CASM SNAP F-Engine Documentation

Release casm_snap_f-8423635:casm_f-84236357-dirty

Jack Hickish

Oct 29, 2024

CONTENTS:

Contents:

CHAPTER

ONE

F-ENGINE SYSTEM OVERVIEW

1.1 Overview

The CASM F-Engine firmware is designed to run on a SNAP[1] FPGA board, and provides channelization of 12 analog data streams, sampled at up to 250 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:

[1] See the CASPER SNAP wiki page

Parameter	Value	Notes
Number of analog inputs	12	
Maximum sampling rate	250 Msps	Limited by ADC speed & timing constraint target
Test inputs	Noise; zeros	Firmware contains 2 independent gaussian noise generators. Any of the 12 data streams may be replaced with any of these digital noise sources, or zeros.
Delay compensation	<=7 samples	Programmable per- input between 0 and 7 samples. Mostly useful for testing
Polyphase Filter Bank Channels	4096	
Polyphase Filter Bank Window	Hamming; 4-tap	
Polyphase Filter Bank Input Bitwidth	8 bits	
FFT Coefficient Width	18 bits	
FFT Data Path Width	18 bits	
Post-FFT Scaling Coefficient Width	16	
Post-FFT Scaling Coefficient Binary Point	4	
Number of Post-FFT Scaling Coefficients	256	One coefficient per analog input. One coefficient per 16 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 10Gb/s (including protocol overhead). 3072 channels = approx 9Gb/s + overhead

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. 1.1.

1.1.1 Block Descriptions

Each block in the firmware design can be controlled using an API described in Section 2. Here the basic functionality of each block is described.

Platform Config (*XSG*)

The platform configuration block – often called the MSSGE or XSG config block by CASPER collaborators – defines the high-level parameters of the firmware design. The CASM firmware is configured to target the SNAP1 FPGA board, with the DSP chain clocking synchronously from the ADC clock at the sample rate of 250 MS/s.

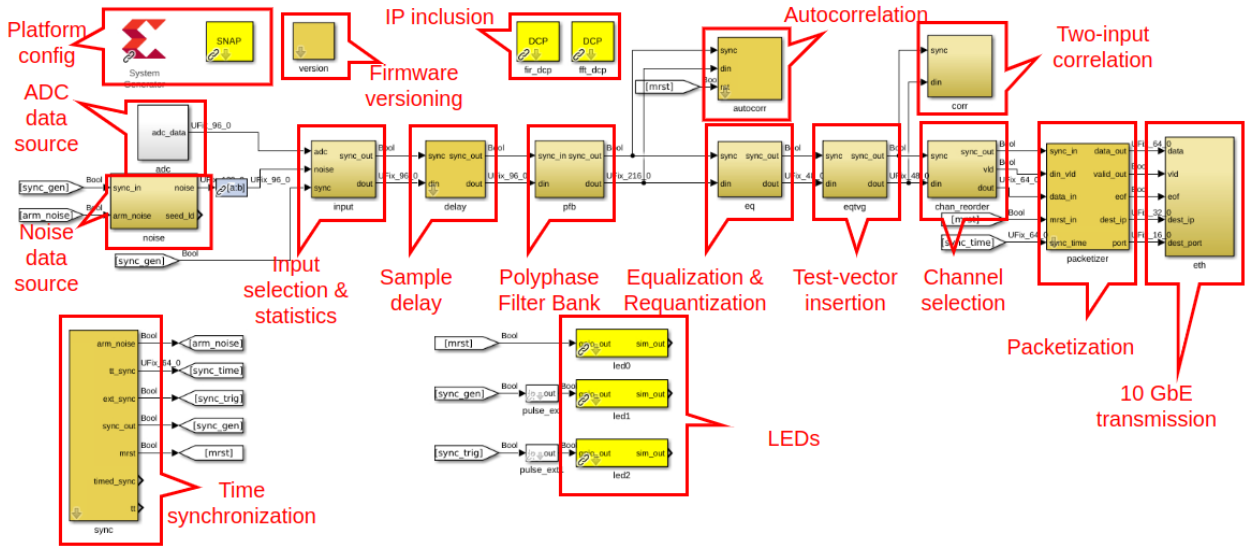

Fig. 1.1: F-Engine top-level Simulink diagram.

