SOUK MKID Readout

Release souk_mkid_readout-9e364da:souk_mkid_readout-v7.4.2.0-66-g9e364dad-dirty

Jack Hickish

Nov 11, 2025

CONTENTS:

CHAPTER

ONE

INSTALLATION

The SOUK MKID Readout pipeline firmware and software is available at https://github.com/realtimeradio/souk-firmware. Follow the instructions here to download and install the pipeline.

1.1 Install Prerequisites

1.1.1 Firmware Requirements

The SOUK MKID Readout firmware can be built with the CASPER toolflow, and was designed using the following software stack:

- Ubuntu 20.04.6 LTS (64-bit)
- MATLAB R2021a
- Simulink R2021a
- MATLAB Fixed-Point Designer Toolbox R2021a
- Xilinx Vivado HLx 2021.2
- Python 3.8.10

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

1.2 Get the Source Code

Specify the repository root directory by defining the `REPOROOT` environment variable, eg:

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

Clone the repository and its dependencies with:

```
# Clone the main repository
cd $REPOROOT
git clone https://github.com/realtimeradio/souk-firmware
# Clone relevant submodules
cd souk-firmware
git submodule init
git submodule update
```

1.3 Create a Local Environment Configuration

Create a local configuration file which specifies the location to which various tools have been installed. An example configuration is given in *$REPOROOT/firmware/startsg.local*:

```
export XILINX_PATH=/data/Xilinx/Vivado/2021.2
export COMPOSER_PATH=/data/Xilinx/Model_Composer/2021.2
export MATLAB_PATH=/data/MATLAB/R2021a
export PLATFORM=lin64
export JASPER_BACKEND=vitis
export CASPER_SKIP_STARTUP_LOAD_SYSTEM=yesplease
export XILINXD_LICENSE_FILE=/home/jackh/.Xilinx/Xilinx.lic
export XLNX_DT_REPO_PATH=/home/jackh/src/souk-firmware/firmware/lib/device-tree-xlnx

# over-ride the MATLAB libexpat version with the OS's one.
# Using LD_PRELOAD=${LD_PRELOAD}:"..." rather than just LD_PRELOAD="..."
# ensures that we preserve any other settings already configured
export LD_PRELOAD=${LD_PRELOAD}:"/usr/lib/x86_64-linux-gnu/libexpat.so"
```

When launching Simulink to modify or compile firmware, use the incantation:

```
cd $REPOROOT/firmware
./startsg <custom_startsg_local_file>
```

CHAPTER

TWO

F-ENGINE SYSTEM OVERVIEW

2.1 Overview

The SOUK MKID readout system is built around multiple instances of a single readout pipeline. Each pipeline has the following capabilities:

1. Digitize a single RF data stream at 5 Gsps (real-sampled).
2. Channelize the resulting 2.5 GHz Nyquist band into 8192 channels each of 610 kHz width, and overlapping by 50%.
3. Select 2048 of 8192 available channels.
4. Mix each channel with an independent, programmable local-oscillator before integrating and storing the result.
5. Output an RF signal with a 2.5 GHz bandwidth (generated from a 5 Gsps real-sampled stream), constructed by frequency multiplexing each of the system's 2048 LOs with a synthesis bank.

A block diagram of a single pipeline is shown in Figure Fig. 2.1.

The readout firmware is defined using the graphical MATLAB Simulink / Xilinx System Generator design tools. As such, the source code very closely resembles the high-level block diagram description of the system.

The Simulink source diagram for a single pipeline is shown in Figure A block diagram of a single pipeline is shown in Figure Fig. 2.2.

Multiple pipelines may be instantiated on a single FPGA processing board. When using an RFSoC processor such as the xczu47p – which has 8 ADC and DAC channels capable of running at 5 Gsps and 9.85 Gsps, respectively – a single chip is, in principle, capable of supporting 8 such pipelines. In practice, other processing limitations mean that a single chip can likely only service 2-3 pipelines.

The top-level Simulink source code for firmware supporting multi-pipelines is shown in Figure Fig. 2.3. Common infrastructure is shared between pipelines, in a processing block whose functionality is shown in Figure Fig. 2.4.

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.

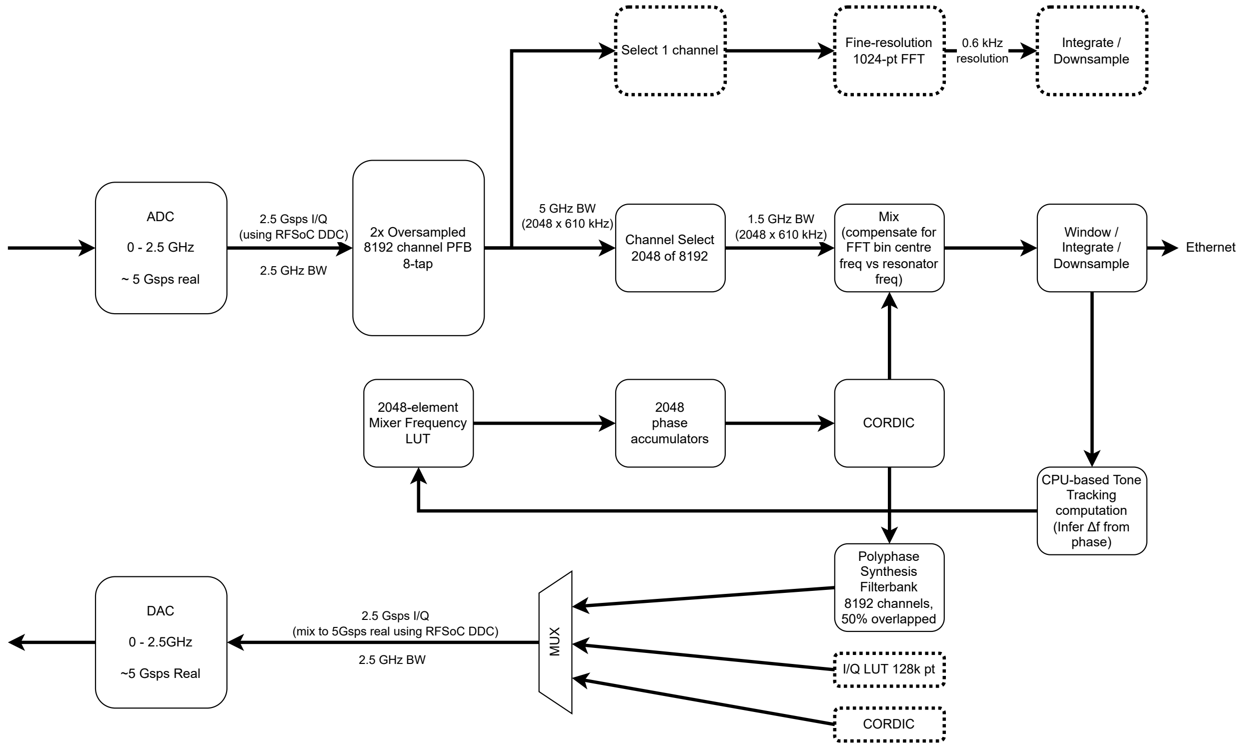
Select 1 channel
Fine-resolution 1024-pt FFT
0.6 kHz resolution
Integrate / Downsample
ADC
0 - 2.5 GHz
~ 5 Gsps real
2.5 Gsps I/Q (using RFSoC DDC)
2.5 GHz BW
2x Oversampled 8192 channel PFB 8-tap
5 GHz BW (2048 x 610 kHz)
Channel Select 2048 of 8192
1.5 GHz BW (2048 x 610 kHz)
Mix (compensate for FFT bin centre freq vs resonator freq)
Window / Integrate / Downsample
Ethernet
2048-element Mixer Frequency LUT
2048 phase accumulators
CORDIC
CPU-based Tone Tracking computation (Infer Δf from phase)
Polyphase Synthesis Filterbank 8192 channels, 50% overlapped
DAC
0 - 2.5GHz
~5 Gsps Real
2.5 Gsps I/Q (mix to 5Gsps real using RFSoC DDC)
2.5 GHz BW
MUX
I/Q LUT 128k pt
CORDIC

