COSMIC-VLA FPGA

Release vla-dev-9b1f3d8:cosmic_f-0.1.0-9b1f3d81

Jack Hickish

May 22, 2023

CONTENTS:

CHAPTER

ONE

OBTAINING THE SOURCE CODE

The COSMIC F-Engine pipeline is available at https://github.com/realtimeradio/vla-dev. Follow the following instructions to download and install the pipeline.

1.1 Get the Source Code

Clone the repository and its dependencies with:

```
# Clone the main repository
git clone https://github.com/realtimeradio/vla-dev
# Clone relevant submodules
cd vla-dev
git submodule update --init --recursive
```

1.2 Install Prerequisites

1.2.1 Firmware Requirements

The COSMIC 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 VLA F-Engine firmware is designed to run on an AlphaData ADM-PCIe-9H7[1] FPGA board, and provides channelization of digital data streams from two VLA antennas. Two different versions of the firmware exist - one for each of the two VLA digitization modes.

In the VLA's *3-bit* mode, F-engine firmware processes (for each of two dual-polarization antennas), 4x2GHz bands.

In the VLA's *8-bit* mode, F-engine firmware processes (for each of two dual-polarization antennas), 2x1GHz bands.

In both cases, the firmware channelizes the received bands into 1 MHz "coarse channels", and transmits these over 100Gb Ethernet at 8-bit (8-bit real + 8-bit imaginary) sample resolution.

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

[1] See https://www.alpha-data.com/product/adm-pcie-9h7

Parameter	Value	Notes
Number of antennas processed per firmware instance	2	
Number of inputs per antenna	4 (VLA 8-bit mode); 8 (VLA 3-bit mode)	Dual-polarization x 2 IFs (8-bit mode); 4 IFs (3-bit mode)
Sampling rate	2.048GSa /s (8-bit mode); 4.096 GSa/s (3-bit mode)	VLA digitization standards
Test inputs	Noise; zeros; additive sine- waves	Firmware contains 2 independent gaussian noise generators. Any of the input data streams may be replaced with any of these digital noise sources, or zeros. User-programmable sine-waves may be added to this data stream
Delay compensation	<256k samples (8-bit mode); <512k samples (3-bit mode)	Programmable per- input. Maximum 256 useconds in all modes (approx 38km free-space light travel)
Polyphase Filter Bank Channels	1024 (8-bit mode); 2048 (3-bit mode)	1 MHz coarse channel resolution in all modes. FFTs are implemented as complex transforms with twice as many points as channels output.
Polyphase Filter Bank Window	Hamming; 4-tap	
Polyphase Filter Bank Input Bitwidth	8 bit (3-bit mode); 16bit (8-bit mode)	Sample growth occurs during LO offset removal
FFT Coefficient Width	18 bits	
FFT Data Path Width	18 bits	
Post-FFT Scaling Coefficient Width	20	
Post-FFT Scaling Coefficient Binary Point	8	
Number of Post-FFT Scaling Coefficients (per input)	256	Coefficients shared over 4 (8-bit mode) or 8 (3-bit mode) coarse channels
Post-Quantization Data Bitwidth	8	8-bit real; 8-bit imaginary

The Simulink block diagram of the dual-antenna F-engine firmware which runs on an ADM-PCIe-9H7 FPGA board is shown in Fig. 2.1. The block diagram showing the DSP elements of a single antenna pipeline is shown in Fig. 2.2.

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.

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

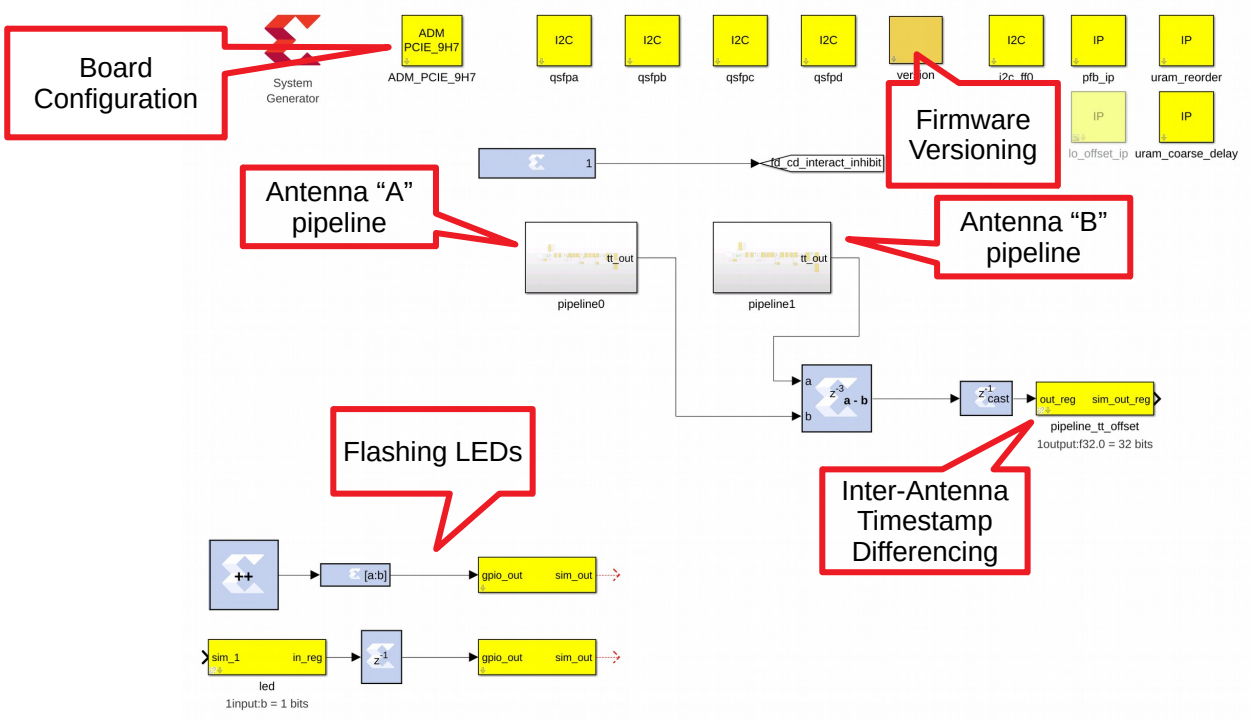

Fig. 2.1: F-Engine top-level Simulink diagram.

