per input stream. Coefficients are shared among neighbouring frequency channels.

clip_count()
Get the total number of times any samples have clipped, since last sync.

Returns Clip count.

Return type int

get_coeffs(*stream*, *return_as_int=False*)

Get the coefficients currently loaded. Reads the actual coefficients from the board, returning these either as floats (which may, for example, be modified and then passed back to `set_coeffs`) or as an integers with a scaling factor (which reflects precisely the values stored in the firmware registers).

Parameters

- **stream** (*int*) – ADC stream index to query.
- **return_as_int** (*bool*) – If True, return a tuple containing integer coefficients as stored on the FPGA, and a binary point scale. If False, return a floating point interpretation of the coefficients being applied to data.

Returns If `return_as_int`, return a tuple `(coeffs, binary_point)`. `coeffs` is an array of `self.n_coeffs` coefficients currently being applied. `binary_point` is the position of the binary point with which these integers are scaled on the FPGA. If not `return_as_int`, return `coeffs`, an array of `self.n_coeffs` floating point coefficients.

Return type (numpy.ndarray, int) or numpy.ndarray

get_status()

Get status and error flag dictionaries.

Status keys:

- clip_count: Number of clip events in the last sync period.
- width: Bit width of coefficients
- binary_point: Binary point position of coefficients
- coefficients<n>: The currently loaded, integer-valued coefficients for ADC stream `n`.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)

Initialize block.

Parameters **read_only** (*bool*) – If False, set all coefficients to some nominally sane value. Currently, this is 100.0. If True, do nothing.

plot_all_coefficients(*db=False*)

Plot EQ coefficients from all input paths.

Parameters **db** (*bool*) – If True, plot 10log10(power). Else, plot linear.

set_coeffs(*stream*, *coeffs*)

Set the coefficients for a data stream. Rounding and saturation will be applied before loading, so the provided coefficients may be integer or floating point.

Parameters

- **stream** (*int*) – ADC stream index to which coefficients should be applied.
- **coeffs** (*list or numpy.ndarray*) – Array of coefficients to load. This should be of length `self.n_coeffs`, else an AssertionError will be raised.

4.2.14 Channel Selection Control

class lwa_f.blocks.chanreorder.**ChanReorder**(*host*, *name*, *n_chans=1024*, *logger=None*)
Instantiate a control interface for a Channel Reorder block.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **n_chans** (*int*) – Number of channels in the system.

initialize(*read_only=False*)
Initialize the block.

Parameters **read_only** (*bool*) – If True, this method is a no-op. If False, initialize the block with the identity map. I.e., map channel *n* to channel *n*.

n_parallel_chans = 8
Number of channels per reorder word

read_reorder(*raw=False*)
Read the currently loaded reorder map.

Parameters **raw** (*bool*) – If True, return the map as loaded onto the FPGA. If False, return the resulting channel map – i.e., the channel ordering being output by this F-engine.

Returns The reorder map currently loaded.

Return type list

set_channel_order(*order*)
Re-order channels so that they are sent with the order in the specified map.

There are various requirements of the map which must be met.

- Every integer multiple of *self.n_parallel_chans* of the map must start on an integer multiple of *n_parallel_chans*. Eg., for *n_parallel_chans = 8 order[0] = 16* is acceptable. *order[0] = 4* is not.
- Blocks of *n_parallel_channels* must be consecutive. Eg., if *n_parallel_chans=8*, *order[16:24] = [0,1,2,3,4,5,6,7]* is acceptable. *order[16:24] = [0,1,2,3,8,9,10,11]* is not.
- The provided map must be *self.n_chans* elements long.

Parameters **order** (*list of int*) – The order to which data should be mapped. I.e., if *order[0] = 16*, then the first channel out of the F-engine will be channel 16. The order map should meet the above criteria. A ValueError exception will be raised if they are not.

4.2.15 Packetization Control

class lwa_f.blocks.packetizer.**Packetizer**(*host*, *name*, *n_chans=4096*, *n_signals=64*, *sample_rate_mhz=200.0*, *logger=None*)

The packetizer block allows dynamic definition of packet sizes and contents. In firmware, it is a simple block which allows insertion of header entries and EOFs at any point in the incoming data stream. It is up to the user to configure this block such that it behaves in a reasonable manner – i.e.

- Output data rate does not overflow the downstream Ethernet core
- Packets have a reasonable size
- EOFs and headers are correctly placed.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **n_chans** (*int*) – Number of frequency channels in the correlation output.
- **n_signals** (*int*) – Number of independent analog streams in the system
- **sample_rate_mhz** (*float*) – ADC sample rate in MHz. Used for data rate checks.

get_packet_info(*n_pkt_chans*, *occupation=0.95*, *chan_block_size=8*)

Get the packet boundaries for packets containing a given number of frequency channels.