Fig. 2.1: A single SOUK MKID readout pipeline. Shown in dotted-boxes are processing features which are not required in production but may be useful for testing in a lab environment. Such features may be shared between different pipelines to save processing resources.

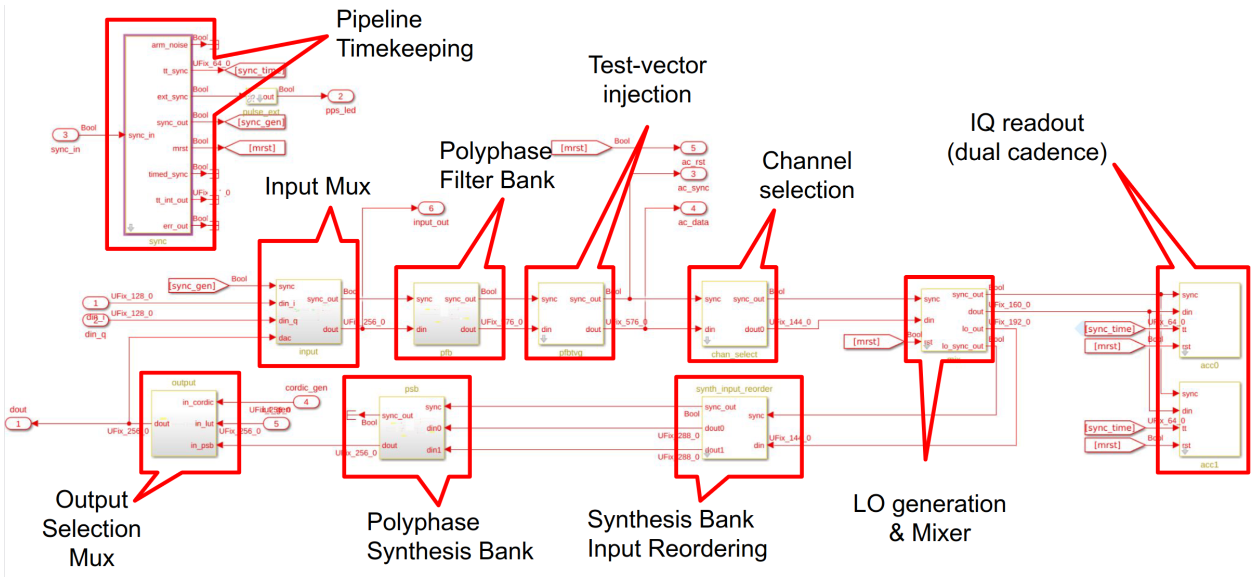

Fig. 2.2: The Simulink source code for a single SOUK MKID readout pipeline.

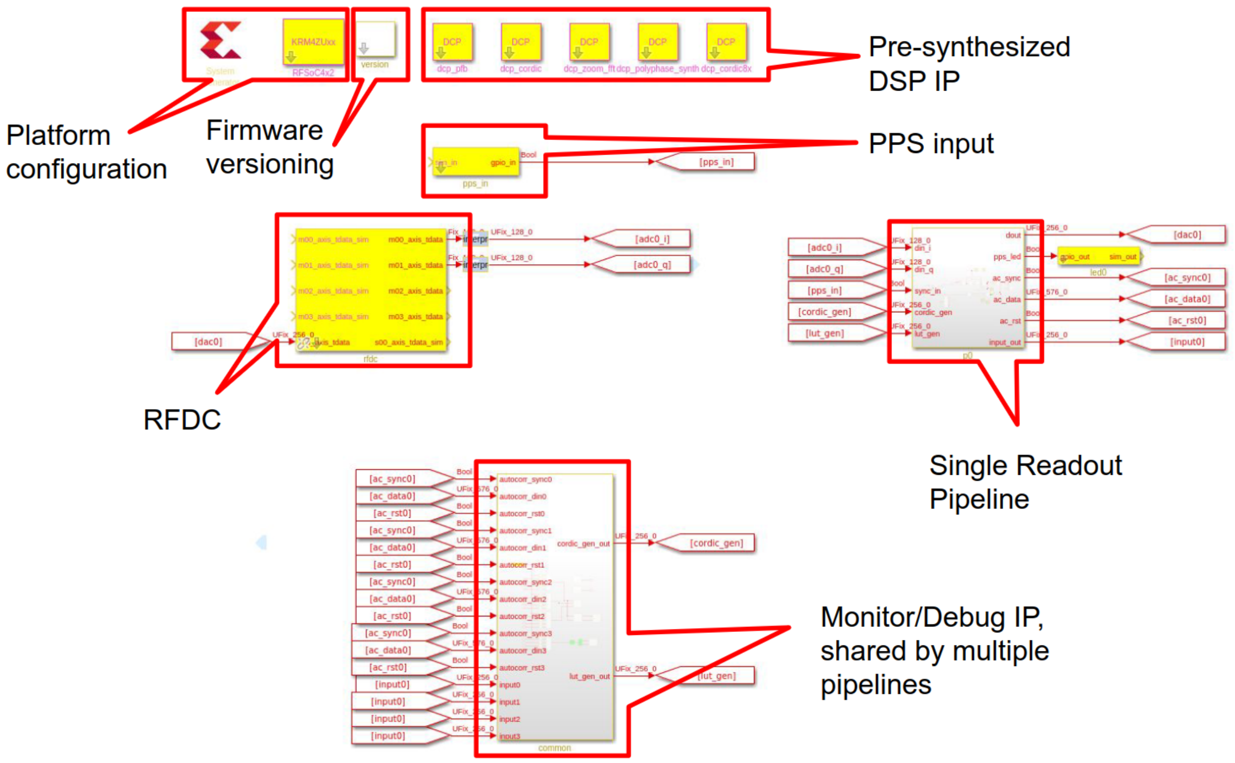

Fig. 2.3: The top-level Simulink source code for a firmware design capable of supporting multiple readout pipelines. A single RFDC instance may feed multiple pipelines. Common testing/monitoring functionality is shared between pipelines to save processing resources.