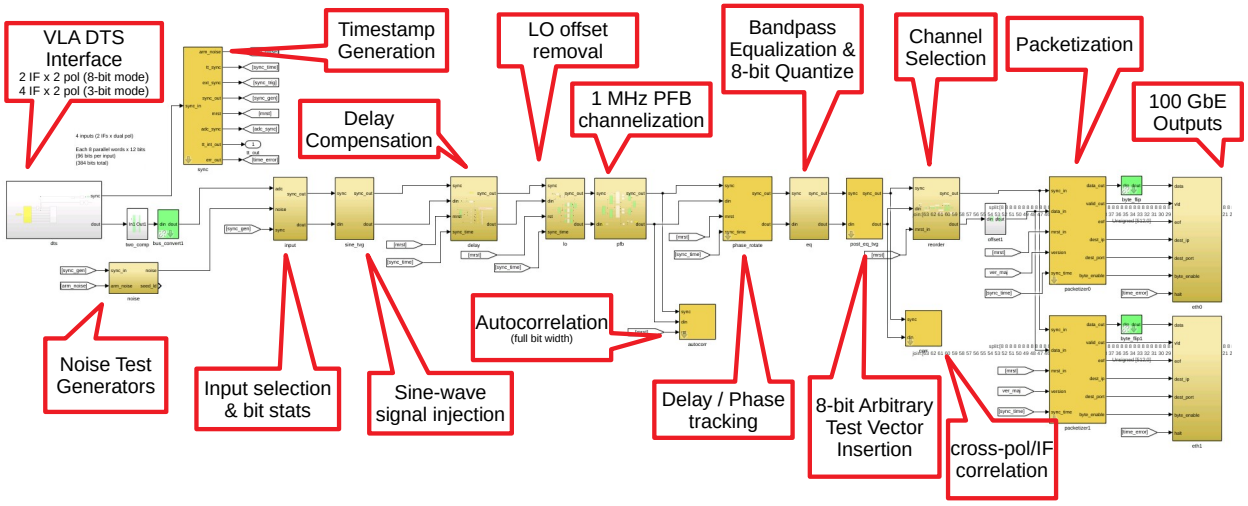

Fig. 2.2: Simulink diagram for a single antenna F-engine pipeline.

```
# Import the COSMIC F-Engine control library
from cosmic_f import CosmicFengine

# Instantiate a CosmicFengine instance, to communicate with
# the F-engine pipeline for the first antenna of a board with
# PCIe enumeration 0x3d, to be programmed with firmware 'firmware.fpg'
f = CosmicFengine('pcie3d', fpgfile=firmware.fpg, pipeline_id=0)

# Program a board
f.program() # Load firmware (if it not already running)

# Initialize all the firmware blocks
f.initialize(read_only=False)

# Issue a reset and synchronization pulse
f.sync.arm_sync()
f.sync.sw_sync()

# Do, something, like plot autocorrelations for all polarizations/IFs
f.autocorr.plot_all_spectra()
```

2.1.2 Block Descriptions

Each block in the firmware design can be controlled using an API described in Section 4.

DTS (*dts*)

The DTS block provides an interface to the VLA DTS (Data Transmission System). It receives streaming DTS data from a VLA antenna on 12 parallel 10.24 Gb/s links, and decodes these streams into a time series of ADC samples.

Timestamp Generation (*sync*)

The timestamp generation block, manages the synchronization of multiple boards. Software control, alongside pulses embedded in the DTS data streams allow multiple boards to lock to a common time origin. This is vital for combining data streams from antennas which are processed by different FPGA boards.

Noise Generators (*noise*)

The Noise Generation block provides multiple white noise streams. These can be used to emulate antenna signals which are either perfectly correlated or perfectly uncorrelated. Designed for testing and verification, replacing DTS signals with artificial noise provides a useful ability to check the functionality of downstream delay / correlation systems.

Input Selection (*input*)

The input block provides bit statistics (mean, RMS, histograms) for F-engine inputs. It also contains a multiplexor which allows F-engine inputs to be replaces with either zeros, or a signal from an upstream *noise* block.

Sine-wave Injection (*sinegen*)

The Sine-wave injection block allows a software-defined sine wave to be added to a data stream. This allows testing of the downstream LO-offset removal scheme, as well as verification of frequency channel labelling.

Delay Compensation (*delay*)

The delay block allows runtime-programmable delays to be inserted into a data stream. These can be used to compensate for cable delays in the upstream system, as well as array geometry when phasing to an astronomical source.

LO-offset Removal (*lo*)

The LO block multiplies F-engine signals by a runtime-programmable LO, to compensate for the frequency offsets in the analog LOs used at the VLA antennas.

PFB (*pfb*)

The PFB implements a 1 MHz channelizer, slicing the wideband DTS inputs into multiple frequency channels.

Autocorrelation (*autocorr*)

The *autocorr* block calculates, prior to any requantization, the autocorrelation of each F-engine input, with a runtime-programmable accumulation length.

Delay / Phase Tracking (*phase_rotate*)

The phase rotation block performs delay and phase tracking, in order to fringe-stop at the phase center of an observation.

Bandpass Equalization (*eq*)

The *eq* block provides the ability to multiply spectra by a set of frequency dependent (but time-independent) real-valued coefficients. This allows the bandpass of each F-engine input to be flattened, and the overall power levels to be set appropriately for downstream requantization.

Test Vector Insertion (*eqtvg*)

The *eqtvg* block provides the ability to replace data streams with a runtime-programmable test pattern, which may vary with frequency channel and input number, but is invariant over time.