Parameters

- **n_pkt_chans** (*int*) – The number of channels per packet.
- **occupation** (*float*) – The maximum allowed throughput capacity of the underlying link. The calculation does not include application or protocol overhead, so must necessarily be < 1.
- **chan_block_size** (*int*) – The granularity with which we can start packets. I.e., packets must start on an n*`chan_block` boundary.

Returns

packet_starts, packet_payloads, channel_indices

packet_starts [list of ints] The word indexes where packets start – i.e., where headers should be written. For example, a value [0, 1024, 2048, ...] indicates that headers should be written into underlying brams at addresses 0, 1024, etc.

packet_payloads [list of range()] The range of indices where this packet's payload falls. Eg: [range(1,257), range(1025,1281), range(2049,2305), ... etc] These indices should be marked valid, and the last given an EOF.

channel_indices [list of range()] The range of channel indices this packet will send. Eg: [range(1,129), range(1025,1153), range(2049,2177), ... etc] Channels to be sent should be re-indexed so that they fall into these ranges.

write_config(*packet_starts*, *packet_payloads*, *channel_indices*, *ant_indices*, *dest_ips*, *dest_ports*, *nchans_per_packet*, *nchans_per_xeng*, *n_signals_per_packet*, *n_signals_per_xeng*, *print_config=False*)

Write the packetizer configuration BRAMs with appropriate entries.

Parameters

- **packet_starts** (*list of int*) – Word-indices which are the first entry of a packet and should be populated with headers (see *get_packet_info()*)
- **packet_payloads** (*list of range()s*) – Word-indices which are data payloads, and should be mared as valid (see *get_packet_info()*)
- **channel_indices** (*list of ints*) – Header entries for the channel field of each packet to be sent
- **ant_indices** (*list of ints*) – Header entries for the antenna field of each packet to be sent
- **dest_ips** – list of str IP addresses for each packet to be sent.
- **dest_ports** (*list of int*) – UDP destination ports for each packet to be sent.
- **nchans_per_packet** (*int*) – Number of frequency channels per packet sent.
- **nchans_per_xeng** (*int*) – Total number of frequency channels sent to each destination IP.
- **n_signals_per_packet** (*int*) – Number of signals in each packet sent.
- **n_signals_per_xeng** (*int*) – Number of signals expected by each destination IP.
- **print** (*bool*) – If True, print config for debugging

All parameters should have identical lengths.

4.2.16 Ethernet Output Control

class lwa_f.blocks.eth.**Eth**(*host*, *name*, *logger=None*)

Instantiate a control interface for a 40 GbE block.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.

add_arp_entry(*ip*, *mac*)

Set a single arp entry.

Parameters

- **ip** (*str*) – The IP address matching the given MAC in dotted-quad string notation. Eg. '10.10.10.1'.
- **mac** (*int*) – The MAC address to be loaded to the ARP cache.

configure_source(*mac*, *ip*, *port*)

Configure the Ethernet interface source address parameters.

Parameters

- **ip** (*str*) – IP address of the interface, in dotted-quad string notation. Eg. '10.10.10.1'
- **mac** (*int*) – MAC address of the interface.
- **port** (*int*) – UDP source port for packets sent from the interface.

disable_tx()
Disable Ethernet transmission.

enable_tx()
Enable Ethernet transmission.

get_packet_rate()
Get the approximate packet rate, in packets-per-second.

Return pps Approximate number of packets sent in the last second.

Rtype pps int

get_status()
Get status and error flag dictionaries. See also: `status_reset`.

Status keys:

- tx_of : Count of TX buffer overflow events.
- tx_full : Count of TX buffer full events.
- tx_vld : Count of 64-bit words marked as valid for transmission.
- tx_ctr: Count of transmission End-of-Frames marked valid.

Returns (status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary are as defined in *error_levels.py* and indicate that values in the status dictionary are outside normal ranges.

initialize(*read_only=False*)
Initialize the block. See also: `configure_source`, which sets transmission source attributes.

Parameters **read_only** (*bool*) – If False, reset error counters. If True, do nothing.

reset()
Disable, then reset the 40 GbE core. It must be enabled with `enable_tx` before traffic will be transmitted.

status_reset()
Reset all status counters.

4.3 `etcd` Interface

The `etcd` F-Engine interface provides a mechanism to control multiple SNAP2 boards running F-Engine firmware via the passing of messages through an `etcd` key-value store.

In order to use the `etcd` control interface, a daemon `Snap2FengineEtcdClient` instance is required to be running on a server with network access to the SNAP2 hardware being controlled.

4.3.1 Key Organization

For an F-Engine running on a SNAP2 `hostname`, there are three relevant `etcd` key paths.

Command

The command key is:

`/cmd/snap/<id>`

`id` is the one-indexed ID number of the SNAP board which should respond to these commands. Commands issues to `id=0` are processed by all SNAP boards.