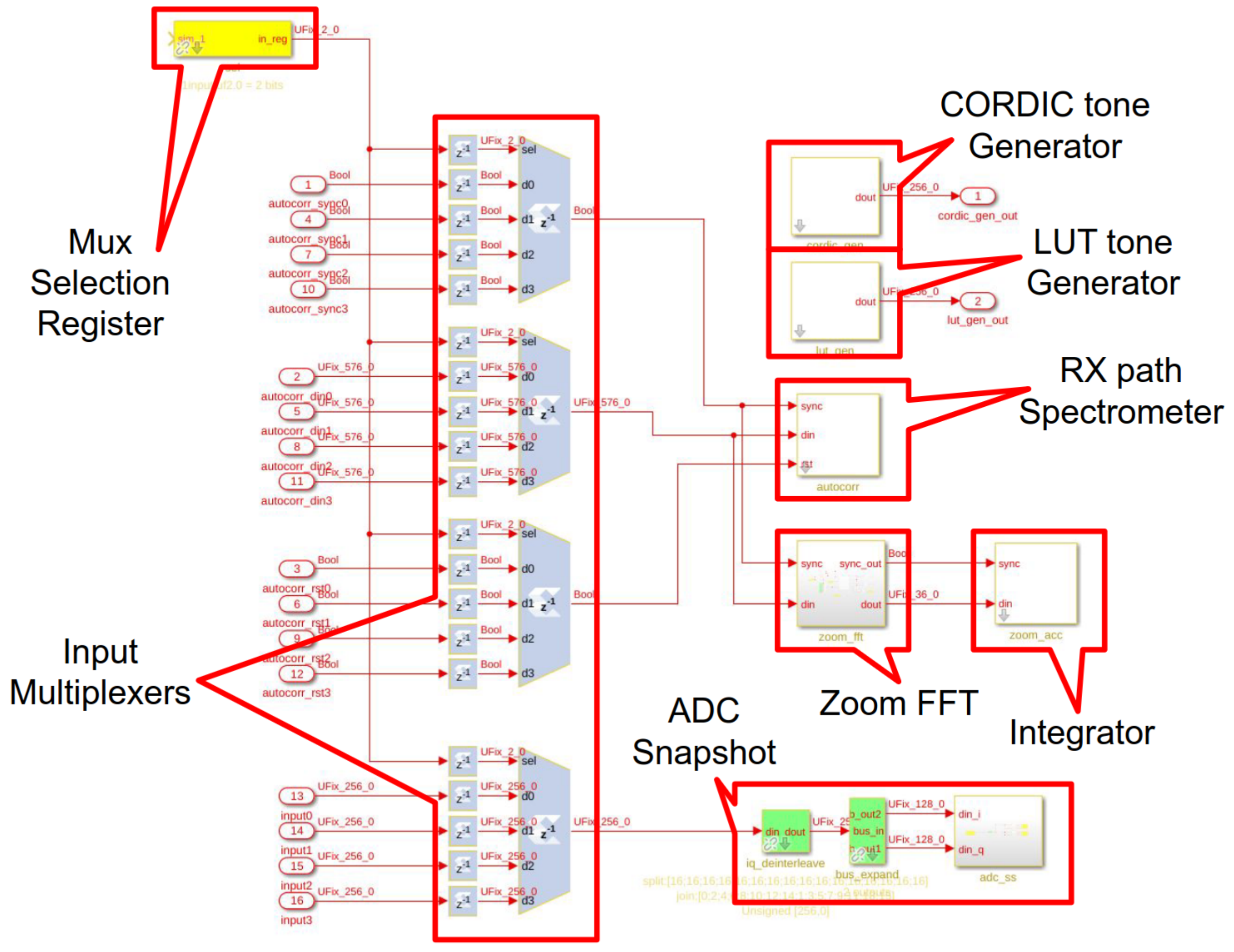

Fig. 2.4: The Simulink source code for processing functionality shared between pipelines. This includes readout of ADC snapshots, pre-mixer spectrometer powers, and high-resolution "zoom" spectra.

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 3 for a full software API description

```
# Import the SNAP2 F-Engine library
from souk_mkid_readout import SoukFirmwareReadout

# Instantiate an SoukFirmwareReadout instance, connecting to a board with
# hostname 'my_zcu111'
f = SoukFirmwareReadout('my_rfsoc_board', config_file='my_config_file.yaml')

# Program a board
f.program() # Load whatever firmware was listed in the config file

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

2.1.2 Block Descriptions

Each block in the firmware design can be controlled using an API described in Section 3.

CHAPTER THREE

CONTROL INTERFACE

3.1 Overview

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

3.2 `SoukMkidReadout` Python Interface

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

```
# Import the RFSoC F-Engine library
from souk_mkid_readout import SoukMkidReadout

# Instantiate a SoukMkidReadout instance to a board with
# hostname 'my_zcu111'
f = SoukMkidReadout('my_zcu111', configfile='my_config.yaml')

# Program a board (if it is not already programmed)
# and initialize all the firmware blocks
if not f.fpga.is_programmed():
  f.program() # Load whatever firmware is specified in the configuration file
  # Initialize firmware blocks
  f.initialize()

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

# Print available block names
print(sorted(f.blocks.keys()))
# Returns (eg):
# ['adc_snapshot', 'rfdc', 'input', 'autocorr', 'pfb', 'pfbtvg', 'chanreorder', 'mix',
# 'gen_lut', 'gen_cordic', 'output', 'accumulator0', 'accumulator1']
```

(continues on next page)

(continued from previous page)