Cross-Correlation (*corr*)

The *corr* block provides the ability to correlate any pair of F-engine inputs (for a single antenna). Since F-engine inputs are different IFs and polariations of a common antenna, this block is mostly useful for debugging and verification, when used alongside the variou test vector insertion modes.

Channel Selection (*chanreorder*)

The *chanreorder* block reorders channels within a spectra. Alongside the downstream packetization block, it can be used to dynamically define which frequency channels are transmitted from the F-engine.

Packetizer (*packetizer*)

The *packetizer* module inserts application headers to the F-engine data streams, and configures the destinations to which F-engine packets are sent.

100 GbE Outputs (*eths[]*)

The *eths* blocks encapsulate (multiple) 100GbE interfaces, and provide control for enabling and disabling Ethernet outputs, and packet transmission statistics.

CHAPTER

THREE

OUTPUT DATA FORMAT

F-Engine output data comprises a continuous stream of voltage samples, encapsulated in UDP packets. The format used is similar to that used at the ATA for 8-bit *Voltage Mode* observations, but with a larger number of time samples per packet.

Each packet contains a data payload of up to 8192 bytes, made up of 32 time samples for up to 128 frequency channels of dual-polarization data:

```
// Number of time samples per packet
#define N_t 32
// Number of polarizations per packet
#define N_p 2

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

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

- `version`; TODO: check how this is populated
- `type`; *Packet type*: Bit [0] is 1 if the axes of data payload are in order [slowest to fastest] channel x time x polarization. This is currently the only supported mode. Bit [1] is 1 if the data payload comprises 8+8 bit complex integers. This is currently the only supported mode.
- `n_chans`; *Number of Channels*: Indicates the number of frequency channels present in the payload of this data packet.
- `chan`; *Channel number*: The index of the first channel present in this packet. For example, a channel number `c` implies the packet contains channels `c` to `c + n_chans - 1`.
- `feng_id`; *Antenna ID*: A runtime configurable ID which uniquely associates a packet with a particular SNAP board.
- `timestamp`; *Sample number*: The index of the first time sample present in this packet. For example, a sample number s implies the packet contains samples s to $s + 15$. Sample number counts in units of spectra since the UNIX epoch, and can be referred to GPS time through knowledge of the system sampling rate and FFT length parameters.

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

CHAPTER

FOUR

CONTROL INTERFACE

4.1 Overview

A Python class `CosmicFengine` 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 `CosmicFengine` class provides an easy way to probe board status for a ADM-PCIe-9H7 board on the local network.

4.2 `CosmicFengine` Python Interface

The `CosmicFengine` class can be instantiated and used to control a single ADM-PCIe-9H7 board running LWA's F-Engine firmware. An example is below:

```
# Import the ADM-PCIe-9H7 F-Engine library
from cosmic_f import CosmicFengine

# Instantiate a CosmicFengine instance, to communicate with
# the F-engine pipeline for the first antenna of a board with
# PCIe enumeration 0x3d, to be programmed with firmware 'firmware.fpg'
f = CosmicFengine('pcie3d', fpgfile=firmware.fpg, pipeline_id=0)

# Program a board
f.program() # Load firmware (if it not already running)

# Initialize all the firmware blocks
f.initialize(read_only=False)

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

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

(continues on next page)

(continued from previous page)

```
# Grab some correlation data from the two pols of the first IF
corr_data = f.corr.get_new_corr(0, 1)
# ...
```

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

4.2.1 Top-Level Control

The Top-level `CosmicFengine` 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 `CosmicFengine` instance can be used to initialize, or get status from, all underlying firmware modules.

class cosmic_f.cosmic_fengine.**CosmicFengine**(*host*, *fpgfile*, *pipeline_id=0*, *neths=1*, *logger=None*, *remote_uri=None*, *redis_host=None*, *redis_port=6379*)

A control class for COSMIC's F-Engine firmware.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host. If *host* is of the form *xdmaA*, then we are connecting to the FPGA card with xdma driver ID *xdmaA*. In this case, connection may be direct, or via a REST server. If a REST server is supplied at *remote_uri*, this *host* parameter will be interpreted on the server's side (see *RemotePcieTransport.__init__()* and *casperfpga_rest_server:getXdmaIdFromTarget()*). If *host* is of the form *pcieB*, then the connection is to the PCIe device with enumeration 0xB
- **fpgfile** (*str*) – .fpg file for firmware to program (or already loaded)
- **pipeline_id** (*int*) – Zero-indexed ID of the pipeline on this host.
- **neths** (*int*) – Number of 100GbE interfaces for this pipeline.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.
- **remote_uri** (*str*) – REST host address, eg. *http://192.168.32.100:5000*. This triggers the transport to be a RemotePcieTransport object.
- **redis_host** (*str*) – Redis server host address
- **redis_port** (*int*) – Redis server port address

check_delay_time(*tolerance=0.1*)

A handy function to be called often in the runing of the delay_tracking thread. Checks the time registered by the delay block against the NTP time of the server and asserts that the difference is within the tolerance specified in the argument.

Args:
tolerance: float

Returns:

boolean: True - time is within tolerance.
False- time is not within tolerance.

check_delay_tracking(*delay_to_load*, *delay_rate_to_load*, *phase_to_load*, *phase_rate_to_load*, *loadtime*, *fshifts*, *sslo*, *sideband*, *clock_rate_hz=2048000000*, *invert_band=False*)

From the delay and delay rate values for this fengine instance, calculate an expected delay slope value for a given time. For that time interval, assert that the firmware reported delay slope matches the expected. This

function is spawned asynchronously and it checks the firmware reported slope/phase against the argument delay, delay rate, phase and phase rate. Finally, it publishes an informative dictionary to FENG_delayStatus for logging purposes.