Writing messages to this key results in the execution of `Snap2Fengine` command methods.

Response

Each command written to the command key will elicit a response published to the key:

`/resp/snap/<id>`

The response message will indicate the success of failure of the command, and may contain a command-dependent data payload. For example, a spectra, or snapshot of ADC samples.

Monitor

In addition to the command/response interface described above, a set of basic status values are continuously-polled and written to etcd. This allows basic monitoring scripts to simply watch the etcd keystore to determine system health, rather than having to issue commands.

Each SNAP board reports monitor information to the key:

`/mon/snap/<id>`

The monitor key is indended to be continuously updated on a ~1 second time cadence, and contains low data-volume status information. For example, FPGA temperature, ADC RMS, number of transmitted Ethernet packets, and similar.

4.3.2 Command/Response Protocol

The Command/Response protocol is designed to be a simple interface to the underlying `Snap2Fengine` control class. It will naturally extend as the control class functionality is expanded.

A simple example of a control client is implemented in the `Snap2FengineEtcdControl` class, which is used for internal testing.

Command Format

Commands sent to the command key are JSON-encoded dictionaries, and should have the following fields:

Field	Type	Description
id	string	A unique string associated with this command, used to identify the command's response
cmd	string	Command name
val	dictionary	**A dictionary containing keys:** • time (string) The time these statistics were gathered, in ISO 3339 format. • block (string): The firmware block name to which this command applies. • kwargs (dictionary): Dictionary of arguments required by the `block.command` method.

Allowed values for ``**block**`` are any of the keys in the `Snap2Fengine` blocks attribute. I.e.:

```
from lwa_f import snap2_fengine
f = snap2_fengine.Snap2Fengine('snap2-rev2-11')
for block in sorted(f.blocks.keys()): print(block)
adc
autocorr
corr
delay
eq
eqtvg
eth
fpga
input
noise
packetizer
pfb
powermon
reorder
sync
```

Additionally, the block name `feng` may be used to refer to the `Snap2Fengine` instance, and the block name `controller` may be used to refer to special commands associated with the `etcd` service process.

Allowed values for ``**cmd**`` are any of the methods which can be called against `Snap2Fengine.blocks[block]`. For example, for the `delay` block, allowed commands are:

- `get_max_delay`
- `set_delay`
- `get_delay`
- `initialize`
- `get_status`

All blocks are instances of the generic `Block` class, and thus it is also possible to call parent class methods such as `read_uint` and `write_uint`. These directly manipulate FPGA registers, and should be used with caution.

The ``**kwargs**`` field should contain any arguments required by the command method being called. For example, the Fengine `delay` block's `set_delay` method requires a `stream` argument (to select which of the 64 SNAP2 data streams is being manipulated) and a `delay` argument (to set the delay for this stream).

As such, in order to set the delay of data stream `5` to `100` adc samples, a command should be issued with:

Field	Value
cmd	"set_delay"
val	{block: "delay", kwargs: {"stream": 5, "delay": 100}

An example of a valid command JSON string, issued with the above parameters at UNIX time 1618060712.60 and with id="1" is:

```
'{"cmd": "set_delay", "val": {"block": "delay", "timestamp": 1618060712.6,
"kwargs": {"stream": 5, "delay": 100}}, "id": "1"}'
```

Consult the Snap2Fengine API details for a list of commands and their arguments.

Response Format

Every command sent elicits the writing of JSON-encoded dictionary to the response key. This dictionary has the following fields:

Field	Type	Description
id	string	A string matching the id field of the command string to which this is a response
val	dictionary	**A dictionary containing keys:** • timestamp (float) The UNIX time when this response was issued. • status (string): The string "normal" if the corresponding command was processed without error, or "error" if it was not. • response (dictionary): The response of the command method, as determined by the command API. If a method would usually return a numpy array, when using the etcd interface the response will be a list. In the event that the status field is "error", The response field will contain an error message string.

Not all Snap2Fengine methods return values, in whice case the response field is null. The previous command example (setting a delay) results in the underlying API call Snap2Fengine.blocks['delay'].set_delay(5, 100) which returns None. The response to the example command (assuming processing the command took 0.2 milliseconds) is thus:

Field	Value
id	"1"
val	{timestamp: 1618060712.8, status: "normal", response: null}

or, in JSON-encoded form:

```
'{"id": "1", "val": {"timestamp": 1618060712.8, "status": "normal",
"response": null}}'
```

If the response status field is "error", common response error messages, and their meanings are:

"JSON decode error"	Command string could not be JSON-decoded.
"Sequence ID not string"	Sequence ID was not provided in the command string or decoded to a non-string value.
"Bad command format"	Received command did not comply with formatting specifications. E.g. was missing a required field such as `block` or `cmd`.
"Command invalid"	Received command doesn't exist in the `Snap2Fengine` API, or is prohibited for `etcd` access.
"Wrong block"	`block` field of the command decoded to a block which doesn't exist.
"Command arguments invalid"	`kwargs` key contained missing, or unexpected keys.
"Command failed"	The underlying `Snap2Fengine` API call raised an exception.