```
# Grab some ADC data from the ADC
adc_data = f.adc_snapshot.get_adc_snapshot()
```

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

3.2.1 Top-Level Control

The Top-level `SoukMkidReadout` 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 LO settings.

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

`class souk_mkid_readout.`**`SoukMkidReadout`**(*host*, *fpgfile=None*, *configfile=None*, *logger=None*, *pipeline_id=0*, *local=False*)

A control class for SOUK MKID Readout firmware on a single board

Parameters

- **`host`** (*`str`*) – Hostname/IP address of FPGA board
- **`fpgfile`** (*`str`*) – Path to .fpg file running on the board
- **`configfile`** (*`str`*) – Path to configuration YAML file for system
- **`logger`** (*`logging.Logger`*) – Logger instance to which log messages should be emitted.
- **`pipeline_id`** (*`int`*) – Pipeline number within a single RFSoC board.
- **`local`** (*`bool`*) – If True, use local memory accesses rather than katcp. Only works as root!

`configfile`

configuration YAML file

`fpgfile`

fpgfile currently in use

`get_status_all`()

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

Returns

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

`hostname`

hostname of FPGA board

`initialize`(*read_only=False*)

Call the ``initialize` methods of all underlying blocks, then optionally issue a software global reset.

Parameters

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

`is_connected()`

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

Return type
bool

`logger`
Python Logger instance

`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 FPGA.

Parameters
`fpgfile` (*`str`*) – The .fpg file to be loaded. Should be a path to a valid .fpg file. If None is given, *self.fpgfile* will be loaded. If this is None, RuntimeError is raised

`reset_psb_outputs()`
Zero out all synthesis bank outputs.

`set_multi_tone`(*freqs_hz*, *phase_offsets_rads=None*, *amplitudes=None*, *los=['rx', 'tx']*)
Configure both TX and RX paths for `i` tones at frequencies `freqs_hz[i]`. Disables all tones except those provided.

Parameters

- **`freqs_hz`** (*`list of float`*) – Tone frequencies, in Hz.
- **`phase_offsets_rads`** (*`list of float`*) – Phase offset of tones, in radians. If none is provided, offsets of 0 are used.
- **`amplitudes`** (*`list of float`*) – Relative amplitude of tones, provided as a list of floats between 0 and 1. If none is provided, amplitudes of 1.0 are used.
- **`los`** (*`list`*) – List of LOs to write to. Can be [‘rx’], [‘tx’] or [‘rx’, ‘tx’]

`set_output_psb_scale`(*nshift*, *scale=1.0*, *check_overflow=True*)
Set the PSB to scale down by 2^nshift in amplitude.

Parameters

- **`nshift`** (*`int`*) – Number of shift down stages in the PSB FFTs
- **`scale`** (*`float`*) – Post PSB scaling factor
- **`check_overflow`** (*`bool`*) – If True, warn about PSB overflow before returning.

Returns
If check_overflow is set, return FENG_OK if no overflows are detected, of FENG_ERROR otherwise. Return FENG_OK if check_overflow is not set.

Return type
int

set_tone(*tone_id*, *freq_hz*, *phase_offset_rads=0.0*, *amp=1.0*)

Configure both TX and RX paths for a tone at frequency `freq_hz` with ID `tone_id`.

Parameters

- **tone_id** (*int*) – Index number of tone to set
- **freq_hz** (*float*) – Tone frequency, in Hz. Or, use `None` to disable this tone index.
- **phase_offset_rads** (*float*) – Phase offset of tone, in radians.
- **amp** (*float*) – Tone amplitude, (<=1.0)

use_dual_dac()

Use dual DAC outputs, with 1 output for “even” frequency tones, and another for “odd”.

use_single_dac()

Use single DAC output, with “even” and “odd” frequency tones summed.

3.2.2 Common Pipeline Infrastructure

The `common` block interface controls the multiplexors feeding logic shared between multiple DSP pipelines.

class souk_mkid_readout.blocks.common.**Common**(*host*, *name*, *logger=None*, *ninput=4*)

Instantiate a control interface for the “common” firmware block, which is common to all DSP pipelines.

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.
- **ninput** (*int*) – Number of inputs this block can handle.

get_input()

Get the currently selected input index.

Returns
Input ID

Return type
int

get_status()

Get status and error flag dictionaries.

Status keys:

- input_select (int) : Currently selected input ID

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 input select mux to input 0. if True, do nothing.

`set_input`(*n*)

Set the currently selected input index to `n`.

Parameters
`n` (*int*) – Input index to select

3.2.3 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 souk_mkid_readout.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.

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

Return type
int

`get_connected_antname`()

Fetch the connected antenna name.

Returns
The name of the connected antennna.

Return type
str

`get_firmware_type`()

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

Returns
Firmware type

Return type
str

`get_firmware_version()`

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

Returns
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 Hz

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_supported (bool) : True if the running firmware is supported by this software. False (and flagged as an error) otherwise.
- 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()`

Check to see if a board is programmed. If the Katcp command *listdev* fails, assume that it isn't

Returns
True if programmed, False otherwise.

Return type
bool

`set_connected_antname`(*antname*)

Set the connected antenna name.

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

3.2.4 Timing Control

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

class `souk_mkid_readout.blocks.sync.Sync`(*host*, *name*, *clk_hz=None*, *sync_delay=1*, *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.
- **sync_delay** (*int*) – The initial delay to load for the delayed sync output in FPGA clock cycles
- **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.

`count_ext()`

Returns
Number of external sync pulses received.

Rtype int

`disable_error_flag()`

Disable error flag.

`enable_error_flag()`

Enable error flag.

`get_delay()`

Get the delay of the delayed sync output, in FPGA clock cycles

Returns
Delay in FPGA clock cycles

Return type
int

`get_drift()`

Get the drift observed between the time determined by a counter reset on the last *arm* call, and the internal telescope time, updated with *update_internal_time* based on external synchronization pulses.

If the drift measurement changes during read, throw a RuntimeError.

Returns
drift, in FPGA clock cycles

Return type
int

`get_error_count()`

Returns
Number of sync errors.

Return type
int

`get_pipeline_latency()`

Get the difference in arrival time of a sync pulse at the start of the RX chain and at the end of the TX chain, in units of FPGA clock cycles. Depending on the *mix* block signal sharing settings, this is either the total latency (when the TX pipeline sync is shared with the RX pipeline sync) or is the residual skew when the RX pipeline sync is a delayed copy of the TX sync.

Returns
Sync time difference, in FPGA clock cycles

Return type
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.

- sync_delay (int) : The number of FPGA clock cycles between the RX and TX sync pulses.
- drift (int) : The number of FPGA clock cycles of drift measured between the last SYNC and PPS

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_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.

Return type
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.

`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.

`set_delay`(*delay*)

Set the delay of the delayed sync output

Parameters