Parameters

- **delay_to_load** (*ndarray{float}*) – The n_streams ns delays that the delay loading thread believes was loaded.
- **delay_rate_to_load** – The n_streams ns/s delay rates that the delay loading thread believes was loaded.
- **phase_to_load** (*ndarray{float}*) – The n_streams phases (radians) that the delay loading thread believes was loaded.
- **phase_rate_to_load** (*ndarray{float}*) – The n_streams phase rates (radians/s) that the delay loading thread believes was loaded.
- **loadtime** (*float*) – The time in seconds at which the delays were loaded.
- **fshifts** (*ndarray{float}*) – The fshift values in use from the phase correction in Hz
- **sslo** (*list*) – The sslo values in use from the delay model in mhz
- **sideband** (*list*) – The sideband values in use from the delay model
- **clock_rate_hz** (*int*) – ADC clock rate in Hz. If None, the clock rate will be computed from the observed PPS interval, which could fail if the PPS is unstable or not present.
- **invert_band** (*bool*) – If True, invert the gradient of the phase-vs-frequency channel. I.e., apply a fractional delay which is the negative of the physical delay.

cold_start(*program=True*, *initialize=True*, *test_vectors=False*, *sync=True*, *sw_sync=False*, *enable_eth=True*, *chans_per_packet=32*, *ninput=2*, *macs={}*, *source_ips=['10.41.0.101']*, *source_port=10000*, *dests=[]*, *dts_lane_map=None*, *fpgfile=None*, *sync_offset_ns=0.0*, *lo_fshift_list=None*, *target_rms=0.125*)

Completely configure an 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.
- **initialize** (*bool*) – If True, put all firmware blocks in their default initial state. 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.
- **chans_per_packet** (*int*) – Number of frequency channels in each output F-engine packet
- **ninput** (*int*) – Number of inputs to be sent. Values of `n*32` may be used to spoof F-engine packets from multiple SNAP2 boards.
- **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_ips** (*list of str*) – The IP addresses from which this board should send packets. A list with one IP entry per interface.
- **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`.
  - 'feng_ids': The identifying numbers of the antenna-streams, as interpreted by the destination. 1 per tuning/input.
  - 'interface_id': The zero-indexed output interface ID from which these data should be transmitted.
- **dts_lane_map** – If not None, override the default DTS lane mapping as part of initialization. This parameter does nothing if *initialize* is not True.
- **fpgfile** – The .fpg file to be loaded, if *program* is True. Should be a path to a valid .fpg file. If None is given, and programming, self.fpgfile will be loaded.
- **sync_offset_ns** (*float*) – Nanoseconds offset to add to the time loaded into the internal telescope time counter.
- **lo_fshift_list** (*List*) – If not None, a list of lo_fshifts in Hz to apply in order of streams.
- **target_rms** (*{float, None}*) – If not None, equalize the F-Engine streams to the target_rms provided using *self.set_equalization*.

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

Completely configure an 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.
- **initialize** (*bool*) – If True, put all firmware blocks in their default initial state. Initialization is always performed if the FPGA has been reprogrammed.
- **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 F-Engine Ethernet output.
- **config_file** (*str*) – Path to a configuration YAML file.

`delay_tracking`()

Started in a thread, this function calls on *set_delays* to update the F-Engine coefficients for delay tracking. This function listens for delay, delay_rate and delay_rate_rate coefficient updates on the channels subscribed to in *start_delay_tracking* and initialises calibration delays from META_calibrationDelays hash. In between receiving coefficient updates, when delay_track is set this function will interpolate new delay, delay_rate, phase and phase_rate values to upload to the F-Engine. When delay_track is cleared, this function will continuously load the calibration delays to the F-Engine.

`disable_tx`()

Disable Ethernet outputs.

Returns

The enable-state of each of N Ethernet interfaces, as a list of N bools, prior to the issuing of the disable command.

`enable_tx`(*enable_state=None*)

Enable Ethernet cores, as long as the DTS monitor hasn't disabled them. Internally, set the `eth_tx` attribute, so that even if the DTS monitor _has_ disabled network egress, it is clear that the user requested it be enabled.

Parameters

`enable_state` (*`list`*) – list of bools. If `enable_state[n]`, enable Ethernet core `n`.

`eth_tx`

A list of bools describing the user enablement of Ethernet cores.

`get_delay_tracking_mode`()

Fetches internal delay tracking mode.

Returns:

delay_mode: str: string indicating current delay tracking mode

`get_lo_fshift_list`(*return_in_hz=True*)

Get a list of the LO Frequency-shifts applied for all streams.

Parameters

`return_in_hz` (*`bool`*) – return the frequency shift in Hz if True. Else return a tuple (phase_step, scale), indicating that the underlying phase is being incremented by phase_step*2pi radians every 2**scale ADC samples.

`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.

`get_status_delay_tracking`()

Get a dictionary describing the state of the delay tracking thread.

Returns:

dict: {

'is_alive': bool|None (the thread is alive or None if the thread has never been instantiated), 'switch_set': bool (the thread's switch is set), 'ok': bool (is_alive == switch_set)

}

`get_status_dts_monitor()`

Get a dictionary describing the state of the DTS monitor thread.

Returns:

dict: {
'is_alive': bool|None (the thread is alive or None if the thread has never been instantiated), 'switch_set': bool (the thread's switch is set), 'ok': bool (is_alive == switch_set)

}

`get_status_tx()`

Returns a status dictionary of whether each eth is transmitting as it is supposed to be.

`hostname`

hostname of the F-Engine's host SNAP2 board

`initialize(read_only=True, allow_unlocked_dts=False)`

Call the `initialize methods of all underlying blocks, then optionally issue a software global reset. Raises RuntimeErrors if a block is not ok.

Parameters

- **read_only** (*bool*) – If True, call the underlying initialization methods in a read_only manner, and skip software reset.
- **allow_unlocked_dts** (*bool*) – If True, do not initialize the dts block

`is_connected()`

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

Return type
bool

`list_blocks()`

Returns
A list of block names which may be controlled

`load_received_phase_calibration_values(phase_cal_to_load)`

To avoid repeated code, function to load phase calibration values to the F-Engine.