Firmware Versioning (*version*)

The *version* block in the Simulink design contains information about the firmware which can be read at runtime. This information includes a user-defined version number (formatted as *major*.`minor`.`revision`.`bugfix`) as well as the firmware build time. The *version* block is used by the control software to detect mismatches between software and firmware versions.

ADC Data Source (*adc*)

The ADC data source block – *adc* in the Simulink diagram – encapsulates an underlying interface to the SNAP ADCs. It is configured to operate the SNAP ADCs in 12-channel mode, with each channel sampled at 250 MS/s This block is the source of 8-bit ADC samples (presented as a parallel 96-bit bus of 12-ADC channels) which feed the CASM DSP pipeline.

Noise Data Source (*noise*)

The Noise data source block – *noise* in the Simulink diagram – implements a pair of independent white gaussian noise generators. Each noise generator can be seeded with a user-supplied value, and generates a pair of uncorrelated noise streams. The *noise* block outputs a stream of 12 parallel data channels, designed such that they have the same format as the 12 ADC channels produced by the *adc* block. Each channel may be populated with data from any one of the 4 internally generated noise streams.

Input Selection & Statistics (*input*)

The input selection and statistics block – *input* in the Simulink diagram – provides two key functions. First, the block accepts as input both noise and ADC inputs, and can switch between these on a per-ADC-channel basis so that the downstream pipeline processes either real, or simulated data. The block can also replace any ADC data stream with a constant zero value, which may be useful for testing. Second, the block provides bit statistics of the selected data streams. Statistics include firmware-calculated mean, variance, and RMS. A histogram of samples and snapshot of a small burst of samples is also provided.

Sample Delay (*delay*)

The *delay* block allows runtime-configurable per-channel delays of up to 8 ADC samples to be inserted into the ADC data streams. While these small delays are unlikely to be useful for cable-delay correction, they can be useful in testing. For example, in conjuction with the *noise* block, the firmware can be used to simulate two ADC channels having identical, but time-delayed, data streams.

Polphase Filter Bank (*pfb*)

The *pfb* block contains a 4-tap, 4096 channel PFB-based channelizer. To speed compilation, this block is built with pre-compiled IP, included in the design via the *fir_dcp* and *fft_dcp* blocks. The output of the *pfb* block is a stream of 18+18 bit complex-integer data for each ADC data. Data are interpretted with 17 bits to the right of the binary point – i.e., values are between +/-1. A full 4096-channel spectrum is presented over 4096 clock cycles for the 6 even-numbered ADC channels in parallel. The following 4096 clock cycles carry spectra from the odd-numbered ADC channels.

Autocorrelation (*autocorr*)

The *autocorr* block provides the ability to read an accumulated autocorrelation spectrum for any of the ADC inputs. The accumulation length is runtime-programmable, and results are integrated with floating point precision.

Equalization & Requantization (*eq*)

The *eq* block multiplies each 18+18-bit complex number by a programmable 16-bit unsigned coefficient. Coefficients are interpreted with 5 bits to the right of the binary point – i.e. the maximum coefficient is 2048, and the coefficient precision is 0.03125. After multiplying data, the resulting values are quantized (with a round-to-even and symmetric saturation scheme) to 4-bit precision. 4-bit values are interpretted with 3 bits to the right of the binary point – i.e., values are between +/-0.875. Spectra from individual ADC channels are multiplied by independent coefficient sets. Each coefficient in a set is applied to 8 consecutive frequency channels – i.e., 512 independent coefficients may be provided per ADC channel.

Test-vector Insertion (*eqtvg*)

The *eqtvg* block allows the channelized data streams to be replaced with a runtime-programmable test value. Independent values may be provided for each ADC channel and for each frequency channel. Values are repeated with each consecutive spectrum. This functionality is particularly useful for testing downstream channel selection and transmission logic.