In the event that a command fails, more information is available in the `Snap2FengineEtcdClient` daemon logs.

Monitoring Interface

The `etcd` F-engine control service can be commanded to periodically push status data to monitoring keys using the following commands.

class lwa_f.snap2_feng_etcd_client.**Snap2FengineEtcdClient**(*fhost*, *fid*, *etcdhost='etcdv3service.sas.pvt'*, *etcdport=2379*, *logger=None*)

An ETCD client to interface a single SNAP2 F-Engine to an etcd store.

Parameters

- **fhost** – Hostname (or IP, in dotted quad notation) of target F-Engine
- **fid** (*int*) – SNAP ID, used to associate a service with an etcd key "/cmd/snap/<ID>".
- **etcdhost** (*string*) – Hostname (or IP, in dotted quad notation) of target etcd server.
- **etcdport** (*int*) – Port on which etcd is served
- **logger** (*logging.Logger*) – Python *logging.Logger* instance to which this class's log messages should be emitted. If None, log to stderr

is_polling()

Returns True if the baground status polling thread is currently running. False otherwise.

Return type bool

poll_stats()

Poll all statistics and push to etcd.

set_log_level(*level*)

Set the logging level.

Parameters **level** (*string*) – Logging level. Should be "debug", "info", or "warning"

start_command_watch()

Listen for commands on this F-Engine's command key, as well as the "all SNAPs" key. Stop listening with *stop_command_watch()*

Internally, this method sets the *_etcd_watch_ids* attribute to allow a watch to later be cancelled.

start_poll_stats_loop(*pollsecs=10*, *expiresecs=- 1*)
Start a loop, calling `poll_stats` every `pollsecs` seconds for a duration of `expiresecs` seconds. Internally, this method launches a background thread to poll for stats, and sets the `_is_polling` Event attribute. Stop polling with `stop_poll_stats_loop()`. Check for current polling state with `is_polling()`.

Parameters

- **pollsecs** (*float*) – Number of seconds between poll loops.
- **expiresecs** (*int*) – Number of seconds for while poll loop should run. If <0, run forever.

stop_command_watch()
Stop listening for commands on this F-Engine's command key, as well as the "all SNAPs" key.

Internally, this method sets the *_etcd_watch_ids* attribute to *[]*.

stop_poll_stats_loop(*wait=True*)
Stop the background polling loop.

Parameters **wait** (*Bool*) – If True, block until polling loop has closed gracefully. If False, set the stop trigger and return immediately.

These should be invoked similarly to commands against firmware blocks, but use the `block` name `controller`.

For example, to turn the monitoring poll loop on for 1 hour, with a polling interval of 30 seconds, issue the command:

Field	Value
`cmd`	`"start_poll_stats_loop"`
`val`	`{block: "controller", kwargs: {"pollsecs": 30, "expiresecs": 3600}`

The monitor key contains a nested dictionary with the following keys:

Field	Type	Description
timestamp	float	The UNIX timestamp at which the monitor key was last updated.
stats	dictionary	A nested dictionary containing status values, keyed first by block name (eg. `delay`, `eq` or `corr`) and then by status key.
flags	dictionary	A nested dictionary containing status values, keyed first by block name (eg. `delay`, `eq` or `corr`) and then by status key. This dictionary contains a subset of the keys present in the `stats` field. Where they exist, a value associated with a key indicates whether the corresponding value in the `stats` dictionary is out of normal range. Values in the flags dictionary are interpretted as in `error_levels.py`; i.e.: • 0: Value is OK • 1: Value is different from operational normal • 2: Value is outside expected range • 3: Value represents an error condition