Parameters
phase_cal_to_load (*ndarray{float}*) – an n_streams by n_chans list of phase calibration values in radians.

`logger`

Python Logger instance

`monitor_dts(poll_time_s=0.05)`

Monitor the DTS timing state, and disable Ethernet output if it looks broken.

Parameters
poll_time_s (*float*) – How often to poll the DTS state registers, in seconds

`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*)

Program an .fpg file to an F-engine FPGA.

Parameters

fpgfile (*str*) – The .fpg file to be loaded. Should be a path to a valid .fpg file. If None is given, the .fpg path provided at FEngine instantiation-time will be loaded.

read_chan_dest_ips_as_json(*portnum*)

Reads the headers of the packetizer block and constructs a summative report of the channels' destination-IPs, as a json string.

Parameters

portnum (*int*) – Which zero-indexed output interface to query.

set_delay_tracking(*delays*, *delay_rates*, *phases*, *phase_rates*, *sideband*, *fshifts*, *load_time=None*, *clock_rate_hz=2048000000*, *invert_band=False*)

Set the delays for this F-Engine once. If no load_time is provided, delays are uploaded to the F-Engine immediately. :param delays: 4-tuple of delays for X and Y polarizations for both tunings. Each value is

the delay, in nanoseconds, which should be applied at the appropriate time. Whole ADC sample delays are implemented using a coarse delay, while sub-sample delays are implemented as a post-FFT phase rotation.

Parameters

- **delay_rates** (*list{float}*) – 4-tuple of delay rates for X and Y polarizations for both tunings. Each value is the delay rate, in nanoseconds per second. This is the incremental delay which should be added to the current delay each second. Internally, delay rate is converted from nanoseconds-per-second to samples-per-spectra. Firmware delays are updated every 4 spectra.
- **phases** (*list{float}*) – 4-tuple of phases for X and Y polarizations for both tunings. Each value is the phase, in radians.
- **phase_rates** (*list{float}*) – 4-tuple of phase_rates for X and Y polarizations for both tunings. Each value is the rate of change of phase, in radians per second. This is the incremental phase which should be added to the current phase each second.
- **sideband** (*ndarray{int}*) – 2-length ndarray of the sideband value applied to each tuning.
- **fshifts** (*ndarray{float}*) – 4-length ndarray of the lo frequency shifts in Hz applied to each stream in the *lo* block.
- **load_time** (*float*) – a unix time in seconds at which to load the delay values provided to the F-Engine.
- **clock_rate_hz** (*int*) – ADC clock rate in Hz. If None, the clock rate will be computed from the observed PPS interval, which could fail if the PPS is unstable or not present.
- **invert_band** (*bool*) – If True, invert the gradient of the phase-vs-frequency channel. I.e., apply a fractional delay which is the negative of the physical delay.

set_delay_tracking_mode(*delay_mode*)

Sets up internal delay tracking mode according to the provided mode.

Parameters

delay_mode (*string or boolean*) – The delay tracking mode, one of the following with False being the fallback - True - Geometric delay tracking including calibration delays "fixed-only" - Calibration delays with no geometric delay tracking "half-off" - "half-cal" - "half-phase" - "half-corrected-phase" - "full-corrected-phase" - Full calibration + delay / phase tracking False - No delays are loaded

set_delays(*delay_to_load*, *delay_rate_to_load*, *phase_to_load*, *phase_rate_to_load*, *sideband*, *clock_rate_hz=2048000000*, *invert_band=False*)

Transform the argument delays, delay rates, phases and phase rates before uploading them to the F-Engine. After sufficient testing, this function will be absorbed into delay_tracking to reduce complexity.

Parameters

- **delay_to_load** (*ndarray{float}*) – 4-tuple of delays for X and Y polarizations for both tunings. Each value is the delay, in nanoseconds, which should be applied at the appropriate time. Whole ADC sample delays are implemented using a coarse delay, while sub-sample delays are implemented as a post-FFT phase rotation.
- **delay_rate_to_load** (*ndarray{float}*) – 4-tuple of delay rates for X and Y polarizations for both tunings. Each value is the delay rate, in nanoseconds per second. This is the incremental delay which should be added to the current delay each second. Internally, delay rate is converted from nanoseconds-per-second to samples-per-spectra. Firmware delays are updated every 4 spectra.
- **phase_to_load** (*ndarray{float}*) – 4-tuple of phases for X and Y polarizations for both tunings. Each value is the phase, in radians.
- **phase_rate_to_load** (*ndarray{float}*) – 4-tuple of phase_rates for X and Y polarizations for both tunings. Each value is the rate of change of phase, in radians per second. This is the incremental phase which should be added to the current phase each second.
- **sideband** (*list{int}*) – 2-long list of integers which are the VLA prescribed sideband factors.
- **clock_rate_hz** (*int*) – ADC clock rate in Hz. If None, the clock rate will be computed from the observed PPS interval, which could fail if the PPS is unstable or not present.
- **invert_band** (*bool*) – If True, invert the gradient of the phase-vs-frequency channel. I.e., apply a fractional delay which is the negative of the physical delay.

Returns:

fshifts: ndarray{floats} the fshift values in Hz used in the phase correction.

set_equalization(*eq_start_chan=100*, *eq_stop_chan=400*, *start_chan=50*, *stop_chan=450*, *filter_ksize=21*, *target_rms=0.2*)

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 1.0 is the saturation level. I.e., an RMS of 0.125 means that the RMS is one LSB of a 4-bit signed signal, or 16 LSBs of an 8-bit signed signal

set_lo_fshift_list(*lo_fshift_list*, *skip_if_equal=False*)

Apply and load the provided list of LO FShift values, blocking until after the load.