`delay` (*`int`*) – Delay in FPGA clock cycles

`set_sync_active_high()`

Set the sync pulse to active on a positive edge.

`set_sync_active_low()`

Set the sync pulse to active on a negative edge.

`sw_sync(`*wait=True*`)`

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

Parameters

`wait` (*bool*) – If True, wait 50ms for a sync to propagate before returning.

`update_internal_time(`*clk_hz=None*, *sync_period=None*, *offset_ns=0.0*, *sync_clock_factor=1*`)`

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

Parameters

- `clk_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.
- `sync_period` – Sync pulse period, in FPGA clock ticks. If None, read period from FPGA counters.
- `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.

Return type

int

`wait_for_sync()`

Block until a sync has been received.

3.2.5 RFDC Control

The `Rfdc` control interface allows control of the RFSoC's ADCs and DACs.

`class souk_mkid_readout.blocks.rfdc.Rfdc(`*host*, *name*, *logger=None*, *lmkfile=None*, *lmxfile=None*, *pipeline_id=0*`)`

Instantiate a control interface for an RFDC firmware 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.
- `lmkfile` (*str*) – LMK configuration file to load to board's PLL chip

- **lmxfile** (*str*) – LMX configuration file to load to board's PLL chip
- **pipeline_id** (*int*) – Index of this pipeline. Used to figure out which control port names are associated with this pipeline's ADC/DAC channels.

`get_lo`(*adc_sample_rate_hz*, *tile*, *block*)

Get current LO frequency.

Parameters

- **adc_sample_rate_hz** (*float*) – ADC sample rate in Hz
- **tile** (*int*) – Zero-indexed tile ID of this ADC.
- **block** (*int*) – Zero-indexed block ID of this ADC.

Returns
LO frequency in Hz

Return type
float

`get_rts_flags`()

Read RTS flags and return as a dictionary.

Returns
Dictionary of status flags. See *get_status* for descriptions

`get_status`()

Get status and error flag dictionaries. See PG269 for RTS port definitions.

Status keys:

- lmkfile (str) : The name of the LMK configuration file being used.
- lmxfile (str) : The name of the LMX configuration file being used.
- **rts_over_range (bool)**
  [True if signal has exceeded full scale ADC input.] Must be cleared manually.
- rts_over_threshold1 (bool) : Signal amplitude is above programmable threshold 1.
- rts_over_threshold2 (bool) : Signal amplitude is above programmable threshold 2.
- **rts_over_voltage (bool)**
  [True if signal has "far exceeded" input range.] Must be cleared manually.
- rts_cm_over_voltage (bool) : True if input signal common mode voltage is too high for safe operation.
- rts_cm_under_voltage (bool) True if input signal common mode voltage is too low for safe operation.

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, initialize the RFDC core and PLL chips. If True, do nothing.

`reset_rts_flags`(*over_range=True*, *over_voltage=True*)

Reset the RTS error flags.

Parameters

- **over_range** (*bool*) – If True, reset the over_range flag
- **over_voltage** (*bool*) – If True, reset the over_voltage flag

3.2.6 ADC Snapshot

The `AdcSnapshot` control interface allows the capture of raw ADC samples.

`class souk_mkid_readout.blocks.adc_snapshot.AdcSnapshot`(*host*, *name*, *logger=None*)

`get_adc_snapshot`()

Same as *get_snapshot* for backwards compatibility

`get_snapshot`()

Get a data snapshot.

Returns

numpy array of complex valued ADC samples

Return type

numpy.ndarray

`plot_adc_snapshot`(*nsamples=None*, *signals=None*)

Same as *plot_snapshot* for backwards compatibility

`plot_adc_spectrum`(*db=False*, *signals=None*)

Same as *plot_spectrum* for backwards compatibility

`plot_snapshot`(*nsamples=None*, *signals=None*)

Plot a data snapshot.

Parameters

- **nsamples** (*int*) – If provided, only plot this many samples
- **signals** (*list of int*) – List of signal IDs to plot. E.g., [0] to plot only the first signal. If None, plot everything.

`plot_spectrum`(*db=False*, *signals=None*)

Plot a power spectrum of a data snapshot using a simple FFT.

Parameters

- **db** (*bool*) – If True, plot in dBs, else linear.
- **signals** (*list of int*) – List of signal IDs to plot. E.g., [0] to plot only the first signal. If None, plot everything.

3.2.7 Input Control

The `Input` control interface controls the multiplexors at the start of the RX pipeline.

`class souk_mkid_readout.blocks.input.`**`Input`**`(`*host*, *name*, *logger=None*`)`

`disable_loopback``()`

Set pipeline to feed pipeline from ADC inputs

`enable_loopback``()`

Set pipeline to internally loop-back DAC stream into ADC.

`get_status``()`

Get status and error flag dictionaries.

Status keys:

- loopback (book) : True is system is in internal loopback mode. If True this is flagged with "WARNING".

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, disable loopback mode. If True, do nothing.

`loopback_enabled``()`

Get the current loopback state.

Returns

True if internal loopback is enabled. False otherwise.

Return type

bool

3.2.8 PFB Control

The `Pfb` interface controls the RX channelizers.

`class souk_mkid_readout.blocks.pfb.`**`Pfb`**`(`*host*, *name*, *logger=None*, *fftshift=4294967295*`)`

`get_fftshift``()`

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".
- fftshift (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, set the FFT shift to the default value, and reset the overflow count. If True, do nothing.

`reset_overflow_count()`

Reset overflow counter.

`set_fftshift(`*shift*`)`

Set the FFT shift schedule.

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

3.2.9 PFB TVG Control

The `PfbTvg` interface allows test vectors to be injected into the RX pipeline, after the PFB block.

`class souk_mkid_readout.blocks.pfbtvg.PfbTvg(`*host*, *name*, *n_inputs=2*, *n_chans=4096*, *n_serial_inputs=1*, *n_rams=2*, *n_samples_per_word=4*, *sample_format='h'*, *logger=None*`)`

Instantiate a control interface for a post-PFB 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_rams` (*int*) – Number of independent bram blocks per input. If 0, block has no RAMs, and just contains counter-based test vectors.
- `n_samples_per_word` (*int*) – Number of complex samples per word in RAM
- `n_chans` (*int*) – Number of frequency channels.
- `sample_format` (*str*) – Struct type code (eg. 'h' for 16-bit signed) for each of the real/imag parts of the TVG data samples.

`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*)

Read the test vector loaded to an ADC input.

Parameters
`input` (*int*) – Index of input from which test vectors should be read.

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 16-bit integer format, and may be complex.

3.2.10 Auto-correlation Control

The `Autocorr` interface controls an spectral-power integrator which can be used to accumulate spectra after the RX PFB.

class `souk_mkid_readout.blocks.autocorr.`**AutoCorr**(*host*, *name*, *acc_len=32768*, *logger=None*, *n_chans=4096*, *n_signals=64*, *n_parallel_streams=8*, *n_cores=4*, *is_descrambled=False*, *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.
- **is_descrambled** (*bool*) – If False, apply descrable map to channel ordering on read.
- **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.

Returns
Current accumulation count

Return type
int

`get_acc_len()`

Get the currently loaded accumulation length in units of spectra.

Returns
Current accumulation length

Return type
int

`get_freqs`(*adc_srate_hz*, *lo_hz=0.0*)

Get the center frequencies of each spectral channel.

Parameters

- `adc_srate_hz` (`float`) – ADC sample rate in Hz.
- `lo_hz` (`float`) – LO frequency, in Hz, implemented within the ADC.

`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 kernel 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*, *adc_srate_hz=None*, *lo_hz=0.0*)

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.