Two-Input Correlation (*corr*)

The *corr* block implements a 2 input correlator. Each input may be selected from any of the 12 ADC inputs, and the resultant correlation is computed and accumulated with a runtime-programmable integration time. The *corr* block can be used with the *noise* and *delay* blocks to verify that the PFB is operating correctly, by correlation identical time-delayed copies of the same noise stream. The *corr* block can also be used with the *autocorr* block to compare signal autocorrelation power before and after equalization, which can aid in setting EQ coefficient levels.

Channel Selection (*chan_reorder*)

The channel selection block, *chan_reorder* provides a runtime-programmable frequency channel reorder, with the end result that 3/4 of the 4096 generated frequency channels are collected ready for packetization and UDP transmission.

Packetization (*packetizer*)

The *packetizer* block inserts application packet headers in the channel down-selected data streams, and defines how large packets are, and to which IP addresses they are transmitted. Each packet contains data from all 12 ADC channels, and any multiple of 8 frequency channels (though smaller packets incur more protocol overhead and the total data rate should be kept below 10 Gb/s). The maximum packet size is 8192 kB. Each packet contains a header describing

- the timestamp of the sample in the packet
- the channel index of the first channel in the packet
- the ID of the SNAP board from which the packet is being sent

10 GbE Transmission (*eth*)

The *eth* block encapsulates a 10 Gb/s Ethernet interface. This interface implements a UDP / IP / Ethernet stack. Statistics are provided to determine data and packet transmission rates

Time syncronization (*sync*)

The *sync* block locks the FPGA's timestamping logic to an externally provided Pulse-Per-Second (PPS) pulse, which is assumed to be locked to GPS time.

LEDs

LEDs are provided to indicate the following events:

1. LED0: Goes high when the board is in a reset condition. This happens prior to a synchronization event.
2. LED1: Goes high for 2^26 clock cycles (approximately 268 ms for an ADC sample rate of 250 MS/s) immediately following a synchronization event.
3. LED2: Goes high for 2^26 clock cycles (approximately 268 ms for an ADC sample rate of 250 MS/s) immediately following the arrival of a PPS pulse.

1.1.2 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 2 for a full software API description

```
# Import the SNAP F-Engine library
from casm_f import snap_fengine

# Instantiate a SnapFengine instance to a board with
# hostname 'snap'
f = snap_fengine.SnapFengine('snap')

# Program a board
f.program(path-to-fpg-file) # Load an fpg firmware binary

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

CHAPTER

TWO

CONTROL INTERFACE

2.1 Overview

A Python class `SnapFengine` 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 `SnapFengine` class provides an easy way to probe board status for a SNAP board on the local network.

2.2 `SnapFengine` Python Interface

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

```
# Import the SNAP F-Engine library
from casm_f import snap_fengine

# Instantiate a SnapFengine instance to a board with
# hostname 'snap'
f = snap_fengine.SnapFengine('snap')

# Program a board (if it is not already programmed)
# and initialize all the firmware blocks
if not f.fpga.is_programmed():
  f.program(path-to-fpg-file) # Load an fpg firmware binary
  # Initialize firmware blocks, including ADC link training
  f.initialize(read_only=False)