Parameters

- **lo_fshift_list** (*List*) – The list of LO FShifts in Hz to apply, in order of streams.
- **skip_if_equal** (*bool*) – Whether or not to only set the values if they are different.

start_delay_tracking()

Start the thread running *delay_tracking* if the thread is dead. Otherwise ignore. In addition to just starting the thread, this function subscribes to the redis channels on which the geometric delays are broadcast and on which the thread is notified of calibration delay updates in META_calibrationDelays hash.

start_dts_monitor()

Start the thread running *monitor_dts* if the thread is dead. Otherwise ignore.

stop_delay_tracking(*timeout_s=5*)

End the thread running *delay_tracking* if the thread is alive. Otherwise ignore. In addition to just ending the thread, this function unsubscribes from the redis channels that are listened to for delays when the thread is running. Furthermore, the *delay* and *phaserotate* blocks are re-initialised.

stop_dts_monitor(*timeout_s=5*)

End the thread running *monitor_dts* if the thread is alive. Otherwise ignore.

tx_enabled()

Returns *[eth.tx_enabled() for eth in self.eths]*.

tx_packet_rate()

Returns *[eth.get_packet_rate() for eth in self.eths]*.

update_packet_header_destination_ports(*ipport_map*)

Iterates through the packet headers of all packetizers, updating the destination port value of any headers that have an IP value in the given map.

Parameters
ipport_map (*dict*) – dotted-quad string IP adress keyed integer ports

Returns
a list of booleans indicating if the corresponding packetizer had its headers re-populated

Return type
list

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 cosmic_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_connected_antname()

Fetch the connected antenna name.

Return self.antname
The name of the connected antennna.

Rtype str

get_firmware_type()

Read the firmware type register and return the contents as an integer.

Return type
Firmware type

Rtype str

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.
- host (str) : The host name of this board.
- antname (str) : The name of the antenna connected to 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_type (int): The firmware type ID 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

`set_connected_antname(antname)`

Set the connected antenna name.

Parameters
antname (*str*) – The antenna name.

4.2.3 Timing Control

The Sync control interface provides an interface to configure and monitor the multi-ADM-PCIe-9H7 timing distribution system.

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

The Sync block controls internal timing signals.

Parameters

- **host** (*casperfpga.CasperFpga*) – CasperFpga interface for host.
- **name** (*str*) – Name of block in Simulink hierarchy.
- **clk_hz** (*int*) – The FPGA clock rate at which the DSP fabric runs, in Hz.
- **logger** (*logging.Logger*) – Logger instance to which log messages should be emitted.

arm_noise()

Arm noise generator resets.

arm_sync(*wait=True*)

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

Parameters
wait (*bool*) – If True, wait for a sync to pass before returning.

check_timekeeping(*sync_rate_hz=20*, *verbose=True*)

Check timekeeping logic, returning True if the system looks OK, and False otherwise.

Parameters

- **sync_rate_hz** (*int*) – Expected sync rate, in Hz
- **verbose** (*bool*) – If True, log errors. If False don't

Tests: 1. Sync pulse period = 1./sync_period_hz 2. Sync arrived in last 1.5*sync_period_ms 3. Internal telescope time within 50ms of NTP

Returns
True if OK, False otherwise

Return type
bool

count_ext()

Returns
Number of external sync pulses received.

Rtype int

disable_error_detect()

Disable error output, which goes high whenever missing syncs or syncs with inconsistent period are detected.

enable_error_detect()

Enable error output, which goes high whenever missing syncs or syncs with inconsistent period are detected.

get_error_count()

Returns
Number of sync errors.

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.
- 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_ntp_offset()`

Get the offset of the FPGA's telescope time from NTP, in seconds.

Returns
Offset of tt from NTP in seconds

Return type
float

`get_tt_of_ext_sync()`

Get the internal TT at which the last sync pulse arrived.

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

`get_tt_of_sync()`

Get the internal TT of the last system sync event.

Returns
tt. The internal TT of the last sync.

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.

`offset()`

Returns
Nanosecond offset.

Return type
float

`period()`

Returns
The number of FPGA clock ticks between the last two external sync pulses.

Rtype int

`reset_error_count()`

Reset internal error counter 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=None, offset_ns=0.0, sync_clock_factor=1)`

Arm sync trigger receivers, having loaded an appropriate telescope time.

Parameters

- **fs_hz** (*int*) – The FPGA DSP clock rate, in Hz. Used to set the telescope time counter. If None is provided, self.clk_hz will be used.
- **offset_ns** (*float*) – Nanoseconds offset to add to the time loaded into the internal telescope time counter.

Returns
next_sync_clocks: The value of the TT counter at the arrival of the next sync pulse. Or, *None*, if the TT counter was loaded late.

Rtype int

`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_sync()`

Block until a sync has been received.

Returns
False if timeout occurs, True otherwise.

4.2.4 DTS Control

The `Dts` control interface allows link training of the DTS->FPGA data link.

class cosmic_f.blocks.dts.**Dts**(*host*, *name*, *nlanes=12*, *lane_map=[0, 1, 3, 7, 6, 8, 2, 4, 5, 9, 11, 10]*, *logger=None*)

Instantiate a control interface for a DTS interface 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.
- **nlanes** (*int*) – Number of parallel DTS lanes used for this interface.
- **lane_map** (*list of int*) – How to reorder DTS lanes to assemble data. A list with `nlanes` entries, where entry `i` defines the physical lane which should be considered to have index `i`. This array is defined by the physical connectivity of the FPGA.

align_lanes(*mux_factor=4*, *retries=0*)

Align the multiple lanes of the DTS interface by advancing/delaying streams so that their sync pulses occur at the same FPGA clock cycle.

Parameters

- **mux_factor** (*int*) – Ratio of FPGA clock to DTS 160-bit frame clock
- **retries** (*int*) – Number of retry attempts

check_lane_ids()

Read the DTS encoded lane IDs, and the currently loaded lane map, and verify that there combine to something plausible.

Returns
True if things look OK. False otherwise

Return type
bool

get_gty_lock_state()

Get the lock state of the DTS's underlying transceivers. Returns a bit array, where bit `i` is 1 if lane `i` is locked, and `0` otherwise. A lane is locked if the transceiver IP is successfully recovering a clock from the data stream.