- **adc_srate_hz** (*float*) – ADC sample rate in Hz. If provided, plot with an appropriate frequency scale on the X-axis.
- **lo_hz** (*float*) – LO frequency, in Hz, implemented within the ADC.

Returns
matplotlib.Figure

`plot_spectra`(*signal_block=0*, *db=True*, *show=True*, *filter_ksize=None*, *adc_srate_hz=None*, *lo_hz=0.0*)

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.
- **adc_srate_hz** (*float*) – ADC sample rate in Hz. If provided, plot with an appropriate frequency scale on the X-axis.
- **lo_hz** (*float*) – LO frequency, in Hz, implemented within the ADC.

Returns
matplotlib.Figure

`set_acc_len`(*acc_len*)

Set the number of spectra to accumulate.

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

3.2.11 Channel Sub-Select Control

The `ChanReorderMultiSample` control interface allows a subset of PFB channels to be selected for further processing. A similar interface class – `ChanReorderMultiSampleIn` – is used to control the assignment of LO tones to PSB channels.

`class souk_mkid_readout.blocks.chanreorder.`**`ChanReorderMultiSample`**`(`*host*, *name*, *n_serial_chans_in=512*, *n_parallel_chans_in=16*, *n_parallel_samples=4*, *support_zeroing=True*, *default_descramble_input=False*, *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_serial_chans_in`** (*`int`*) – Number of serial channels input to the reorder
- **`n_parallel_chans_in`** (*`int`*) – Number of parallel channels input to the reorder
- **`n_parallel_samples`** (*`int`*) – Number of parallel samples output
- **`default_descramble_input`** (*`bool`*) – If True, the default is to compensate for an upstream scrambled FFT channel order.
- **`support_zeroing`** (*`bool`*) – If True, allow the use of channel index `-1` to mean "zero out this channel"

`get_channel_outmap``(`*descramble_input=None*`)`

Read the currently loaded reorder map.

Parameters

`descramble_input` (*`bool`*) – If True, descramble the recovered channel map. If not provided, descramble if the _descramble_default attribute is True.

Returns

The reorder map currently loaded. Entry *i* in this map is the channel number which emerges in the `i`th output position.

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_channel_outmap``(`*outmap*, *descramble_input=None*`)`

Remap the channels such that the channel outmap[i] emerges out of the reorder map in position i.

The provided map must be *self.n_chans_out* elements long, else *ValueError* is raised

Parameters

- **outmap** (*list of int*) – The outmap to which data should be mapped. I.e., if *outmap[0] = 16*, then the first channel out of the reorder block will be channel 16.
- **descramble_input** (*bool*) – If True, descramble the provided channel map. If not provided, descramble if the _descramble_default attribute is True.

`set_single_channel`(*outidx*, *inidx*, *descramble_input=None*)

Set output channel number `outidx` to input number `inidx`. Do this by reading the total channel map, modifying a single entry, and writing back.

Example usage:
# Set the first channel out of the reorder to 33 `` `set_single_channel(0, 33) ``

Parameters

- **outidx** (*int*) – Index of output channel to set.
- **inidx** (*int*) – Input channel index to select.
- **descramble_input** (*bool*) – If True, descramble the provided channel map. If not provided, descramble if the _descramble_default attribute is True.

`class souk_mkid_readout.blocks.chanreorder.`**`ChanReorderMultiSampleIn`**(*host*, *name*, *n_serial_chans_out=512*, *n_parallel_chans_out=16*, *n_parallel_samples=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_serial_chans_out** (*int*) – Number of serial channels output from the reorder
- **n_parallel_chans_out** (*int*) – Number of parallel channels output from the reorder
- **n_parallel_samples** (*int*) – Number of parallel samples input

`get_channel_outmap`()

Read the currently loaded reorder map.

Returns
The reorder map currently loaded. Entry *i* in this map is the channel number which emerges in the `i`th output position.

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_channel_outmap`(*outmap*)

Remap the channels such that the channel outmap[i] emerges out of the reorder map in position i.

The provided map must be *self.n_chans_out* elements long, else *ValueError* is raised

Parameters

`outmap` (*list of int*) – The outmap to which data should be mapped. I.e., if *outmap[0] = 16*, then the first channel out of the reorder block will be channel 16.

`set_single_channel`(*outidx*, *inidx*)

Set output channel number `outidx` to input number `inidx`. Do this by reading the total channel map, modifying a single entry, and writing back.

Example usage:

# Set the first channel out of the reorder to 33 `` ` ``set_single_channel(0, 33)``

Parameters

- **`outidx`** (*int*) – Index of output channel to set.
- **`inidx`** (*int*) – Input channel index to select.

3.2.12 Zoom FFT Control

The `ZoomPfb` control interface controls a second stage PFB which operates on a single RX PFB channel.

`class souk_mkid_readout.blocks.zoom_pfb.ZoomPfb`(*host*, *name*, *logger=None*, *fftshift=4294967295*)

`get_channel`()

Get the currently selected channel.

Returns

Channel currently selected

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”.
- fftshift (str) : Currently loaded FFT shift schedule, formatted as a binary string, prefixed with “0b”.
- channel (int) : Currently selected input channel

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, set the FFT shift to the default value, reset the overflow count, and set the channel selection to 0. If True, do nothing.

`set_channel`(*chan*)

Set the channel to select

Parameters

`chan` (*`int`*) – Channel to select

3.2.13 Zoom FFT Accumulator Control

The `Accumulator` control interface controls a accumulation of PFB spectral power.

class `souk_mkid_readout.blocks.accumulator.`**`Accumulator`**(*host*, *name*, *logger=None*, *acc_len=32768*, *n_chans=4096*, *n_parallel_chans=8*, *n_parallel_samples=1*, *is_complex=True*, *dtype='>i4'*, *has_dest_ip=False*)

Instantiate a control interface for an Accumulator Block. This provides a vector accumulation of post-FFT 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.
- **`acc_len`** (*`int`*) – Accumulation length initialization value, in spectra.
- **`n_chans`** (*`int`*) – Number of frequency channels.
- **`n_parallel_chans`** (*`int`*) – Number of chans processed by the firmware module in parallel.
- **`n_parallel_samples`** (*`int`*) – Number of samples processed by the firmware module in parallel.
- **`is_complex`** (*`Bool`*) – If True, block accumulates complex-valued data.
- **`dtype`** (*`str`*) – Data type string (as recognised by numpy's *frombuffer* method) for accumulated data. If data are complex, this is the data type of one of a single real/imag component.

`get_acc_cnt`()

Get the current accumulation count.

Returns

Current accumulation count

Return type

int

`get_acc_len`()

Get the currently loaded accumulation length in units of spectra.

Returns

Current accumulation length

Return type

int

`get_new_burst`()

Trigger the collection of a new burst of data, then read it.

Returns

array of *self.n_chans* complex-values, representing consecutive samples in a burst.

Return type
numpy.ndarray

`get_new_spectra`(*gpio_count=[]*, *get_tt=False*)

Wait for a new accumulation to be ready then read it.

Parameters
`gpio_count` – List of GPIO counter registers to return with the accumulator data. E.g., [0,1,3] will return counters for pulse edges on GPIOs 0, 1, and 3.

Get_tt
If True, return timestamp corresponding to last sample of accumulation.

Returns
spectra_data, gpio_counts, timestamp, spectra_data is an array of *self.n_chans* complex-values. If gpio_count is not an empty list gpio_values is a list of the same length as gpio_count. Otherwise gpio_count is None If get_tt, timestamp is the accumulation timestamp. Otherwise it is None