Entries in the `stats` dictionary are shown below

Dictionary Key	Dictionary sub-key	type	Description
autocorr	acc_len	int	Accumulation length, in spectra, of the internal autocorrelation module.
corr	acc_len	int	Accumulation length, in spectra, of the internal correlation module.
delay	delay``n``	int	Delay of each of the `n` data streams.
delay	maxdelay	int	The maximum delay supported by the firmware.
eq	binary_point	int	The position of the EQ coefficient binary point
eq	width	int	The bit width of the EQ coefficients
eq	clip_count	int	The number of clips when requantizing spectra to 4-bit. This counter resets every [TODO: when??]
eq	coefficients``n``	list (int)	The currently loaded coefficients, in integer form (i.e., interpretted with all bits above the binary point)
eq_tvg	tvg_enabled	bool	True if the post- FFT test vector generator is enabled. False otherwise.
eth	tx_ctr	int	Running count of number of packets transmitted.
eth	tx_err	int	Running count of number of packet errors detected.
eth	tx_full	int	Running count of number of transmission buffer overflow events.
eth	tx_vld	int	Running count of number of 256-bit words transmitted.
feng	flash_firmware	string	The current .fpg bitstream file stored in flash memory
feng	flash_firmware_md5	int	The MD5 checksum of the .fpg bitstream stored in flash
feng	host	string	The hostname of the SNAP2 board
feng	programmed	bool	True if the board appears to be running DSP firmware. False otherwise.
feng	serial	string	A notional "serial number" of this hardware. Not yet implemented.
feng	sw_version	string	The software version of the `lwa_f` python library currently in use
feng	sys_mon	string	"reporting" if the current firmware has a working system monitor module. "not reporting" if no monitoring is available.
feng	temp	float	FPGA junction temperature, in degrees C, reported by system monitor (if available)
feng	vccaux	float	Voltage of the VCCAUX FPGA power rail reported by system monitor (if available)
feng	vccbram	float	Voltage of the VCCBRAM FPGA power rail reported by system monitor (if available)
feng	vccint	float	Voltage of the VCCINT FPGA power rail reported by system monitor (if available)

continues on next page

Table 4.1 – continued from previous page

Dictionary Key	Dictionary sub-key	type	Description
input	mean``n``	float	Mean of ADC sample values for input ADC `n`.
input	rms``n``	float	RMS of ADC sample values for input ADC `n`.
input	power``n``	float	Mean of squares of ADC sample values for input ADC `n`.
input	switch_position``n``	string	Switch position for input data stream `n`. `noise` for internal noise generators, `adc` for analog inputs, `zero` for zero.
noise	noise_core00_seed	int	Seed value for internal noise generator 0.
noise	noise_core01_seed	int	Seed value for internal noise generator 1.
noise	noise_core02_seed	int	Seed value for internal noise generator 2.
noise	output_assignment``n``	int	Noise source (0 - 5) currently assigned to data stream `n`.
pfb	fft_shift	int	Currently loaded FFT shift schedule.
pfb	overflow_count	int	Running count of FFT overflow events.
sync	ext_count	int	Running count of number of external sync pulses received since FPGA was programmed.
sync	int_count	int	Running count of number of internal sync pulses received since FPGA was programmed.
sync	period_fpga_clks	int	Detected period of external sync pulses in units of FPGA clock cycles.
sync	uptime_fpga_clks	int	Number of FPGA clock ticks since the FPGA was last programmed

CHAPTER

FIVE

F-ENGINE REGISTER DEFINITIONS

F-Engine firmware is designed to be manipulated using the provided `Snap2FEngine` class, or its higher level etcd-based control interface.

However, individual F-engine registers and their purposes are described below.

5.1 Register Descriptions

Register Name	Size (Bytes)	Permission	Description
ADC Registers			
ads5296_controller_<i>_<j>	128	r/w	ADS5296 board control register for the `j` th ADC card in FMC `i`. Should be manipulated using the `ADS5296` class of the `casperfpga` python library.
ads5296_spi_controller<i>	16	r/w	ADS5296 SPI controller register for ADC cards on FMC slot `i`. Should be manipulated using the `ADS5296` class of the `casperfpga` python library.
ads5296_wb_ram<i>_<j>_<k>	4096	r/w	ADC snapshot RAM for ADC chip `k` on board `j` of FMC port `i`. Should be manipulated using the `ADS5296` class of the `casperfpga` python library.
adc_rst	4	r/w	Set to `1` to assert the ADC reset. This flushes the ADC output FIFOs. Data will not begin flowing again until the ADC `sync` input transitions from `0` to `1`.
adc_snapshot_trigger	4	r/w	When this register transitions from `0` to `1`, all ADC channels will simultaneously capture a burst of ADC samples.
adc_sync	4	r/w	Write `1` to this register to assert the ADC's `sync` input. This input is edge sensitive.
Auto-Correlator Registers			
autocorr_acc_cnt	4	r	32-bit wrapping accumulation counter, which increments each time a new accumulation is recorded to RAM. There is no buffer locking when correlation data are read, so this counter can be used to ensure that data have not changed during a read.
autocorr_acc_len	4	r/w	Accumulation length, in spectra.