Return locked
bit array

Rtype locked
int

get_lane_ids()

Get the DTS lane IDs, as encoded in the underlying data streams.

get_lane_map()

Get the currently loade lane map.

Returns
Lane map

Return type
list

`get_lock_state()`

Get the lock state of the DTS interface. Returns a bit array, where bit `i` is 1 if lane `i` is locked, and `0` otherwise. A lane is locked if the DTS receiver has correctly inferred data frame edges from the data stream sync pattern.

Return locked
bit array

Rtype locked
int

`get_status(`*lanes='all'*`)`

Get a dictionary of status values, with optional warning of error flags. To be overridden by individual blocks

Returns
(status_dict, flags_dict) tuple. *status_dict* is a dictionary of status key-value pairs, defined on a per-block basis. flags_dict is a dictionary with all, or a sub-set, of the keys in *status_dict*. The values held in this dictionary should be as defined in *error_levels.py*.

`initialize(`*read_only=False*`)`

Individual blocks should override this method to configure themselves appropriately

Parameters
read_only (*bool*) – If False, initialize blocks in a way that might change the configuration of running hardware. If True, read runtime info from blocks, but don't change anything.

`mute()`

Internally disable data output.

`reset()`

Reset underlying interface.

`reset_stats()`

Reset internal statistics counters

`unmute()`

Internally enable data output.

4.2.5 Input Control

`class cosmic_f.blocks.input.Input(`*host*, *name*, *n_inputs=4*, *n_bits=12*, *n_parallel_samples=8*, *logger=None*`)`

Instantiate a control interface for an Input block. This block allows switching data inputs between constant-zeros, digital noise, and ADC inputs.

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_inputs** (*int*) – Number of independent inputs.

- **n_parallel_samples** (*int*) – Number of parallel samples per input stream.
- **n_bits** (*int*) – Number of bits per ADC sample.

Variables

- **n_inputs** – Number of inputs 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_input x 2**n_bits]` list of histogram data.

`get_bit_stats()`

Get the mean, RMS, and mean powers of all ADC inputs.

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

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

`get_histogram(`*input*, *sum_cores=True*`)`

Get a histogram for an ADC input.

Parameters

- **input** (*int*) – ADC input 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 input `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 input `n`, where `n` is a two-digit integer starting at 00. In units of (ADC LSBs)**2.
- rms<n> (float) : RMS of input input `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 input `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 input), 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`(*input*)

Plot a histogram.

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

`print_histograms`()

Print histogram stats to screen.

`use_adc`(*input=None*)

Switch input to ADC.

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

`use_noise`(*input=None*)

Switch input to internal noise source.

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

`use_zero`(*input=None*)

Switch input to zeros.

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

4.2.6 Noise Generator Control

class cosmic_f.blocks.noisegen.**NoiseGen**(*host*, *name*, *n_noise=5*, *n_outputs=64*, *n_parallel_samples=8*, *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_parallel_samples** (*int*) – Number of parallel samples per noise stream.
- **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 seed 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.

4.2.7 Sine Generator Control

class cosmic_f.blocks.sinegen.**SineGen**(*host*, *name*, *n_sine=2*, *n_outputs=4*, *n_samples=8192*, *logger=None*)

Sine Generator controller

This block controls a digital TVG, which can generate programmable ADC data

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_sine** (*int*) – The number of independent test vector brams.
- **n_outputs** (*int*) – The number of output channels from the block.
- **n_samples** (*int*) – The number of samples in the test vector buffer.

assign_output(*output*, *n*)

Assign an output channel to a given data stream.

Parameters

- **output** (*int*) – The index of the output stream to be assigned.
- **n** (*int*) – The index of the sine stream to assign to *output*. Use *None* to turn off test vector insertion.

get_output_assignment(*output*)

Get the index of the sine stream assigned to an output.

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

Returns
The index of the sine stream to assign to *output*. Or, *None* if no test vector is being inserted

Return type
int

get_status()

Get status and error flag dictionaries.

Status keys:

- output_assignment<n> (int): The sine generator ID currently assigned to output stream n, where n is a two-digit integer starting at 00. None, if the generators are not being used.

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, turn off test vector insertion If True, do nothing.

write_sine(*n*, *period*, *amp=0.1*, *cos=False*)

Write a sine wave.

Parameters

- **n** (*int*) – Which sine wave generator to write to.
- **period** (*int*) – Sine period in ADC samples
- **cos** (*bool*) – If True, use a cosine, rather than a sine.
- **amp** (*float*) – Sine wave amplitude. 1 will saturate the logic.

4.2.8 Delay Control

class cosmic_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

disable_load()

Disable the loading of values when target TT is reached

enable_load()

Enable the loading of values when target TT is reached

force_load()

Force immediate load of all delays.

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.

get_target_load_time()

Get currently set target load time.

Returns
target_tt

Return type
int

`get_time_to_load()`

Get number of FPGA clocks until load trigger. The returned value will be negative if the trigger time is in the past.

Returns
time_to_load

Return type
int

`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, force=False)`

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.

`set_target_load_time(value, enable_trig=True)`

Load a new target TT

Parameters

- **value** (*int*) – Telescope time to load
- **enable_trig** (*bool*) – If True, enable the triggering of a sync pulse at this time. Else, set the load_time registers but don't enable the triggering system.

4.2.9 PFB Control

`class cosmic_f.blocks.pfb.Pfb(host, name, nchan=1024, nchan_parallel=8, logger=None)`

`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".

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.

`sel_band`(*start_chan=0*)

Select which half of the band is output.

Parameters
start_chan (*int*) – First channel to be output. Half the spectrum will be output, starting with this channel.

4.2.10 Auto-correlation Control

`class cosmic_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, return_list=False)`

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.
- **return_list** (*Bool*) – If True, return a list else numpy.array

Returns
Float32 2D list of dimensions [POLARIZATION, FREQUENCY CHANNEL] containing autocorrelations with accumulation length divided out.