Return type
numpy.ndarray[, gpio_counters]

`get_status`()

Get status and error flag dictionaries.

Status keys:

- acc_len (int) : Currently loaded accumulation length in number of spectra.
- burst_mode (bool) : True if the accumulator is in burst mode, 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, 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.

`is_burst_mode`()

Query whether the block is in burst mode.

Returns
True is in burst mode, else False

Return type
bool

`plot_spectra`(*power=True*, *db=True*, *show=True*, *fftshift=False*, *sample_rate_hz=None*)

Plot the spectra of all signals in a single signal_block, with accumulation length divided out

Parameters

- `power` (*bool*) – If True, plot power, else plot complex
- `db` (*bool*) – If True, plot 10log10(power). Else, plot linear.

- **show** (*bool*) – If True, call matplotlib's *show* after plotting
- **fftshift** (*bool*) – If True, fftshift data before plotting.
- **sample_rate_hz** (*float*) – Effective FFT input sampling rate, in Hz. If provided, generate an appropriate frequency axis

Returns
matplotlib.Figure

read_gpio_counter(*n*)

Read GPIO counter n

Parameters
n (*int*) – GPIO counter to read

set_acc_len(*acc_len*)

Set the number of spectra to accumulate.

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

set_burst_mode(*is_burst*)

Set or unset burst mode.

Parameters
is_burst (*bool*) – True to enter burst mode, False to exit.

3.2.14 Mixer Control

The `Mixer` control interface allows configuration of multi-channel CORDIC-based LO generators.

class souk_mkid_readout.blocks.mixer.**Mixer**(*host*, *name*, *n_chans=4096*, *n_upstream_chans=8192*, *upstream_oversample_factor=2*, *n_parallel_chans=4*, *phase_bp=31*, *phase_offset_bp=31*, *n_scale_bits=8*, *n_ri_step_bits=16*, *n_phase_slots=4096*, *logger=None*)

Instantiate a control interface for a Mixer 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 this block processes
- **n_upstream_chans** (*int*) – Number of channels in the upstream PFB prior to downselection
- **upstream_oversample_factor** (*int*) – Oversampling factor of upstream system. This, with the number of upstream channels, should allow this block to figure out how wide channels are.
- **n_parallel_chans** (*int*) – Number of channels this block processes in parallel
- **phase_bp** (*int*) – Number of phase fractional bits
- **n_ri_step_bits** (*int*) – Number of bits in each of the real/image per-sample rotation values.

`disable_power_mode()`

Use phase rotation, rather than power.

`enable_power_mode()`

Instead of applying a phase rotation to the data streams, calculate their power.

`get_current_buffer()`

Get the current ping-pong buffer index (either 0 or 1) used for reading.

Returns
Buffer index

Return type
int

`get_phase_offset`(*chan*, *lo='rx'*)

Get the currently loaded phase increment being applied to channel *chan*.

Parameters
`lo` (*str*) – Which LO to read. 'rx' or 'tx'

Returns
(phase_step, phase_offset, scale) A tuple containing the phase increment (in radians) being applied to channel *chan* on each successive sample, the start phase in radians, and the scale factor being applied to this channel.

Return type
float

`get_phase_switch_pattern`(*lo='rx'*)

Get the currently loaded phase switch pattern.

Parameters
`lo` (*str*) – Which LO to read. 'rx' or 'tx'

Returns
(phase pattern, spectra_per_step) A tuple containing the phase pattern as an array of 1s and 0s, and the number of spectra to which each phase is applied.

Return type
(numpy.ndarray, int)

`initialize`(*read_only=False*)

Initialize the block.

Parameters
`read_only` (*bool*) – If True, this method is a no-op. If False, set this block to phase rotate mode, but initialize with each channel having zero phase increment.

`is_power_mode()`

Get the current block mode.

Returns
True if the block is calculating power, False if it is applying phase rotation.

Return type
bool

`set_amplitude_scale`(*chan*, *scale=1.0*, *los=['rx', 'tx']*, *next_buf=False*)

Apply an amplitude scaling <=1 to an output channel.

Parameters

- **chan** (*int*) – The channel index to which this phase-rate should be applied
- **scaling** (*float*) – optional scaling (<=1) to apply to the output tone amplitude.
- **los** (*list*) – List of LOs to write to. Can be ['rx'], ['tx'] or ['rx', 'tx']
- **next_buf** (*bool or int*) – If False, write directly to the buffer currently being read by the firmware. If True, write to the next buffer, leaving the new parameters inactive until the buffer is switched (eg. by *switch_current_buffer*). If 0 or 1, write to that buffer.

set_chan_freq(*chan*, *freq_offset_hz=None*, *phase_offset=0*, *sample_rate_hz=2500000000*, *next_buf=False*)

Set the frequency of output channel *chan*.

Parameters

- **chan** (*int*) – The channel index to which this phase-rate should be applied
- **freq_offset_hz** (*float*) – The frequency offset, in Hz, from the channel center. If None, disable this oscillator.
- **phase_offset** – The phase offset at which this oscillator should start in units of radians.
- **sample_rate_hz** (*float*) – DAC sample rate, in Hz
- **next_buf** (*bool or int*) – If False, write directly to the buffer currently being read by the firmware. If True, write to the next buffer, leaving the new parameters inactive until the buffer is switched (eg. by *switch_current_buffer*). If 0 or 1, write to that buffer.

set_current_buffer(*buf_id*)

Set the current ping-pong buffer index (either 0 or 1) used for reading.

Returns
Buffer index

Return type
int

set_freqs(*freqs_hz*, *phase_offsets*, *scaling=1.0*, *sample_rate_hz=2500000000*, *los=['rx', 'tx']*, *next_buf=False*)

Configure the amplitudes, phases, and frequencies of multiple tones.

Parameters

- **freqs_hz** (*numpy.ndarray*) – The frequencies, in Hz, to emit.
- **phase_offsets** (*np.ndarray*) – The phase offsets at which oscillators should start, in units of radians.
- **scaling** (*np.ndarray*) – optional scaling (<=1) to apply to the output tone amplitudes. If a single number, apply this scale to all tones.
- **sample_rate_hz** (*float*) – DAC sample rate, in Hz
- **los** (*list*) – List of LOs to write to. Can be ['rx'], ['tx'] or ['rx', 'tx']
- **next_buf** (*bool or int*) – If False, write directly to the buffer currently being read by the firmware. If True, write to the next buffer, leaving the new parameters inactive until the buffer is switched (eg. by *switch_current_buffer*). If 0 or 1, write to that buffer.

set_phase_step(*chan*, *phase=None*, *phase_offset=0.0*, *los=['rx', 'tx']*, *next_buf=False*)

Set the phase increment to apply on each successive sample for channel *chan*.

Parameters

- **chan** (*int*) – The channel index to which this phase-rate should be applied
- **phase** (*float*) – The phase increment to be added each successive sample in units of radians. If None, disable this oscillator.
- **phase_offset** – The phase offset at which this oscillator should start in units of radians.
- **los** (*list*) – List of LOs to write to. Can be [‘rx’], [‘tx’] or [‘rx’, ‘tx’]
- **next_buf** (*bool*) – If False, write directly to the buffer currently being read by the firmware. If True, write to the next buffer, leaving the new parameters inactive until the buffer is switched (eg. by *switch_current_buffer*). If 0 or 1, write to that buffer.

set_phase_switch_pattern(*pattern*, *spectra_per_step*, *los=['rx', 'tx']*, *n_blank=0*)

Set the phase switching pattern.

For a pattern (eg) [1, 0] The first *spectra_per_step* spectra will have LOs phase inverted, the next *spectra_per_step* spectra will not be inverted, and then the pattern will reset.