continues on next page

Table 5.1 – continued from previous page

Register Name	Size (Bytes)	Permission	Description
autocorr_common_dout0_bram	262144	r/w	For memory name suffix `<n>_bram`, stores the autocorrelation powers for frequency channels `8n` of 16 consective F-engine inputs. Data are stored in `[input_number, freq_channel]` order.
autocorr_common_dout1_bram	262144	r/w	For memory name suffix `<n>_bram`, stores the autocorrelation powers for frequency channels `8n` of 16 consective F-engine inputs. Data are stored in `[input_number, freq_channel]` order.
autocorr_common_dout2_bram	262144	r/w	For memory name suffix `<n>_bram`, stores the autocorrelation powers for frequency channels `8n` of 16 consective F-engine inputs. Data are stored in `[input_number, freq_channel]` order.
autocorr_common_dout3_bram	262144	r/w	For memory name suffix `<n>_bram`, stores the autocorrelation powers for frequency channels `8n` of 16 consective F-engine inputs. Data are stored in `[input_number, freq_channel]` order.
autocorr_common_dout4_bram	262144	r/w	For memory name suffix `<n>_bram`, stores the autocorrelation powers for frequency channels `8n` of 16 consective F-engine inputs. Data are stored in `[input_number, freq_channel]` order.
autocorr_common_dout5_bram	262144	r/w	For memory name suffix `<n>_bram`, stores the autocorrelation powers for frequency channels `8n` of 16 consective F-engine inputs. Data are stored in `[input_number, freq_channel]` order.
autocorr_common_dout6_bram	262144	r/w	For memory name suffix `<n>_bram`, stores the autocorrelation powers for frequency channels `8n` of 16 consective F-engine inputs. Data are stored in `[input_number, freq_channel]` order.
autocorr_common_dout7_bram	262144	r/w	For memory name suffix `<n>_bram`, stores the autocorrelation powers for frequency channels `8n` of 16 consective F-engine inputs. Data are stored in `[input_number, freq_channel]` order.
autocorr_mux_sel	4	r/w	Input selection controller. Set bits `[1:0]` to `n` to compute the autocorrelations of F-engine inputs `16n..16(n+1)-1`.
Channel Reorder Registers			
chan_reorder_dynamic_map1	16384	r/w	Determines the remapping of input samples to output samples. I.e., if the first entry is `5`, then the first sample into the reorder will come out 5th
Correlation Registers			
corr_0_acc_cnt	4	r	32-bit wrapping accumulation counter, which increments each time a new accumulation is recorded to RAM. There is no buffer locking when correlation data are read, so this counter can be used to ensure that data have not changed during a read.

continues on next page

Table 5.1 – continued from previous page

Register Name	Size (Bytes)	Permission	Description
corr_0_acc_len	4	r/w	Accumulation length, in FPGA clock cycles. To specify an accumulation length as a number of spectra this register should be written with `number_of_spectra x frequency_channels_per_spectra`. Here the number of frequency channels per spectra is the number *output* by the correlation subsystem. Glitches may occur (resulting in the no new accumulations for a few seconds) if the value in this register is reduced without re-applying a sync to the correlation module.
corr_0_dout	32768	r/w	Correlation data buffer. Word `[2n]` is the real part of the correlation of frequency channel `n` (after any channel averaging). Word `[2n+1]` is the imaginary part of this correlation.
corr_0_input_sel	4	r/w	Input selector control register. Bits `[5:0]` select the unconjugated correlation input. Bits `[13:8]` select the conjugated correlation input.
Delay Registers			
delay_<n>_delay	4	r/w	The delay applied to stream `n`. Any value entered in this 32-bit register will be interpretted modulo the maximum supported delay.
delay_max_delay	4	r	Interpretted as a unint32 value, contains the maximum allowed delay, in ADC samples, which may be applied to a data stream.
Equalization Registers			
eq_core<n>_clip_cnt	4	r	A count of the number of times a sample has been clipped, for any of F-engine inputs `16n..16(n+1)-1`. This counter is reset only by a global system reset.
eq_core<n>_coeffs	131072	r/w	This memory holds coefficients for F-engine streams `16n..16(n+1)-1`. Coefficients are stored as a `[16, n_fft_channels / 8]` array, where the first axis (over input number) varies *slowest*. Coefficients are shared over 8 consecutive frequency channels, with coefficient `m` being applied to frequency channel `m..8(m+1)-1`. Coefficients are interpretted with 5 bits below the binary point.
40GbE Registers			
eth_ctrl	4	r/w	40GbE control register. Bit `0` is an active high reset of the 40GbE interface core. Bit `1` is an active high transmission enable for the core, which takes effect either after the next packet is sent, or when the `force` flag is asserted. Bit `18` is an active high reset for statistics provided by the Ethernet core. Bit `19` is an active high `force` signal, which causes the current transmission enable flag to immediately take effect.
eth_forty_gbe_txctr	4	r	Counter which increments each time a valid end-of-frame is seen on a packet to be transmitted