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

# 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']
```

(continues on next page)

(continued from previous page)

```
# Grab some ADC data from the first ADC input
adc_data = f.adc.get_snapshot()
print(adc_data.shape) # returns (12 [ADC inputs], 1024 [time samples])
```

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

2.2.1 Top-Level Control

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

2.2.2 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 CASM F-Engine firmware design.

class casm_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.

check_firmware_support()

Check the software packages firmware support version against the running firmware version.

Returns
True if firmware is supported, False otherwise.

Rtype bool

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.
- timestamp (str) : The current time, as an ISO format string.
- **fpga_clk_mhz (float)**
  [The estimated FPGA clock rate in MHz. This] is the same as the estimated ADC sampling rate.

  Flagged with an error if not between 190 and 200 MHz
- 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_supported (bool) : True if the running firmware is supported by this software. False (and flagged as an error) otherwise.
- 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

2.2.3 Power Monitoring

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

2.2.4 Timing Control

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

class casm_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*, *current_msb=None*)

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.

`load_timed_sync`(*target_tt*)

Set a timed sync to be emitted when TT = `target_tt`

Parameters
target_tt (*int*) – Target telescope time, in FPGA clocks.

`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*, *quiet=False*`)`

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.
- **`quiet`** (*bool*) – If True, supress log messages

`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.

Returns
Sync count after sync has passed

Rtype int

2.2.5 ADC Control

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

class casm_f.blocks.adc.**Adc**(*host*, *name*, *logger=None*, *sample_rate_mhz=250.0*)

Instantiate a control interface for an ADC block.

Parameters

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

initialize(*read_only=False*)

Initialize the block.

Parameters

read_only (*bool*) – If True, do nothing. If False, configure the ADC.

2.2.6 Input Control

class casm_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_snapshot`(*sw_trigger=True*)

Get a snapshot of ADC samples

Parameters
sw_trigger (*bool*) – If True, trigger an instant sample capture, rather than starting capture on an external hardware pulse.

Returns
Array of shape [n_signals, n_samples] containing ADC samples

Rval
numpy.ndarray

`get_status`()

Get status and error flag dictionaries.

Status keys:

- switch_position<n> (str) : Switch position ('noise', 'adc', 'zero' or 'counter') 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.

`plot_snapshot`(*n_sample=-1*, *sw_trigger=True*)

Plot a snapshot of ADC samples

Parameters
sw_trigger (*bool*) – If True, trigger an instant sample capture, rather than starting capture on an external hardware pulse.

`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_counter`(*stream=None*)

Switch input to counter.

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.

2.2.7 Noise Generator Control

class casm_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.

2.2.8 Delay Control

class casm_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.

2.2.9 PFB Control

`class casm_f.blocks.pfb.Pfb(host, name, 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.

`rst_stats()`

Reset overflow event counters.

`set_fft_shift(shift)`

Set the FFT shift schedule.

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

2.2.10 Flagging Control

2.2.11 Auto-correlation Control

`class casm_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

2.2.12 Correlation Control

class casm_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

2.2.13 Post-FFT Test Vector Control

class casm_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.

2.2.14 Equalization Control

class casm_f.blocks.eq.**Eq**(*host*, *name*, *n_inputs=12*, *n_parallel_inputs=6*, *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_inputs** (*int*) – Number of independent inputs to which coefficients are applied
- **n_parallel_inputs** (*int*) – Number of parallel inputs to which coefficients are applied
- **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(*inputid*, *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

- **inputid** (*int*) – ADC inputid 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(*inputid*, *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

- **inputid** (*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.

2.2.15 Channel Selection Control

class casm_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.

2.2.16 Packetization Control

class casm_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.

2.2.17 Ethernet Output Control

class casm_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 256-bit words marked as valid for transmission.

- tx_ctr: Count of transmission End-of-Frames marked valid.
- gbps: Approximate Gbits/s transmission rate

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.

CHAPTER

THREE

OUTPUT DATA FORMATS

This section defines the output packet formats for each of the pipeline output data products. Unless otherwise specified, all data products are transmitted in network- (i.e. big-) endian format.

Packet sizing is partially determined by the pipeline configuration. Specifically:

- `NCHAN` – The number of channels per packet.
- `NINPUTS` – The number of input RF signals on an FPGA board.

In this design:

- `NCHAN` = TBC
- `NINPUT` = 12

3.1 Voltage Packets

Data from the SNAP board(s) are transmitted as a series of UDP packets, with each packet carrying data for multiple frequencies and multiple inputs. Each packet has an 8 byte header followed by a payload of 4+4 bit complex signed integer data.

Packets are formatted as:

```
struct voltage_packet {
};
```

Packet fields are as follows:

Field	Format	Units	Description
`sync_time`	??	UNIX seconds	The timestamp of the spectra in the packet, referenced to ??
`chan0`	??	channel number	The index of the first frequency channel in this packet's data payload
`input0`	??	input number	The index of the first input in this packet's data payload.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

I

L

M

N

P

R

S

T

U

W