The maximum number of slots in the pattern is *self.n_phase_slots*.

Parameters

- **pattern** (*list*) – List or array of ones and zeros. One indicates that phase should be inverted in this slot. 0 indicates phase should not be inverted.
- **spectra_per_step** (*int*) – The number of spectra to which each element of pattern should be applied.
- **los** (*list*) – List of LOs to modify. Can be [‘rx’], [‘tx’] or [‘rx’, ‘tx’]
- **n_blank** (*int*) – Number of spectra to blank before and after a phase switch transition

switch_current_buffer()

Flip the current buffer by reading *get_current_buffer*, swapping 0 <-> 1 and writing to *set_current_buffer*

3.2.15 Windowed Accumulator Control

The `WindowedAccumulator` control interface allows configuration of an accumulator which provides runtime-programmable windowing of input data.

class souk_mkid_readout.blocks.accumulator.**WindowedAccumulator**(*host*, *name*, *logger=None*, *acc_len=32768*, *n_chans=4096*, *n_parallel_chans=8*, *n_parallel_samples=1*, *is_complex=True*, *dtype='>i4'*, *has_dest_ip=False*, *include_window=False*, *window_bp=14*, *window_dtype='>i2'*, *window_n_points=2048*, *max_reuse_bits=9*)

Instantiate a control interface for a Windowed Accumulator Block. This block applies a window to data prior to a vector accumulation of post-FFT 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.
- **acc_len** (*int*) – Accumulation length initialization value, in spectra.
- **n_chans** (*int*) – Number of frequency channels.
- **n_parallel_chans** (*int*) – Number of chans processed by the firmware module in parallel.
- **n_parallel_samples** (*int*) – Number of samples processed by the firmware module in parallel.
- **is_complex** (*Bool*) – If True, block accumulates complex-valued data.
- **dtype** (*str*) – Data type string (as recognised by numpy's *frombuffer* method) for accumulated data. If data are complex, this is the data type of one of a single real/imag component.

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.

get_window(*n=None*)

Get the currently loaded window coefficients, duplicating as required to match the behaviour of firmware.

Returns
Vector of coefficients with length matching the number of samples accumulated.

Return type
np.ndarray

get_window_step()

Get log2 of the number of samples stepped through the coefficient vector with each new set of parallel samples.

Returns
log2 step length

Return type
int

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, and set the window function to all ones. If True, do nothing.

set_window(*windfunc=<function ones>*)

Set the filter window.

Parameters
windfunc (*Function*) – A function which returns a vector of coefficients when passed an argument *n* indicating the number of points in the window. E.g. np.ones

`set_window_step`(*n*)

Set how many samples to step through the coefficient vector with each new set of parallel samples.

Parameters
`n` (*int*) – 2^? Number of samples to step through each block of parallel samples. This must be at least the number of parallel samples itself.

3.2.16 PSB Scaling

The `PsbScale` control interface allows configuration of PSB voltage scaling.

`class souk_mkid_readout.blocks.psbscale.PsbScale`(*host*, *name*, *logger=None*)

`get_overflow_count`()

Get the number of overflow events since the last scale factor change.

Returns
Number of overflows

Return type
int

`get_scale`()

Get the scale factor.

Returns
Scale factor

Return type
float

`get_status`()

Get status and error flag dictionaries.

Status keys:

- scale (float) : Current scale factor
- overflow (bool) : True if overflows detected, false otherwise.

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, set scale to 1.0 If True, do nothing.

`set_scale`(*scale*)

Set the scale factor.

Parameters
`scale` (*float*) – Scale factor to be written.

3.2.17 Generator

The `Generator` control interface controls CORDIC- or LUT-based tone generators.

class `souk_mkid_readout.blocks.generator.`**`Generator`**(*host*, *name*, *logger=None*)

`get_cordic_overflows`()

Get the number of overflow events in the CORDIC pipeline since the last phase reset. Return 0 if no CORDIC generators exist.

Returns
Overflows since last reset

Return type
bool

`get_lut_output`(*n*)

Get waveform stored in LUT output *n*.

Parameters
n (*int*) – Which generator to target.

Returns
waveform

Return type
numpy.ndarray

`initialize`(*read_only=False*)

Parameters
read_only (*bool*) – If True, do nothing. If False, reset phase and generator contents

`reset_phase`()

Reset the phase of the output(s).

`set_cordic_output`(*n*, *p*, *amplitude=None*)

Set CORDIC output *n* to increment by phase *p* every sample.

Parameters

- **n** (*int*) – Which generator to target.
- **p** (*float*) – phase increment, in units of radians
- **amplitude** (*float*) – Set the output of amplitude of the CW signal. If not provided, use maximum scale.

`set_lut_output`(*n*, *x*, *scale=True*)

Set LUT output *n* to sample array *x*.

Parameters

- **n** (*int*) – Which generator to target.
- **x** (*list or numpy.array*) – Array (or list) of complex sample values
- **scale** (*bool*) – If True, scale to the maximum possible amplitude range in event of overflow. Otherwise, saturate overflowing values.

`set_output_freq`(*n*, *freq_hz*, *sample_rate_hz=2457600000*, *amplitude=None*, *round_freq=True*, *window=False*)

Set an output to a CW tone at a specific frequency. Frequency specified here does not take into account any digital mixers downstream of this block.

Parameters

- **`n`** (*`int`*) – Which generator to target. Use -1 to mean "all"
- **`freq_hz`** (*`float`*) – Output frequency, in Hz
- **`sample_rate_hz`** (*`float`*) – DAC sample rate, in Hz
- **`amplitude`** (*`float`*) – Set the output of amplitude of the CW signal. If not provided, use maximum scale.
- **`round_freq`** (*`bool`*) – If True, round `freq_hz` to the nearest frequency which can be represented with a `self.n_samples` circular buffer. This option affects only LUT generators.
- **`window`** (*`bool`*) – If True, apply a Hann (a.k.a. Hanning) window to data samples. This option affects only LUT generators.

3.2.18 Output

The `Output` control interface allows configuration of TX output multiplexors.

`class souk_mkid_readout.blocks.output.`**`Output`**(*host*, *name*, *logger=None*)

`get_mode`()

Get the current output mode.

Returns
string describing output mode, eg. "CORDIC"

Return type
str

`get_status`()

Get status and error flag dictionaries.

Status keys:

- mode (str) : 'CORDIC' or 'LUT' or 'PSB'

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 True, do nothing. If False, initialize to LUT mode.

`use_cordic`()

Set output pipeline to use CORDIC generators `use_lut()`

Set output pipeline to use LUT generators

`use_psb()`

Set output pipeline to use Polyphase Synthesis generators

3.2.19 Output Delay

Programmable output delay is controlled via a `Delay` control interface.

`class souk_mkid_readout.blocks.delay.Delay`(*host*, *name*, *logger=None*)

`get_delay()`

Get the delay applied to the output stream, in units of FPGA clocks. Each FPGA clock is 8 DAC clocks. I.e., a system with output bandwidth of 2500 MHz has an FPGA clock of 312.5 MHz.

Returns
Number of FPGA clock cycles of delay

Return type
int

`get_status()`

Get status and error flag dictionaries.

Status keys:

- delay (int) : Number of FPGA clocks of delay applied to output stream

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, set delay to 0. If True, do nothing.

`set_delay`(*n_fpga_clocks*)

Set the delay applied to the output stream, in units of FPGA clocks. Each FPGA clock is 8 DAC clocks. I.e., a system with output bandwidth of 2500 MHz has an FPGA clock of 312.5 MHz.

Parameters
`n_fpga_clocks` (*`int`*) – Number of FPGA clock cycles delay to apply.

INDEX