continues on next page

Table 5.1 – continued from previous page

Register Name	Size (Bytes)	Permission	Description
eth_forty_gbe_txfullctr	4	r	Counter which increments every time the transmission buffer is full
eth_forty_gbe_txofctr	4	r	Counter which increments on every transmission overflow event
eth_forty_gbe_txvldctr	4	r	Counter which increments with every valid word of data input to the core
Input Control Registers			
input_bit_stats_histogram_output	32768	r/w	BRAM storing histogram data. Word `n` in the bottom half of the RAM stores the number of occurances of of ADC sample code `n` in _even_ numbered ADC samples. Word `n` in the _top_ half of the RAM stores the number of occurances of ADC sample code `n` in _odd_ numbered ADC samples.
input_bit_stats_input_sel	4	r/w	Input selection. The least significant 6 bits should be set to `n` to compute the histogram for input `n`.
input_rms_enable	4	r/w	Enable statistics recording. If the least-significant bit is `1`, statistics will be recorded to RAM.
input_rms_levels	32768	r/w	Multi-channel statistics. For the `n` th 64-bit word, the least signficant 32 bits represent the accumulated ADC power, accumulated over 65536 samples and stored as an unsigned 32-bit integer. The most significant 32 bits represent the sum of 65536 ADC samples stored as a signed 32-bit integer.
input_source_sel0	4	r/w	Selector control to determine if signal streams carry ADC samples, digital noise samples, or zeros. Streams `n=0..15` is controlled by register bits `[2n+1:2n]`. Value `0` selects noise; `1` selects ADC; `2` selects zeros.
input_source_sel1	4	r/w	Selector control to determine if signal streams carry ADC samples, digital noise samples, or zeros. Streams `n=16..31` is controlled by register bits `[2(n-16)+1:2(n-16)]`. Value `0` selects noise; `1` selects ADC; `2` selects zeros.
input_source_sel2	4	r/w	Selector control to determine if signal streams carry ADC samples, digital noise samples, or zeros. Streams `n=32..47` is controlled by register bits `[2(n-32)+1:2(n-32)]`. Value `0` selects noise; `1` selects ADC; `2` selects zeros.
input_source_sel3	4	r/w	Selector control to determine if signal streams carry ADC samples, digital noise samples, or zeros. Streams `n=48..63` is controlled by register bits `[2(n-48)+1:2(n-48)]`. Value `0` selects noise; `1` selects ADC; `2` selects zeros.
Noise Generator Registers			
noise_octal_mux0_sel	4	r/w	The lower 2 bits define the select signal for the noise multiplexor. If this register has value `v`, then noise generator `v` is selected.

continues on next page

Table 5.1 – continued from previous page

Register Name	Size (Bytes)	Permission	Description
noise_octal_mux1_sel	4	r/w	The lower 2 bits define the select signal for the noise multiplexor. If this register has value `v`, then noise generator `v` is selected.
noise_octal_mux2_sel	4	r/w	The lower 2 bits define the select signal for the noise multiplexor. If this register has value `v`, then noise generator `v` is selected.
noise_octal_mux3_sel	4	r/w	The lower 2 bits define the select signal for the noise multiplexor. If this register has value `v`, then noise generator `v` is selected.
noise_octal_mux4_sel	4	r/w	The lower 2 bits define the select signal for the noise multiplexor. If this register has value `v`, then noise generator `v` is selected.
noise_octal_mux5_sel	4	r/w	The lower 2 bits define the select signal for the noise multiplexor. If this register has value `v`, then noise generator `v` is selected.
noise_octal_mux6_sel	4	r/w	The lower 2 bits define the select signal for the noise multiplexor. If this register has value `v`, then noise generator `v` is selected.
noise_octal_mux7_sel	4	r/w	The lower 2 bits define the select signal for the noise multiplexor. If this register has value `v`, then noise generator `v` is selected.
noise_seeds0	4	r/w	Noise generator seed values. If the 32-bit value in this register is `v`, the seed for the first noise generator is (using Verilog syntax) `{8'h5c, v[7:0], v[0:7], 8'ha3}`. The seed for the second noise generator is `{8'h5c, v[15:8], v[8:15], 8'ha3}`. The seed for the third noise generator is `{8'h5c, v[23:16], v[16:23], 8'ha3}`. The seed for the fourth noise generator is `{8'h5c, v[31:24], v[24:31], 8'ha3}`
Packetizer Registers			
packetizer_ants	262144	r/w	Antenna header entry map. Word `n``contains the header antenna ID field for sample ``n` in a transmission period. This field is only relevant for samples accompanied by `valid` and `header` flags.
packetizer_chans	262144	r/w	Channel ID header entry map. Bits `[23:0]` of word `n` contain the header channel ID field for sample `n` in a transmission period. This header field should hold the index of the first channel in a packet. Bits `[31:24]` contain the header channel block index field for this sample. These field is only relevant for samples accompanied by `valid` and `header` flags.
packetizer_flags	262144	r/w	Packet flags. For word `n`, bit `0` is an active high flag which indicates that sample `n` in a transmission period is a packet header. Bit `8` is an active high flag which indicates that this word is valid and should be transmitted. Bit `16` is an active high flag indicating that this word is the last in a packet.