Return type
list

`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 cosmic_f.blocks.corr.**Corr**(*host*, *name*, *acc_len=1024*, *logger=None*, *n_chans=1024*, *n_signals=64*, *n_parallel_chans=4*)

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_parallel_chans** (*int*) – Number of frequency channels processed in parallel.
- **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*, *return_list=False*)

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.
- **return_list** (*Bool*) – If True, return data as a dictionary of lists, keyed by 'real', and 'imag'. Else, return a numpy array of complex data.

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

Return type
numpy.array or dict(str:list, str:list)

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 cosmic_f.blocks.eqtvg.EqTvg`(*host*, *name*, *n_inputs=64*, *n_chans=4096*, *n_serial_inputs=16*, *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_inputs** (*int*) – Number of independent inputs which may be emulated
- **n_serial_inputs** (*int*) – Number of independent inputs sharing a data bus

- **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_input_tvg(*input*, *makecomplex=False*)

Read the test vector loaded to an ADC input.

Parameters

- **input** (*int*) – Index of input 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_input()

Write a constant to all the channels of a input, with input *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_input_tvg`(*input*, *test_vector*)

Write a test vector pattern to a single signal input.

Parameters

- **input** (*int*) – Index of input 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 cosmic_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*)

Get the coefficients currently loaded. Reads the actual coefficients from the board.

Parameters
stream (*int*) – ADC stream index to query.

Returns
(coeffs, binary_point). `coeffs` is an list of `self.n_coeffs` integer coefficients currently being applied. `binary_point` is the position of the binary point wy which these integers are scaled on the FPGA.

Return type
(list, int)

`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. Clipping and saturation will be applied before loading.

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 cosmic_f.blocks.chanreorder.**ChanReorder**(*host*, *name*, *n_ants=4*, *n_times=64*, *n_chans=1024*, *n_parallel_chans=4*, *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_ants** (*int*) – Number of dual-pol ants handled by this block.
- **n_chans** (*int*) – Number of channels in the system.
- **n_parallel_chans** (*int*) – Number of channels reordered in parallel.

get_antchan_order(*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

`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*.

`set_antchan_order`(*ant_order*, *chan_order*)

Reorder the antenna-channels such that antenna-channel ant_order[i]-chan_order[i] emerges out of the reorder in position i.

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

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

In the underlying firmware, input data order is assumed to be: serial_channel x antenna x [parallel channel x dual-pol sample] i.e. serial_channel x antenna <parallel dimensions>

Parameters

- **ant_order** (*list of int*) – The antenna order to which data should be mapped. I.e., if *ant_order[0] = 1*, then the first ant-chan out of the F-engine will be antenna 1. The order map should meet the above criteria. A ValueError exception will be raised if it does not.
- **chan_order** (*list of int*) – The chan order to which data should be mapped. I.e., if *chan_order[0] = 16*, then the first ant-chan out of the F-engine will be channel 16. The order map should meet the above criteria. A ValueError exception will be raised if it does not.

4.2.15 Packetization Control

`class cosmic_f.blocks.packetizer.Packetizer`(*host*, *name*, *n_chans=4096*, *n_ants=4*, *sample_rate_mhz=200.0*, *sample_width=1*, *word_width=64*, *line_rate_gbps=100.0*, *n_time_packet=16*, *granularity=32*, *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_ants** (*int*) – Number of dual-polarization inputs streams in the system.
- **sample_rate_mhz** (*float*) – ADC sample rate in MHz. Used for data rate checks.
- **sample_width** (*int*) – Sample width in bytes (e.g. 4+4 bit complex = 1)
- **word_width** (*int*) – Ethernet interface word width, in bytes
- **line_rate_gbps** (*float*) – Link speed in gigabits per seconds.
- **n_time_packet** (*int*) – Number of time samples per packet
- **granularity** (*int*) – The number of words per packetizer data block.

get_packet_info(*n_pkt_chans*, *n_chan_send*, *n_ant_send*, *occupation=0.985*, *chan_block_size=4*)

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

Parameters

- **n_pkt_chans** (*int*) – The number of channels per packet.
- **n_chan_send** (*int*) – The number of channels to send per input
- **n_ant_send** (*int*) – The number of antennas to send
- **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, word_indices, antchan_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.

word_indices
[list of range()] The range of input word indices this packet will send. Eg: [range(1,129), range(1025,1153), range(2049,2177), ... etc]. Data to be sent should be places in these ranges. Data words outside these ranges won't be sent anywhere.

antchan_indices
[list of range()] The range of input antchan indices this packet will send. Eg: [range(1,129), range(1025,1153), range(2049,2177), ... etc]. Data to be sent should be places in these antchan ranges. Data words outside these ranges won't be sent anywhere.

write_config(*packet_starts*, *packet_payloads*, *channel_indices*, *antenna_ids*, *dest_ips*, *dest_ports*, *nchans_per_packet*, *enable=None*)

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 marked as valid (see *get_packet_info()*)
- **channel_indices** (*list of ints*) – Header entries for the channel field of each packet to be sent
- **antenna_ids** – Header entries for the antenna field (feng_id) of each packet to be sent
- **dest_ips** – list of str IP addresses for each packet to be sent. If an IP is '0.0.0.0', the corresponding packet will be marked invalid.
- **dest_ports** (*list of int*) – UDP destination ports for each packet to be sent.
- **nchans_per_packet** (*list of int*) – Number of frequency channels per packet sent.
- **enable** (*list of bool*) – List of booleans to enable each packet. If None, all packets are assumed valid.

All parameters should have identical lengths.

4.2.16 Ethernet Output Control

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

Instantiate a control interface for a 100 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_eth_core_details(read_arp=False)`

Get Ethernet core parameters, returned as a dictionary of strings.

Parameters
read_arp (*Bool*) – If True, read the ARP table.

Returns
Dictionary of status keys, with each holding a string value.

`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 512-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 and disable and reset core. 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.

`tx_enabled()`

Ethernet transmission enabled.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

H

I

L

M

N

O

P

R

S

T

U

W