continues on next page

Table 5.1 – continued from previous page

Register Name	Size (Bytes)	Permission	Description
packetizer_ips	262144	r/w	IP Destination address map. Word `n` contains the IP address to which sample `n` in a transmission period should be sent. Only entries accompanied by `valid` and `end of frame` flags result in a packet's destination being set from this register.
packetizer_n_chans	4	r/w	32-bit `n_chans` header field for F-engine output data packets, indicating the number of frequency channels per packet.
packetizer_n_pols	4	r/w	32-bit `n_pols` header field for F-engine output data packets, indicating the number of antenna-polarizations present in a packet.
packetizer_ports	262144	r/w	UDP Destination port map. For word `n`, bits `[15:0]` contain the UDP port to which sample `n` in a transmission period should be sent. Only samples accompanied by `valid` and `end of frame` flags result in a packet's destination being set from this register.
PFB Registers			
pfb_ctrl	4	r/w	PFB control register. Bit `18` is an active-high reset for overflow statistics counters. Bits `15:0` hold the FFT shift schedule for the PFB processing pipeline, with bit `n` an active high shift signal for the `n` th FFT stage.
pfb_pfb16x_<n>_status	4	r	A 32-bit counter which, for register name suffix `<n>_status` increments every time an FFT overflow event is detected in any of FFT channels `16n..16(n+1)-1`. Resets only when commanded via the PFB control register, and will wrap once the maximum value is reached.
Post-EQ Test Vector Generator Registers			
post_eq_tvg_core<n>_tv	524288	r/w	This memory holds 4+4 bit complex test vector streams for F-engine streams `16n..16(n+1)-1`. Test vectors are stored as a `[16, n_fft_channels]` array, where the first axis (over input number) varies *slowest*, and vectors are stored in order of increasing FFT channel. An 8-bit word in location `[m, c]` is interpretted as a 4+4 bit complex value for channel `c` of F-engine stream `16n + m`. The most significant 4 bits are interpretted as the real part of the complex value.
post_eq_tvg_tvg_en	4	r/w	Test vector multiplexor control. If the least significant bit is `1`, ADC data will be replaced with software-controllable test vectors.
Synchronization Registers			

continues on next page

Table 5.1 – continued from previous page

Register Name	Size (Bytes)	Permission	Description
sync_ctrl	4	r/w	Timing control register. Bit `0`: active high enable allowing telescope time counter to be loaded from telescope time load registers on next synchronization pulse. Bit `1`: When transitioned from `0` to `1`, forces telescope time counter to be loaded from telescope time load registers immediately. Bit `2`: active high telescope time counter reset. Bit `3`: active high reset for error counters. Bit `4`: active high `arm` signal, which causes a global system reset, released on the next synchronization pulse; a system sync on the next synchronization pulse. Bit `5`: When transittioned from `0` to `1`, emulates the arrival of an external sync pulse. Bit `6`: When transittioned from `0` to `1`, arms noise generator seed loaders.
sync_ext_sync_count	4	r	Number of external synchronization pulses received.
sync_ext_sync_period	4	r	Measured number of FPGA clock ticks between previous two external synchronization pulses.
sync_ext_sync_tt_lsb	4	r	Least significant 32 bits of the telescope time counter, at the point of the last external synchronization pulse.
sync_ext_sync_tt_msb	4	r	Most significant 32 bits of the telescope time counter, at the point of the last external synchronization pulse.
sync_int_sync_count	4	r	Number of synchronization pulses emitted via GPIO.
sync_latency	4	r	Measured latecy, in FPGA clock ticks, between transmitting a synchronization pulse and receiving it from the timing distribution system.
sync_sync_div_bits	4	r	Holds the log2 value of the sync period – i.e. the rate at which sync pulses are emitted from the board via GPIO – in FPGA clock ticks.
sync_tt_load_lsb	4	r/w	Telescope time load register. Least significant 32 bits to load to the telescope time counter.
sync_tt_load_msb	4	r/w	Telescope time load register. Most significant 32 bits to load to the telescope time counter.
sync_tt_lsb	4	r	Least significant 32 bits of the telescope time counter.
sync_tt_msb	4	r	Most significant 32 bits of the telescope time counter.
sync_uptime_msb	4	r	Board uptime, in units of `2^32` FPGA clock ticks.
Version Registers			
version_timestamp	4	r	Unix timestamp, in integer seconds, when firmware was last compiled.
version_version	4	r	Version register. Bits `[31:24]`: major version. Bits `[23:16]`: minor version. Bits `[15:8]`: revision. Bits `[7:0]`: buxfix.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

H

I

L

M

N

P

R

S

T

U

W